

CSE 111

Programming with Functions

Brigham Young University - Idaho

Contents

This is an unofficial listing of the course content for CSE 111 Programming with Functions. Please see I-Learn for the official list of materials, including quizzes and due dates.

Lesson	Domain Topics	Computing Topics
--------	---------------	------------------

Overview

Syllabus

Lesson 1 - Getting Started

Development environment

Prepare

input, data types, arithmetic, f-strings, print, if, logic

Prepare: Checkpoint

exercise heart rates

input, int, arithmetic, f-strings, print

Teach: Class Activity (for campus sections)

pendulum

input, float, arithmetic, f-strings, print

Prove: Milestone

tire volume

input, float, arithmetic, f-strings, print

Lesson 2 - Calling Functions

Prepare

calling built-in functions, functions in standard modules, and methods; named arguments.

Prepare: Checkpoint

items per box

math.ceil

Teach: Team Activity

purchase discount

selection, datetime.now, datetime.weekday

Prove: Assignment

tire volume

datetime.now, datetime format codes, open, print

Lesson 3 - Writing Functions

Prepare

user-defined function, header, parameter, body, return, main

Lesson	Domain Topics	Computing Topics
Prepare: Checkpoint	fuel efficiency	writing functions, main
Teach: Team Activity	fitness: body mass index, basal metabolic rate	writing functions, main
Prove: Milestone	2-D graphics	writing and calling functions

Lesson 4 - Function Details

Prepare		variable scope, default parameter values, optional arguments, function design heuristics
Prepare: Checkpoint	cone volume	variable scope: fix a program with a broken function that tries to use variables that are defined in another function
Teach: Team Activity	storage efficiency of steel cans	writing functions, local variables, calling functions
Prove: Assignment	2-D graphics	writing and calling functions

Lesson 5 - Testing Functions

Prepare		assert, pytest, pytest.approx, pytest.main, if __name__ == "__main__":
Prepare: Checkpoint	prefix and suffix	pytest, assert, test string functions
Teach: Team Activity	given name, surname, full name	pytest, assert, test string functions, fix errors in functions
Prove: Milestone	English parts of speech, generate a sentence	pytest, assert

Lesson 6 - Troubleshooting Functions

Prepare	syntax error, logic error, error messages, print statements, test functions, debugger
-------------------------	---

Lesson	Domain Topics	Computing Topics
Prepare: Checkpoint	fuel usage	debugger
Teach: Team Activity	self-esteem measure	debugger
Prove: Assignment	English parts of speech, generate a sentence	debugger

Lesson 7 - Lists and Repetition

Prepare		Lists, repetition, pass by value and reference
Prepare: Checkpoint		demonstration of pass by value and reference
Teach: Team Activity	pseudo random numbers	lists, append, write a function with default parameter values, call a function with optional arguments
Prove: Milestone	molar mass calculator, molecules, elements	create and return a compound list, retrieve and print individual elements from a compound list

Lesson 8 - Dictionaries

Prepare		Dictionaries, compound values, find one item, process all items, convert between list and dictionary
Prepare: Checkpoint	vehicles	in operator, use a key to retrieve a value
Teach: Team Activity	family history	retrieve a value from a dictionary, process all items in a dictionary, lists, indexes
Prove: Assignment	molar mass calculator, molecules, elements	create a dictionary, process all items in a dictionary, retrieve elements from a list

Lesson 9 - Text Files

Prepare	text files, open, for each line, CSV files, csv module
-------------------------	--

Lesson	Domain Topics	Computing Topics
Prepare: Checkpoint	Canadian Provinces	text files, for loop, lists, append, pop, count
Teach: Team Activity	student IDs	csv module, dictionaries
Prove: Milestone	grocery store	csv module, dictionaries

Lesson 10 - Handling Exceptions

Prepare		exception, try, except, else, finally, TypeError, ValueError, ZeroDivisionError, IndexError, KeyError, FileNotFoundError, PermissionError, validate user input
Prepare: Checkpoint	text file and line number	exception handling, FileNotFoundError, PermissionError, ValueError, IndexError
Teach: Team Activity	vehicle accidents	exception handling, FileNotFoundError, PermissionError, ValueError, csv.Error, KeyError, ZeroDivisionError
Prove: Assignment	grocery store	summation, round, datetime.now, datetime format codes, try, except, FileNotFoundError, PermissionError, KeyError

Lesson 11 - Functional Programming

Prepare		higher-order functions, nested functions, lambda functions, map, filter, sort key
Prepare: Checkpoint	U.S. phone numbers	higher-order functions, map function, read a text file into a list
Teach: Team Activity	student given name, surname, and birthdate	higher-order functions, nested functions, lambda functions, sorted function, read a CSV file into a compound list
Prove: Milestone (block) (semester) .		student chosen program

Lesson 12 - Using Objects

Prepare	creating an object, dot operator, attributes, methods, lists and dictionaries as objects
-------------------------	--

Lesson	Domain Topics	Computing Topics
<u>Prepare: Checkpoint</u>	fruit	modify a list using object oriented programming
<u>Teach: Team Activity</u>	GUI	creating objects, accessing attributes, calling methods
Prove: Milestone <u>(block)</u> <u>(semester)</u>	student chosen program	

Lesson 13 - Student Chosen Program

No prepare content, checkpoint, or team activity. Students work on their own chosen program.

Prove: Assignment
[\(block\)](#)
[\(semester\)](#).

Lesson 14 - Conclusion

[Prepare](#)

CSE 111 Syllabus

Overview

CSE 111 students become more organized, efficient, and powerful computer programmers by learning to research and call functions written by others; to write, call, debug, and test their own functions; and to handle errors within functions. CSE 111 students write programs with functions to solve problems in many disciplines, including business, physical science, human performance, and humanities.

Prerequisites

Before beginning CSE 111, you must successfully complete one of the following:

- CSE 110 - Programming Building Blocks (2)
- CS 101 - Introduction to Programming (2)
- CIT 160 - Introduction to Programming (3)
- A minimum score of 170 on the LUC test
- Pathway Connect

Learning Outcomes

Successful graduates of CSE 111 will do the following:

1. Write and call functions in programs to accomplish meaningful tasks in a variety of domains
2. Research and call functions written by others
3. Write programs that can detect and recover from invalid conditions
4. Use libraries and objects written by others
5. Follow good practices in designing, writing, and debugging functions

Topics

- functions, parameters, default parameter values, return, arguments, named arguments
- lambda functions
- lists, dictionaries
- text files, CSV files, csv Reader
- testing using pytest

- exception handling
- tkinter

Textbook

There is no textbook for this course. Instead, I-Learn contains links to videos and web pages with the preparation material students will need.

Technology

In this course, you will use Python 3 and Visual Studio Code (VS Code). These applications are free and available for Windows, MacOS, and Linux. Each student must have a laptop or desktop computer that can run these applications.

Students will use Microsoft Teams for communication about the course and I-Learn to submit assignments and quizzes.

Organization

This course is organized into a series of lessons. In the semester version of this course, students will complete one lesson each week. In the block version of this course, students will complete two lessons each week.

Each lesson is organized as follows:

- Prepare: Content—Articles and videos that each student should read and watch before beginning the other activities in the lesson.
- Prepare: Checkpoint—A small individual programming assignment designed to help each student practice the concepts taught in the prepare content.
- Teach: Team Activity—A one-hour programming activity that students will complete in groups of 3–5 students. Campus students will complete this activity during class. Online students will complete this activity during a synchronous video conference that they arrange.
- Prove: Programming Assignment—An individual programming project. Most of these span two lessons, with a milestone deliverable due at the end of the first lesson.
- Ponder: Check your understanding—Every other lesson includes a multiple-choice quiz to help students check their understanding of the topic.
- Ponder: Reflection—Every other lesson will contain a two question quiz to allow students to reflect on the things they are learning.

Learning Model

We encourage you to learn by study and also by faith [D&C 88:118](#)).

The three processes (prepare, teach one another, ponder and prove) of the BYU-Idaho [Learning Model](#) will help you deepen your learning experience. In this course, the Prepare phase of the Learning Model is delivered through the prepare content (articles and videos). The Teach One Another phase is facilitated through the team activities. The Ponder and Prove phase is measured through the weekly prove assignment.

The Five Principles (exercise faith; teach by the Spirit; lay hold on the word of God; take action; and love, serve, and teach) of the learning model is where you, the student, can take personal responsibility and invite the Spirit to be part of your study and learning process.

Grading

Each assignment in this course fits into one of five groups. Each group of assignments will contribute to your final grade according to the following percentages:

- 15%—Checkpoints
- 20%—Team Activities
- 50%—Prove Assignments
- 10%—Check Your Understanding
- 5%—Reflections

Prove Assignments will be graded in broad categories according to the following:

- 0%—Nothing submitted
- 50%—Some attempt made
- 75%—Developing (but significantly deficient)
- 85%—Slightly deficient
- 93%—Meets requirements
- 100%—Shows creativity and exceeds requirements. This will be explained in the individual assignments, but there is an expectation to show creativity and extend your assignments beyond the minimum standard that is specifically required.

Letter grades will be awarded as follows:

Percentage	Letter
Range	Grade
93%	100.00% A
90%	92.99% A-
87%	89.99% B+

Percentage Range	Letter Grade
83%	B
80%	B-
77%	C+
73%	C
70%	C-
67%	D+
63%	D
60%	D-
0%	F

Late Work

For all assignments, there is a 10% penalty for each day that has passed since the due date, up to a maximum of 50% penalty for any given assignment. This means that you can earn partial credit for assignments that you submit after the due date. Extenuating circumstances should be discussed with the instructor prior to the assignment due date.

Student Support

Support is available in many ways including via other class members and discussion in Microsoft Teams. In addition, help is available through the university's [academic support center](#).

BYU-Idaho Policies

Academic Honesty

You are expected to follow the university's [policies for academic honesty](#).

You may work with your classmates, but you must submit your own work for all assignments. Share ideas with your classmates; do not share code! Assistance from a classmate should be on par with the help students would expect from a lab assistant.

If you work closely with another student, helping teach and learn from each other, make sure you each write your own code, but in this case, your solutions may end up being very similar. This is fine, but please make sure to put a comment in your code stating that you

wrote your own program, but worked closely with that person, and that is why they are similar.

We encourage you to use the Internet as a resource, but you should not copy and paste someone else's work as your own. Cite all sources and follow copyright laws. ***When in doubt, give credit and be upfront.***

Do not look for or share solutions on "note sharing" internet sites.

The penalty for copying or plagiarism of assignments might be one or more of the following: a score of zero (0) on an assignment, a failing grade in the class, being asked to withdraw from the class, or disciplinary action by the University.

Dress and Grooming

Students are expected to follow the university's [Dress and Grooming Standards](#)

Academic Grievances

If you have a concern about your course, we encourage you to contact your instructor. If your concern cannot be resolved in this way, you may contact the [BYU-Idaho Support Center](#) to formally register a concern or grievance. You can read more in the [Student Grievance Policy](#).

Preventing Sexual Misconduct

BYU-Idaho prohibits sex discrimination by its employees and students in all of its education programs or activities. This includes all forms of sexual harassment, such as sexual assault, dating violence, domestic violence, stalking, conditioning a grade or job on participation in sexual conduct, and other forms of unwelcome sexual conduct.

As an instructor, one of my responsibilities is to help create a safe learning environment for my students and for the campus as a whole. University policy requires deans and department chairs, and encourages all faculty, to report every incident of sexual harassment that comes to their attention. If you encounter or experience sexual harassment, please contact the Title IX Coordinator at titleix@byui.edu or [208-496-9209](tel:208-496-9209). Additional information about sex discrimination, sexual harassment, and available resources can be found at www.byui.edu/titleix.

Disability Services

BYU-Idaho does not discriminate against persons with disabilities in providing its educational and administrative services and programs and follows applicable federal and

state law. This policy extends to the University's electronic and information technologies (EIT).

Students with qualifying disabilities should contact the Disability Services Office at disabilityservices@byui.edu or [208-496-9210](tel:208-496-9210). Additional information about Disability Services resources can be found at www.byui.edu/disabilities.

01 Prepare Your Development Environment

Each student in CSE 111 must install and use three pieces of free software which are:

- The Python Programming Language version 3.10
- Visual Studio Code (also known as VS Code)
- The Microsoft Python extension in VS Code

If you recently completed CSE 110, it is possible that you already have all three pieces of software installed on your computer. Watch these three videos from Microsoft that explain a bit about Python, Visual Studio Code, and the Microsoft Python extension in VS Code.

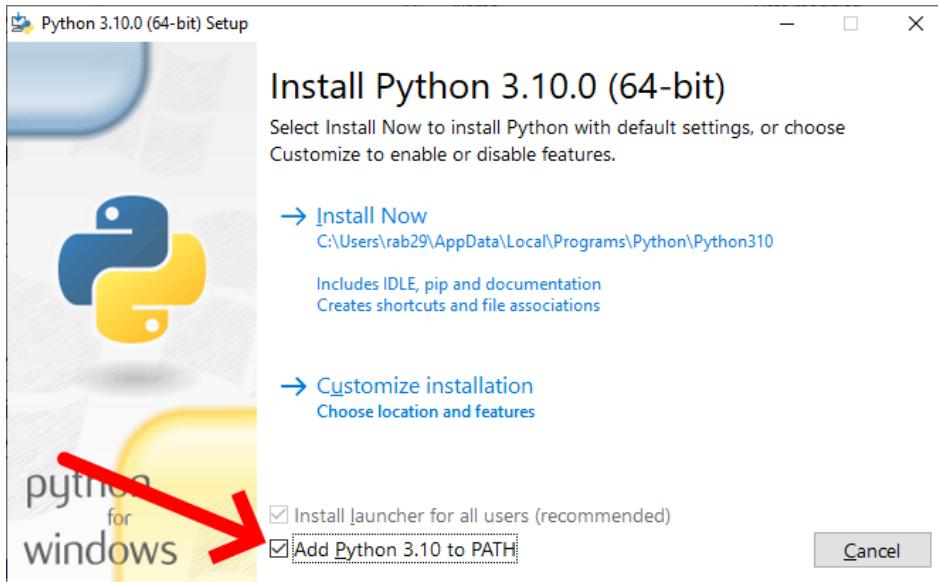
[Introducing Python](#) (3 minutes)

[Getting Started](#) (3 minutes)

[Configuring VS Code](#) (3 minutes)

If your computer doesn't have Python, VS Code, and the Python extension for VS Code already installed, follow these steps to install the software:

1. Download and install the latest stable version of Python 3.10. As of November 2021, the latest stable version of Python 3.10 is Python 3.10.0.
 - a. Open the [Python downloads page](#) in your browser.
 - b. Download and save the Python 3.10 installer for your operating system.
 - c. Double click the file that you downloaded to start installing Python.
- d. On the first screen of the installer, be certain to check both boxes: "Install launcher..." and "Add Python 3.10 to PATH" as shown in this screen shot:



- e. Click the "Install Now" link and Python will be installed on your computer.
2. Download and install VS Code.
 - a. Open the [Visual Studio Code download page](#) in your browser.
 - b. Download VS Code for your operating system.
 - c. Install VS Code.
 - d. Accept all the default settings.
3. Download a sample Python program.
 - a. Create a folder named cse111 on your computer where you will store all the programs that you write for CSE 111.
 - b. Download and save this [sample Python file](#) into the folder that you created.
4. Install the Microsoft Python extension in VS Code.
 - a. Run VS Code on your computer.
 - b. In VS Code, open the sample Python file that you downloaded.
 - c. In the lower right corner of VS Code, a small dialog will open that says, "The 'Python' extension is recommended for this file type."
 - d. Click the "Install" button in that dialog.
5. Execute the sample Python program.
 - a. Execute the sample Python file that you downloaded by clicking the run icon in the upper right corner of VS Code. The run icon is a green triangle (or).
 - b. The sample file should run without errors, and its output should be similar to this:

```
Hello! What is your name?   
3  
2  
1  
Welcome to CSE 111, Amelia!
```

Summary

To prepare your computer so that you can develop Python programs in CSE 111, you should have done the following:

1. Install the Python programming language version 3.10
2. Create a folder named `cse111` where you will save your Python programs
3. Install a text editor named Visual Studio Code
4. Install the Microsoft Python extension inside VS Code
5. Download and run a sample Python program to ensure your computer is ready

lesson01/sample.py

```
1 # Import the sleep function from the time module, so
2 # that the sleep function can be used in this program.
3 from time import sleep
4
5 # Prompt the user to enter her name.
6 name = input("Hello! What is your name? ")
7
8 # Print the numbers 3, 2, 1.
9 for i in range(3, 0, -1):
10     print(i, flush=True)
11     sleep(0.5) # Pause for 1/2 second
12
13 # Use a Python f-string to format a greeting
14 # for the user and then print the greeting.
15 print(f"Welcome to CSE 111, {name}!")
```

01 Prepare: Review Python

The concepts in CSE 111 build on the concepts that you learned in [CSE 110](#). In order to be successful in CSE 111, it is important that you remember and understand the concepts from CSE 110. To help you remember those concepts, during lesson 1 of CSE 111, you will review programming concepts that you learned in CSE 110.

Concepts

Here is a list of the Python programming concepts and topics from CSE 110 that you should review during this lesson.

Comments

A **comment** in a computer program is a note or description that is supposed to help a programmer understand the program. Computers ignore comments in a program. In Python, a comment begins with the hash symbol (#) and extends to the end of the current line.

User Input

In a Python program, we use the [input\(\) function](#) to get input from a user in a terminal window. The `input` function always returns a string of characters.

Variables

A **variable** is a location in a computer's memory where a program stores a value. A variable has a name, a data type, and a value. In Python, we assign a value to a variable by using the assignment operator, which is the equals symbol (=). A computer may change the value and data type of a variable while executing a program.

Data Types

Python has many **data types** including `str`, `bool`, `int`, `float`, `list`, and `dict`. Most of the data types that you will use in your programs in CSE 111 are shown in the following list.

- A **str** (string) is any text inside single or double quotes, any text that a user enters, and any text in a text file. For example:

```
greet = "Hello"  
text = "23"
```

```
| color = input("What is your favorite color? ") |
```

- A **bool** (Boolean variable) is a variable that stores either True or False. A Boolean variable may not store any other value besides True or False. For example:

```
found = True
```
- An **int** (integer) is a whole number like 14. An int may not have a fractional part or digits after the decimal point. For example:

```
x = 14
```
- A **float** (floating point number) is a number that may have a fractional part or digits after the decimal point like 7.51. For example:

```
sample = 7.51
```
- A **list** is a collection of values. Each value in a list is called an element and is stored at a unique index. The primary purpose of a list is to efficiently store many elements. In a Python program, we can create a list by using square brackets ([and]) and separating the elements with commas (,). For example here are two lists named colors and samples:

```
colors = ["yellow", "red", "green", "yellow", "blue"]
samples = [6.5, 7.2, 7.0, 8.1, 7.2, 6.8, 6.8]
```

You will study lists in lesson 7 of this course.

- A **dict** (dictionary) is a collection of items. Each item is a key value pair. The primary purpose of a dictionary is to enable a computer to find items very quickly. In a Python program, we can create a dictionary by using curly braces ({ and }) and separating the items with commas (,). For example:

```
students = {
    "42-039-4736": "Clint Huish",
    "61-315-0160": "Amelia Davis",
    "10-450-1203": "Ana Soares",
    "75-421-2310": "Abdul Ali",
    "07-103-5621": "Amelia Davis"
}
```

You will study dictionaries in lesson 8 of this course.

It is possible to convert between many of the data types. For example, to convert from any data type to a string, we use the [str\(\) function](#). To convert from a string to an integer, we use the [int\(\) function](#), and to convert from a string to a float, we use the [float\(\) function](#). The int() and float() functions are especially useful to convert user input, which is always a string, to a number. See the program in [example 2](#) below.

Displaying Results

In a Python program, we use the `print()` function to display results to a user. The easiest way to print both text and numbers together is to use a [formatted string literal](#) (also known as an f-string).

The Python program in example 1 creates ten different variables. Some of the variables are of type `str`, some are of type `bool`, some of type `int`, and some of type `float`. The program uses f-strings to print the name, data type, and value of each variable.

```
1 # Example 1
2
3 # Create variables of different data types and then
4 # print the variable names, data types, and values.
5
6 a = "Her name is " # string
7 b = "Isabella" # string
8 c = a + b # string plus string makes string
9 print(f"a: {type(a)} {a}")
10 print(f"b: {type(b)} {b}")
11 print(f"c: {type(c)} {c}")
12 print()
13
14 d = False # boolean
15 e = True # boolean
16 print(f"d: {type(d)} {d}")
17 print(f"e: {type(e)} {e}")
18 print()
19
20 f = 15 # int
21 g = 7.62 # float
22 h = f + g # int plus float makes float
23 print(f"f: {type(f)} {f}")
24 print(f"g: {type(g)} {g}")
25 print(f"h: {type(h)} {h}")
26 print()
27
28 i = "True" # string because of the surrounding quotes
29 j = "2.718" # string because of the surrounding quotes
30 print(f"i: {type(i)} {i}")
31 print(f"j: {type(j)} {j}")
```

```
> python example_1.py
a: <class 'str'> Her name is
b: <class 'str'> Isabella
c: <class 'str'> Her name is Isabella

d: <class 'bool'> False
e: <class 'bool'> True

f: <class 'int'> 15
g: <class 'float'> 7.62
h: <class 'float'> 22.62

i: <class 'str'> True
j: <class 'str'> 2.718
```

The Python program in example 2 creates six different variables, some of type `string`, some of type `int`, and some of type `float`. Lines [4–5](#) and [7–8](#) of example 2 demonstrate

that no matter what the user types, the `input()` function always returns a string. Lines 13 and 14 show how to use the `int()` and `float()` functions to convert a string to a number so that the numbers can be used in calculations.

```
1 # Example 2
2
3 # The input function always returns a string.
4 k = input("Please enter a number: ")          # string
5 m = input("Please enter another number: ")    # string
6 n = k + m          # string plus string makes string
7 print(f"k: {type(k)} {k}")
8 print(f"m: {type(m)} {m}")
9 print(f"n: {type(n)} {n}")
10 print()
11
12 # The int and float functions convert a string to a number.
13 p = int(input("Please enter a number: "))      # int
14 q = float(input("Please enter another number: ")) # float
15 r = p + q          # int plus float makes float
16 print(f"p: {type(p)} {p}")
17 print(f"q: {type(q)} {q}")
18 print(f"r: {type(r)} {r}")
```

```
> python example_2.py
Please enter a number: 6
Please enter another number: 4
k: <class 'str'> 6
m: <class 'str'> 4
n: <class 'str'> 64

Please enter a number: 5
Please enter another number: 3
p: <class 'int'> 5
q: <class 'float'> 3.0
r: <class 'float'> 8.0
```

Arithmetic

Python has many **arithmetic operators** including power (**), negation (-), multiplication (*), division (/), floor division (//), modulo (%), addition (+), and subtraction (-).

Operator Precedence

When we write an arithmetic expression that contains more than one operator, the computer executes the operators according to their **precedence**, also known as the **order of operations**. This table shows the precedence for the arithmetic operators.

Operators	Description	Precedence
()	parentheses	highest
**	exponentiation (power)	↑
-	negation	

Operators	Description	Precedence
* / // %	multiplication, division, floor division, modulo	
+ -	addition, subtraction	↓
=	assignment	lowest

When an arithmetic expression includes two operators with the same precedence, the computer evaluates the operators from left to right. For example, in the arithmetic expression `x / y * c` the computer will first divide `x` by `y` and then multiply that result by `c`. If you need the computer to evaluate a lower precedence operator before a higher precedence one, you can add parentheses to the expression to change the evaluation order. The computer will always evaluate arithmetic that is inside parentheses first because parentheses have the highest precedence of all the arithmetic operators.

If this is the first time that you have encountered arithmetic operator precedence, you should watch this Khan Academy video: [Introduction to Order of Operations](#) (10 minutes).

The Python program in example 3 gets input from the user and converts the user input into two numbers on lines 9 and 10. Then at line 13 the program computes the length of a cable from the two numbers. Finally at line 17, the program uses an f-string to print the length rounded to two places after the decimal point.

```

1 # Example 3
2
3 # Given the distance that a cable will span and the distance
4 # it will sag or dip in the middle, this program computes the
5 # length of the cable.
6
7 # Get user input and convert it from
8 # strings to floating point numbers.
9 span = float(input("Distance the cable must span in meters: "))
10 dip = float(input("Distance the cable will sag in meters: "))
11
12 # Use the numbers to compute the cable length.
13 length = span + (8 * dip**2) / (3 * span)
14
15 # Print the cable length in the
16 # console window for the user to see.
17 print(f"Length of cable in meters: {length:.2f}")

```

```

> python example_3.py
Distance the cable must span in meters: 500
Distance the cable will sag or dip in meters: 18.5
Length of cable in meters: 501.83

```

In example 3, the arithmetic that is written on line 13 comes from a well known formula. Given the distance that a cable must span and the vertical distance that the cable will be allowed to sag or dip in the middle of the cable, the formula for calculating the length of the cable is:

$$length = span + \frac{8 dip^2}{3 span}$$

Shorthand Operators

The Python programming language includes many **augmented assignment operators**, also known as **shorthand operators**. All the shorthand operators have the same precedence as the assignment operator (`=`). Here is a list of some of the Python shorthand operators:

```
**= *= /= /= %= += -=
```

To understand what the shorthand operators do and why Python includes them, imagine a program that computes the price of a pizza. The price of a large pizza with cheese and no other toppings is \$10.95. The price of each topping, such as ham, pepperoni, olives, and pineapple is \$1.45. Here is a short example program that asks the user for the number of toppings and computes the price of a pizza:

```
1 # Example 4
2
3 # Compute the total price of a pizza.
4
5 # The base price of a large pizza is $10.95
6 price = 10.95
7
8 # Ask the user for the number of toppings.
9 number_of_toppings = int(input("How many toppings? "))
10
11 # Compute the cost of the toppings.
12 price_per_topping = 1.45
13 toppings_cost = number_of_toppings * price_per_topping
14
15 # Add the cost of the toppings to the price of the pizza.
16 price = price + toppings_cost
17
18 # Print the price for the user to see.
19 print(f"Price: ${price:.2f}")
```

```
> python example_4.py
How many toppings? 3
Price: $15.30
```

The statement at [line 16](#) in example 4 causes the computer to get the value in the `price` variable which is 10.95, then add the cost of the toppings to 10.95, and then store the sum back into the `price` variable. Python includes a shorthand operator that combines addition (+) and assignment (=) into one operator (`+=`). We can use this shorthand operator to rewrite line 16 like this:

```
price += toppings_cost
```

This statement with the shorthand operator is equivalent to the statement on [line 16](#) of example 4, meaning the two statements cause the computer to do the same thing.

Example 5 contains the same program as example 4 but uses the shorthand operator `+=` at [line 16](#).

```
1 # Example 5
2
3 # Compute the total price of a pizza.
4
5 # The base price of a large pizza is $10.95
6 price = 10.95
7
8 # Ask the user for the number of toppings.
9 number_of_toppings = int(input("How many toppings? "))
10
11 # Compute the cost of the toppings.
12 price_per_topping = 1.45
13 toppings_cost = number_of_toppings * price_per_topping
14
15 # Add the cost of the toppings to the price of the pizza.
16 price += toppings_cost
17
18 # Print the price for the user to see.
19 print(f"Price: ${price:.2f}")
```

```
> python example_5.py
How many toppings? 3
Price: $15.30
```

if Statements

In Python, we use `if` statements to cause the computer to make decisions; `if` statements are also called **selection** statements because the computer selects one group of statements to execute and skips the other group of statements.

There are six comparison operators that we can use in an `if` statement:

<code><</code>	less than
<code><=</code>	less than or equal
<code>></code>	greater than
<code>>=</code>	greater than or equal
<code>==</code>	equal to
<code>!=</code>	not equal to

Example 6 contains Python code that checks if a number is greater than 500.

```
1 # Example 6
2
3 # Get an account balance as a number from the user.
4 balance = float(input("Enter the account balance: "))
5
```

```
6 # If the balance is greater than 500, then
7 # compute and add interest to the balance.
8 if balance > 500:
9     interest = balance * 0.03
10    balance += interest
11
12 # Print the balance.
13 print(f"balance: {balance:.2f}")
```

```
> python example_6.py
Enter the account balance: 350
balance: 350.0

> python example_6.py
Enter the account balance: 525
balance: 540.75
```

If you have written programs in other programming languages such as JavaScript, Java, or C++, you always used curly braces to mark the start and end of the body of an `if` statement. However, notice in example 6 that `if` statements in Python do not use curly braces. Instead, we type a colon (`:`) after the comparison of the `if` statement as shown on line 8. Then we indent all the statements that are in the body of the `if` statement as shown on lines 9 and 10. The body of the `if` statement ends with the first line of code that is not indented, like line 13.

It may seem strange to not use curly braces to mark the start and end of the body of an `if` statement. However, the Python way forces us to write code where the indentation matches the functionality or in other words, the way we indent the code matches the way that the computer will execute the code.

if ... elif ... else Statements

Each `if` statement may have an `else` statement as shown in example 7 on line 13. We can combine `else` and `if` into the keyword `elif` as shown on lines 9 and 11.

```
1 # Example 7
2
3 # Get the cost of an item from the user.
4 cost = float(input("Please enter the cost: "))
5
6 # Determine a discount rate based on the cost.
7 if cost < 100:
8     rate = 0.10
9 elif cost < 250:
10    rate = 0.15
11 elif cost < 400:
12    rate = 0.18
13 else:
14     rate = 0.20
15
16 # Compute the discount amount
17 # and the discounted cost.
18 discount = cost * rate
```

```
19 cost -= discount
20
21 # Print the discounted cost for the user to see.
22 print(f"After the discount, you will pay {cost:.2f}")
```

```
> python example_7.py
Please enter the cost: 300
After the discount, you will pay 246.0
```

Logical Operators

Python includes two **logic operators** which are the keywords `and`, `or` that we can use to combine two comparisons. Python also includes the logical `not` operator. Notice in Python that the logical operators are literally the words: `and`, `or`, `not` and `not` symbols as in other programming languages:

```
if driver >= 54 or (driver >= 32 and passenger >= 54):
    message = "Enjoy the ride!"
```

Videos

If any of the concepts or topics in the previous list seem unfamiliar to you, you should review them. To review the unfamiliar concepts, you could rewatch some of the Microsoft videos about Python that you watched for CSE 110:

[Input and print Functions](#) (4 minutes)

[Demonstration of print Function](#) (6 minutes)

[Comments](#) (3 minutes)

[String Data Type](#) (5 minutes)

[Numeric Data Types](#) (6 minutes)

[Conditional Logic](#) (5 minutes)

[Handling Multiple Conditions](#) (6 minutes)

[Complex Conditions](#) (4 minutes)

Tutorials

Reading these tutorials may help you recall programming concepts from CSE 110.

[Why Choose Python?](#)

[Interacting with Python](#)

[Basic Data Types in Python](#)

[Variables in Python](#)

[Operators and Expressions in Python](#)

[Conditional Statements in Python](#)

You could also read some of the Python tutorials at [W3 Schools](#). Or you could search for "Python" in the [BYU-Idaho library online catalog](#) and read one of the online books from the result set.

Summary

During this lesson, you are reviewing the programming concepts that you learned in CSE 110. These concepts include how to do the following:

- write a comment
- use the `input` and `print` functions
- use the `int` and `float` functions to convert a value from a string to a number
- store a value in a variable
- perform arithmetic
- write `if ... elif ... else` statements
- use the logical operators `and`, `or`, `not`

01 Checkpoint: Review Python

Purpose

Write a Python program that gets input from a user, uses variables, performs arithmetic, and displays results for the user to see.

Problem Statement

When you physically exercise to strengthen your heart, you should maintain your heart rate within a range for at least 20 minutes. To find that range, subtract your age from 220. This difference is your maximum heart rate per minute. Your heart simply will not beat faster than this maximum ($220 - \text{age}$). When exercising to strengthen your heart, you should keep your heart rate between 65% and 85% of your heart's maximum rate.

Assignment

Write a Python program named `heart_rate.py` that asks for a person's age and computes and outputs the slowest and fastest rates necessary to strengthen his heart. To start your program, copy and paste the following code into your program and use it as an outline as you write code. Note that in a Python program, a triple quoted string at the top of the file acts as a comment for the entire program.

```
"""
When you physically exercise to strengthen your heart, you
should maintain your heart rate within a range for at least 20
minutes. To find that range, subtract your age from 220. This
difference is your maximum heart rate per minute. Your heart
simply will not beat faster than this maximum (220 - age).
When exercising to strengthen your heart, you should keep your
heart rate between 65% and 85% of your heart's maximum rate.
"""
```

Helpful Documentation

The [prepare content](#) for this lesson explains how to write code to do the following:

- [Get input](#) from a user
- [Convert user input](#) from a string to a number
- [Calculate results](#)

- [Display results](#) for the user to see

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the input shown below. Ensure that your program's output matches the output below.

```
> python heart_rate.py
Please enter your age: 23
When you exercise to strengthen your heart, you should
keep your heart rate between 128 and 167 beats per minute.
```

2. Run your program using your age or the age of one of your parents. Use a calculator to ensure that the output is correct.

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Ponder

During this assignment, you wrote a Python program that gets input from a user, uses variables, performs arithmetic, and displays results for the user to see. Because you should have learned how to write this type of program in CSE 110, this assignment should have been fairly easy for you. If this assignment was difficult for you, you should review the concepts from [CSE 110](#) and the programs that you wrote in that course.

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson01/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 When you physically exercise to strengthen your heart, you
4 should maintain your heart rate within a range for at least 20
5 minutes. To find that range, subtract your age from 220. This
6 difference is your maximum heart rate per minute. Your heart
7 simply will not beat faster than this maximum (220 - age).
8 When exercising to strengthen your heart, you should keep your
9 heart rate between 65% and 85% of your heart's maximum rate.
10 """
11
12 # Get the user's age as a string.
13 text = input("Please enter your age: ")
14
15 # Convert the string that the user entered into an integer.
16 age = int(text)
17
18 # Compute the slowest and fastest beneficial
19 # heart rates from the user's age.
20 max_rate = 220 - age
21 slowest = max_rate * 0.65
22 fastest = max_rate * 0.85
23
24 # Use an f-string to create and print a message for the user to see.
25 print("When you exercise to strengthen your heart, you should")
26 print(f"keep your heart rate between {slowest:.0f} and {fastest:.0f}")
27 print("beats per minute.")
```

01 Class Activity: Review Python

Purpose

Write a Python program that gets input from a user, uses variables, performs arithmetic using the `math` module, and displays results for the user to see.

Problem Statement

The time in seconds that a pendulum takes to swing back and forth once is given by this formula:

$$t = 2\pi \sqrt{\frac{h}{9.81}}$$

- t is the time in seconds,
- π is the constant PI, which is the ratio of the circumference of a circle divided by its diameter (use `math.pi`),
- h is the length of the pendulum in meters.

Activity

Write a program named `pendulum.py` that prompts a user to enter the length of a pendulum in meters and then computes and prints the time in seconds that it takes for that pendulum to swing back and forth. To start your program, copy and paste the following code into your program and use it as an outline as you write code. Note that in a Python program, a triple quoted string at the top of the file acts as a comment for the entire program.

```
"""
The time in seconds that a pendulum takes to swing back and forth once is given by this formula:

      / h
t = 2π / -----
      \ 9.81

t is the time in seconds,
π is the constant PI, which is the ratio of the circumference
      of a circle divided by its diameter (use math.pi),
h is the length of the pendulum in meters.

Write a program that prompts a user to enter the length of a
```

```
pendulum in meters and then computes and prints the time in  
seconds that it takes for that pendulum to swing back and forth.  
"""
```

Helpful Documentation

The [prepare content](#) for this lesson explains how to write code to do the following:

- [Get input](#) from a user
- [Convert user input](#) from a string to a number
- [Calculate results](#)
- [Display results](#) for the user to see

The Python [math module](#) contains mathematical constants and functions including [math.pi](#) and [math.sqrt\(\)](#).

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the input shown below. Ensure that your program's output matches the output below.

```
> python pendulum.py  
Length of pendulum (meters): 1.5  
Time (seconds): 2.46
```

Sample Solution

Please work diligently with your class to complete this activity. After class is over, please compare your approach to the [sample solution](#). Please **do not look at the sample solution** until you have either finished the program or diligently worked with your class.

Ponder

During this assignment, you wrote a Python program that gets input from a user, uses variables, performs arithmetic, and displays results for the user to see. Because you should have learned how to write this type of program in CSE 110, this assignment should have been fairly easy for you. If this assignment was difficult for

you, you should review the concepts from [CSE 110](#) and the programs that you wrote in that course.

lesson01/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3     The time in seconds that a pendulum takes to swing back and
4     forth once is given by this formula:
5
6         / ----
7         t = 2π / ----
8             √ 9.81
9
10    t is the time in seconds,
11    π is the constant PI, which is the ratio of the circumference
12        of a circle divided by its diameter (use math.pi),
13    h is the length of the pendulum in meters.
14
15    Write a program that prompts a user to enter the length of a
16    pendulum in meters and then computes and prints the time in
17    seconds that it takes for that pendulum to swing back and forth.
18 """
19
20
21 # Import the math module so that
22 # we can use math.pi and math.sqrt()
23 import math
24
25 # Get the length from the user and
26 # convert it to a floating point number.
27 length = float(input("Length of pendulum (meters): "))
28
29 # Compute the time in seconds required for
30 # the pendulum to swing back and forth.
31 time = 2 * math.pi * math.sqrt(length / 9.81)
32
33 # Display the time rounded to two digits
34 # after the decimal for the user to see.
35 print(f"Time (seconds): {time:.2f}")
```

01 Prove Milestone: Review Python

Purpose

Prove that you can write a Python program that gets input from a user, performs arithmetic, and displays results for the user to see.

Problem Statement

The size of a car tire in the United States is represented with three numbers like this: 205/60R15. The first number is the width of the tire in millimeters. The second number is the aspect ratio. The third number is the diameter in inches of the wheel that the tire fits. The volume of space inside a tire can be approximated with this formula:

$$v = \frac{\pi w^2 a (w a + 2,540 d)}{10,000,000,000}$$

- v is the volume in liters,
- π is the constant PI, which is the ratio of the circumference of a circle divided by its diameter (use `math.pi`),
- w is the width of the tire in millimeters,
- a is the aspect ratio of the tire, and
- d is the diameter of the wheel in inches.

If you're curious about how the above formula was derived, you can read the [tire volume formula derivation](#).

Assignment

Write a Python program named `tire_volume.py` that reads from the keyboard the three numbers for a tire and computes and outputs the volume of space inside that tire.

* For all assignments in CSE 111, please write your program in a file named as the assignment states. This assignment requires you to name your program `tire_volume.py`. If you name your program something else, it will be harder for the graders to score your submitted assignment.

Helpful Documentation

The [prepare content](#) for this lesson explains how to write code to do the following:

[Get input from a user](#)

[Convert user input](#) from a string to a number

[Calculate results](#)

[Display results](#) for the user to see

The Python `math` module contains mathematical constants and functions, including `math.pi`, and is described in the [Math Module Reference](#).

The short video titled [How to Write a Python Program that Computes the Surface Area of a Cone](#) (10 minutes), shows a BYU-Idaho faculty member solving a problem that is similar to this prove assignment.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python tire_volume.py
Enter the width of the tire in mm (ex 205): 185
Enter the aspect ratio of the tire (ex 60): 50
Enter the diameter of the wheel in inches (ex 15): 14

The approximate volume is 24.09 liters

> python tire_volume.py
Enter the width of the tire in mm (ex 205): 205
Enter the aspect ratio of the tire (ex 60): 60
Enter the diameter of the wheel in inches (ex 15): 15

The approximate volume is 39.92 liters
```

Ponder

During this assignment, you wrote a Python program that gets input from a user, uses variables, performs arithmetic, and displays results for the user to see. Because you should have learned how to write this type of program in CSE 110, this assignment should have been fairly easy for you. If this assignment was difficult for you, you should review the concepts from [CSE 110](#) and the programs that you wrote in that course.

Submission

On or before the due date, return to I-Learn and report your progress on this milestone.

O2 Prepare: Calling Functions

Because most useful computer programs are very large, programmers divide their programs into parts. Dividing a program into parts makes it easier to write, debug, and understand. A programmer can divide a Python program into modules, classes, and functions. In this lesson, you will learn how to call existing functions, and in the next lesson, you will learn how to write your own functions.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is a Function?

A **function** is a group of statements (computer commands) that together perform one task. Broadly speaking, there are four types of functions in Python which are:

1. Built-in functions
2. Standard library functions
3. Third-party functions
4. User-defined functions

A programmer (you) can save lots of time by using existing functions. In this lesson, you will learn how to use (call) the first two types of functions. In [lesson 5](#), you will learn how to install third-party modules and call third-party functions. In the [next lesson](#), you will learn how to write and call user-defined functions.

Built-in Functions

Python includes many **built-in functions** such as: `input`, `int`, `float`, `str`, `len`, `range`, `abs`, `round`, `list`, `dict`, `open`, and `print`. These are called built-in functions because you don't have to import any module to use them. They are simply a built-in part of the Python language. You can read about the built-in functions in the [Built-in Functions](#) section of the official Python online reference.

How to Call a Function

A programmer uses a function by calling it (also known as invoking it). To **call** (or **invoke**) a function means to write code that causes the computer to execute the code that is inside that function. Regardless of the type of function (built-in, standard, third-party, or

user-defined), a function is called by writing its name followed by a set of parentheses (). During CSE 110 and 111, you often wrote code that called the built-in `input` and `print` functions like this:

```
name = input("Please enter your name: ")  
print(f"Hello {name}")
```

```
> python example_1.py  
Please enter your name: Miyuki  
Hello Miyuki
```

To call a function you must know the following three things:

1. The name of the function
2. The parameters that the function takes
3. What the function does

These three pieces of information are normally available in online documentation. For example, from the online Python reference for the `input` function, we read this:

```
input(prompt)  
Write the prompt parameter to the terminal window, then read a line of user input from the terminal window, convert the input to a string, and return the input as a string.
```

From this short description, we know the following:

1. The name of the function is `input`.
2. The function takes one parameter named *prompt*.
3. The function writes the prompt to a terminal window and then reads user input from the terminal and returns that input to the calling function.

A **parameter** is a piece of data that a function needs in order to complete its task. In the online reference for the `input` function, we see that the `input` function has one parameter named *prompt*.

An **argument** is the value that is passed through a parameter into a function. In other words, parameters are listed in a function's documentation, and arguments are listed in a call to a function.

To write code that calls a function, we normally do the following:

1. Type a new variable name and use the assignment operator (=) to assign a value to the variable.
2. Type the name of the function followed by a set of parentheses.
3. Between the parentheses, type arguments that the computer will pass into the function through its parameters.

For example, the following code calls the built-in `input` function and passes the string "Please enter your name: " as the argument for the `prompt` parameter.

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

When a function has more than one parameter and a programmer writes code to call that function, the programmer nearly always writes the arguments in the same order as the parameters. Consider the description of the built-in `round` function:

`round(number, ndigits)`

Return `number` rounded to `ndigits` precision after the decimal point. If `ndigits` is omitted or is `None`, `round` returns the nearest integer to `number`.

Now consider this Python code that gets a number from a user, rounds that number to two digits after the decimal, and then prints the rounded number.

```
1 n = float(input("Please enter a number: "))
2 r = round(n, 2)
3 print(r)
```

```
> python example_2.py
Please enter a number: 95.716
95.72
```

In the previous example,

- The code on line 1 causes the computer to call the built-in `input` function and then call the built-in `float` function.
- Line 2 causes the computer to call the built-in `round` function and pass two arguments. Notice that the order of the arguments matches the order of the parameters. Specifically, the number to be rounded (`n`) is the first argument, and the number of digits after the decimal point (2) is the second argument.
- Line 3 causes the computer to call the built-in `print` function to print the rounded number.

Optional Arguments

When calling a function or method, some arguments are **optional**. Again consider the description of the built-in `round` function:

`round(number, ndigits)`

Return `number` rounded to `ndigits` precision after the decimal point. If `ndigits` is omitted or is `None`, `round` returns the nearest integer to `number`.

From the description, we read that the second argument is optional. If the programmer doesn't type a second argument, the value in the `number` parameter will be rounded to an integer. The next code example is similar to the previous example. The only difference

is that at [line 2](#) of the next example the programmer typed only one argument to the `round` function. Because the programmer omitted the second argument, the `round` function will round the number in its first parameter to an integer, which is shown in the output below.

```
1 n = float(input("Please enter a number: "))
2 r = round(n)
3 print(r)
```

```
> python example_3.py
Please enter a number: 95.716
96
```

Named Arguments

For some optional arguments, we must pass a **named argument**, which is an argument that is preceded by the name of its matching parameter. For example, here is an excerpt from the documentation for the `print` function:

```
print(*objects, sep=" ", end="\n", file=sys.stdout, flush=False)
    Print objects to the text stream file, separated by sep and followed by end.
    sep, end, file and flush, if present, must be given as named arguments.
```

Notice from the excerpt that the `print` function can take many objects that will be printed. Optionally, it can take parameters named `sep`, `end`, `file`, and `flush` that must be named when they are used. For example, this code calls the `print` function to print three words all separated by a vertical bar (|). Notice the named arguments `sep` and `flush`.

```
x = "sun"
y = "moon"
z = "stars"
print(x, y, z, sep="|", flush=True)
```

```
> python example_4.py
sun|moon|stars
```

How to Call a Function that Is inside a Module

A Python **module** is a collection of related functions. The Python **standard library** includes many modules which have more functions, such as the `math` module—which includes the `floor`, `ceil`, and `sqrt` functions and the `random` module—which includes the `randint`, `choice`, and `shuffle` functions. Consider the description of the `sqrt` function that is in the standard `math` module:

```
math.sqrt(x)
    Return the square root of x.
```

From this short description, we know the following:

1. The name of the containing module is `math`.
2. The name of the function is `sqrt`.
3. The function takes one parameter named `x`.
4. The function computes and returns the square root of the number that is in `x`.

To use any code that is in a module, you must import the module into your program and precede the function name with the module name. For example, if you wish to call the `math.sqrt` function, you must first import the `math` module and then type `math.` in front of `sqrt` like this:

```
import math
r = math.sqrt(71)
print(r)
```

```
> python example_5.py
8.426149773176359
```

In the above example, 71 is the argument that will be passed through the parameter `x` into the `math.sqrt` function. The `math.sqrt` function will compute the square root of 71 and return the computed value that will then be stored in the variable `r`. You can read more about the standard modules in the official documentation for the [Python Standard Library](#).

How to Call a Method

Python is an object-oriented language and includes many classes and objects. A **method** is a function that belongs to a class or object. Even though classes and objects are not part of this course (CSE 111), calling a method in Python is so common and so easy that you should know how to do it. A method is a kind of function, so calling a method is similar to calling a function. The difference is that to call a method we must type the name of the object and a period (.) in front of the method name.

Consider the program in example 6 that gets a string of text from a user and prints the number of characters in the string and prints the string in all upper case characters.

```
1 # Example 6
2
3 # Get a string of text from the user.
4 text1 = input("Enter a motivational quote: ")
5
6 # Call the built-in len function to get
7 # the number of characters in the text.
8 length = len(text1)
9
10 # Call the string upper method to convert
11 # the quote to upper case characters.
```

```
12 text2 = text1.upper()  
13  
14 # Call the built-in print function to print  
15 # the length of the text and the text in all  
16 # upper case for the user to see.  
17 print(length, text2)
```

```
> python example_6.py  
Enter a motivational quote: Rise, take up thy bed, and walk.  
32 RISE, TAKE UP THY BED, AND WALK.
```

Notice the code on [line 8](#) calls the built-in `len` function and the code on [line 12](#) calls the string `upper` method. Compare the function call in [line 8](#) to the method call in [line 12](#). To call the `len` function, we type the name of the function followed by a list of arguments inside parentheses. To call the `upper` method, we type the name of the object (`text1`) and a period, then the method name (`upper`), and then a list of arguments inside parentheses.

A method can receive arguments just like a function can. However, in example 6 at [line 12](#), there are no arguments passed to the `upper` method, so the parentheses are empty. In order for the computer to call the `upper` method, a programmer must type the empty parentheses. In other words, if you write a line of code to call the `upper` method but don't type the empty parentheses, like this:

```
text2 = text1.upper # Does NOT call the upper method
```

the computer will not call the `upper` method. Instead the computer will assign a reference to the `upper` method to the `text2` variable. You don't want the computer to do this because assigning a function reference won't make sense to you until you study functional programming.

How to Store a Returned Value

All the previous examples in this prepare content use the assignment operator (=) to store the value returned from a function in a variable. For example:

```
text = input("Enter a motivational quote: ")
```

While it's usually a good practice, you don't *have* to store the value that is returned from a function in a variable. Sometimes you will see it used directly as shown in example 7 at [lines 10, 14, and 16](#).

```
1 # Example 7  
2  
3 import math  
4  
5 # Get a number from the user.  
6 number = float(input("Enter a number: "))  
7  
8 # Call the math.sqrt function and
```

```

9 # immediately print its return value.
10 print( math.sqrt(number) )
11
12 # Call the math.sqrt function again and
13 # use its return value in an if statement.
14 if math.sqrt(number) < 100:
15     print(f"The square root is less than 100.")
16 elif math.sqrt(number) > 100:
17     print(f"The square root is more than 100.")
18 else:
19     print(f"The square root is exactly 100.")

```

```

> python example_7.py
Enter a number: 675
25.98076211353316
The square root is less than 100.

```

Notice in example 7, there are three statements that call the `math.sqrt` function, one at line 10 to print the square root, another at line 14 to check if the square root is less than 100, and yet another at line 16 to check if the square root is greater than 100. Every time the computer calls a function, the computer will execute the code that is inside that function. In example 7, because the arguments are the same at lines 10, 14 and 16, the returned result will be the same in all three cases. So it would be faster to save the result in a variable and reuse the variable instead, as shown in example 8 at lines 10, 12, 14, and 16.

```

1 # Example 8
2
3 import math
4
5 # Get a number from the user.
6 number = float(input("Enter a number: "))
7
8 # Call the math.sqrt function and store its
9 # return value in a variable to use later.
10 root = math.sqrt(number)
11
12 print(f"The square root is {root:.2f}")
13
14 if root < 100:
15     print(f"The square root is less than 100.")
16 elif root > 100:
17     print(f"The square root is more than 100.")
18 else:
19     print(f"The square root is exactly 100.")

```

```

> python example_8.py
Enter a number: 675
The square root is 25.98
The square root is less than 100.

```

Tutorial

If you are uncertain about any of the concepts in the previous section, you could reread the section. Also, you could read about the same concepts in the [Python functions tutorial](#) at w3schools.

Summary

A function is a group of statements that together perform one task. The computer will not execute the code in a function unless you write code that calls the function. In this lesson, you learned how to call built-in functions, functions that are in a module, and functions (methods) that belong to an object.

1. To call a built-in function, write code that follows this template:

```
variable_name = function_name(arg1, arg2, ... argN)
```

2. To call a function from a module, import the module and write code that follows this template:

```
import module_name  
  
variable_name = module_name.function_name(arg1, arg2, ... argN)
```

3. To call a method, write code that follows this template:

```
variable_name = object_name.method_name(arg1, arg2, ... argN)
```

02 Checkpoint: Calling Functions

Purpose

Check your understanding of calling built-in Python functions and functions that are in a standard Python module.

Problem Statement

In our modern world economy, many items are manufactured in large factories, then packed in boxes and shipped to distribution centers and retail stores. A common question for employees who pack items is "How many boxes do we need?"

Assignment

A manufacturing company needs a program that will help its employees pack manufactured items into boxes for shipping. Write a Python program named `boxes.py` that asks the user for two integers:

1. the number of manufactured items
2. the number of items that the user will pack per box

Your program must compute and print the number of boxes necessary to hold the items. This must be a whole number. Note that the last box may be packed with fewer items than the other boxes.

Helpful Documentation

The [prepare content](#) for this lesson explains how to call a function and a method.

The [math.ceil\(\) function](#) rounds a number up to the nearest integer that is greater than or equal to a number.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python boxes.py
Enter the number of items: 
Enter the number of items per box: 
For 8 items, packing 5 items in each box, you will need 2 boxes.

> python boxes.py
Enter the number of items: 
Enter the number of items per box: 
For 25 items, packing 4 items in each box, you will need 7 boxes.
```

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Ponder

During this assignment, you wrote code that calls the `math.ceil()` function. What did the `math.ceil()` function do in your program? If the `math.ceil()` function didn't exist, would this assignment have been more difficult to complete?

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson02/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 A manufacturing company needs a program that will help its employees
4 pack manufactured items into boxes for shipping. Write a Python
5 program named boxes.py that asks the user for two integers: 1) the
6 number of manufactured items and 2) the number of items that the user
7 will pack per box. Your program must compute and print the number of
8 boxes necessary to hold the items. This must be a whole number. Note
9 that the last box may be packed with fewer items than the other boxes.
10 """
11
12 # Import the math module so that we can call the math.ceil function.
13 import math
14
15 # Get two numbers from the user.
16 num_items = int(input("Enter the number of items: "))
17 items_per_box = int(input("Enter the number of items per box: "))
18
19 # Compute the number of boxes by dividing
20 # and then calling the math.ceil function.
21 num_boxes = math.ceil(num_items / items_per_box)
22
23 # Display a blank line.
24 print()
25
26 # Display the results for the user to see.
27 print(f"For {num_items} items, packing {items_per_box}"
28       f" items in each box, you will need {num_boxes} boxes.")
```

02 Team Activity: Calling Functions

Instructions

Each lesson in CSE 111 contains a team activity that is designed to take about one hour to complete. You should prepare for all team activities by completing the preparation material and the individual checkpoint assignment before starting a team activity. The goal of the team activities is for students to work together and teach and learn from each other. As your team completes a team activity, instead of moving through it as quickly as you can, you should help everyone understand the concepts.

Face-to-Face Students

Face-to-face students will complete the team activities in their classroom during class time.

Online Students in a Semester Section

Online students in a semester section will arrange and participate in a one hour synchronous video meeting with your team for each team activity. As your team works through the assignment, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Online Students in a Block Section

Students in a block section complete two lessons each week. This means that online students in a block section will complete two team activities each week. For one of these two activities, your team should arrange and participate in a one hour synchronous video meeting. For the other team activity, your team may meet in a synchronous video meeting or your team may collaborate, ask and answer questions, and share code in Microsoft Teams.

As your team works through a team assignment in a synchronous video meeting, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Purpose

Improve your understanding of calling built-in Python functions and calling functions and methods that are in a standard Python module.

Problem Statement

You work for a retail store that wants to increase sales on Tuesday and Wednesday, which are the store's slowest sales days. On Tuesday and Wednesday, if a customer's subtotal is \$50 or greater, the store will discount the customer's subtotal by 10%.

Assignment

Work as a team to write a Python program named `discount.py` that gets a customer's subtotal as input and gets the current day of the week from your computer's operating system. Your program must **not** ask the user to enter the day of the week. Instead, it must get the day of the week from your computer's operating system.

If the subtotal is \$50 or greater and today is Tuesday or Wednesday, your program must subtract 10% from the subtotal. Your program must then compute the total amount due by adding sales tax of 6% to the subtotal. Your program must print the discount amount if applicable, the sales tax amount, and the total amount due.

Core Requirements

1. Your program asks the user for the subtotal but does **not** ask the user for the day of the week. Your program gets the day of the week from your computer's operating system.
2. Your program correctly computes and prints the discount amount if applicable.
3. Your program correctly computes and prints the sales tax amount and the total amount due.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add code to your program that the computer will execute if today is Tuesday or Wednesday and the customer is not purchasing enough to receive the discount. This added code should compute and print the difference between \$50 and the subtotal which is the additional amount the customer would need to purchase in order to receive the discount.

- Near the beginning of your program replace the code that asks the user for the subtotal with a loop that repeatedly asks the user for a price and a quantity and computes the subtotal. This loop should repeat until the user enters "0" for the price.

Helpful Documentation

- The [prepare content](#) for this lesson explains how to call a function and a method.
- The [datetime.now\(\) method](#) from the standard Python `datetime` module will get the current date and time from your computer's operating system. Here is an excerpt from the official reference for the `datetime.now` method:

```
datetime.now(tz=None)
    Return the current local date and time.

    tz is optional, but if it is not None, it must be a tzinfo (time zone
    information) object
```

The [weekday\(\) method](#) will get an integer that represents the day of the week from a `datetime` object. Here is an excerpt from the official documentation for the `weekday` method:

```
dt.weekday()
    Return the day of the week as an integer, where Monday is 0 and
    Sunday is 6.
```

These two Microsoft videos explain how to use methods from the standard `datetime` module.

[Date data types](#) (8 minutes)

[Demonstration: Dates](#) (9 minutes)

The following Python code imports the `datetime` class from the `datetime` module, calls the `datetime.now` method to get the current date and time from a computer's operating system, and then calls the `weekday` method to get the day of the week as an integer.

```
1 # Import the datetime class from the datetime
2 # module so that it can be used in this program.
3 from datetime import datetime
4
5 # Call the now() method to get the current
6 # date and time as a datetime object from
7 # the computer's operating system.
8 current_date_and_time = datetime.now()
9
10 # Call the weekday() method to get the day of the
```

```
11 # week from the current_date_and_time object.  
12 day_of_week = current_date_and_time.weekday()  
13  
14 # Print the day of the week for the user to see.  
15 print(day_of_week)
```

```
> python day_of_week.py  
4
```

After the computer executes [line 8](#) in the above code, the *current_date_and_time* variable will hold the current date and time. After the computer executes [line 12](#), the *day_of_week* variable will hold 0 if today is Monday, 1 if today is Tuesday, and so on to 6 if today is Sunday.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. If today is any day except Tuesday or Wednesday, run your program using the inputs shown below. Ensure that your program's output matches the output shown below.

```
> python discount.py  
Please enter the subtotal: 42.75  
Sales tax amount: 2.56  
Total: 45.31  
  
> python discount.py  
Please enter the subtotal: 55.20  
Sales tax amount: 3.31  
Total: 58.51
```

2. If today is Tuesday or Wednesday, run your program using the input shown below. Ensure that your program's output matches output shown below.

```
> python discount.py  
Please enter the subtotal: 42.75  
Sales tax amount: 2.56  
Total: 45.31  
  
> python discount.py  
Please enter the subtotal: 55.20  
Discount amount: 5.52  
Sales tax amount: 2.98  
Total: 52.66
```

3. Is there a simple way to test your program for all the days of the week without testing on seven consecutive days and without changing your computer's operating system date? Hint: In your program immediately after these three lines of code:

```
# Call the weekday() method to get the day of the  
# week from the current_date_and_time object.
```

```
day_of_week = current_date_and_time.weekday()
```

temporarily add a line of code like this one:

```
day_of_week = 2
```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your program to the [sample solution](#) or the [stretch solution](#). Please **do not look at the sample solutions** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Ponder

During this assignment, you wrote code that calls the `datetime.now()` and `datetime.weekday()` methods. What did these two methods do in your program? If these two methods didn't exist, would this assignment have been more difficult to complete?

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson02/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 You work for a retail store that wants to increase sales on Tuesday
4 and Wednesday, which are the store's slowest sales days. On Tuesday
5 and Wednesday, if a customer's subtotal is greater than $50, the
6 store will discount the customer's purchase by 10%.
7 """
8
9
10 # Import the datetime module so that
11 # it can be used in this program.
12 from datetime import datetime
13
14 # The discount rate is 10% and the sales tax rate is 6%.
15 DISC_RATE = 0.10
16 SALES_TAX_RATE = 0.06
17
18 # Get the subtotal from the user.
19 subtotal = float(input("Please enter the subtotal: "))
20
21 # Call the now() method to get the current
22 # date and time as a datetime object from
23 # the computer's operating system.
24 current_date_and_time = datetime.now()
25
26 # Call the weekday() method to get the day of the
27 # week from the current_date_and_time object.
28 weekday = current_date_and_time.weekday()
29
30 # If the subtotal is greater than 50 and today is
31 # Tuesday or Wednesday, compute the discount amount.
32 if subtotal >= 50 and (weekday == 1 or weekday == 2):
33     discount = round(subtotal * DISC_RATE, 2)
34     print(f"Discount amount: {discount:.2f}")
35     subtotal -= discount
36
37 # Compute the sales tax. Notice that we compute the sales tax
38 # after computing the discount because the customer does not
39 # pay sales tax on the full price but on the discounted price.
40 sales_tax = round(subtotal * SALES_TAX_RATE, 2)
41 print(f"Sales tax amount: {sales_tax:.2f}")
42
43 # Compute the total by adding the subtotal and the sales tax.
44 total = subtotal + sales_tax
45
46 # Display the total for the user to see.
47 print(f"Total: {total:.2f}")
```

lesson02/teach_stretch.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 You work for a retail store that wants to increase sales on Tuesday
4 and Wednesday, which are the store's slowest sales days. On Tuesday
5 and Wednesday, if a customer's subtotal is greater than $50, the
6 store will discount the customer's purchase by 10%.
7 """
8
9
10 # Import the datetime module so that
11 # it can be used in this program.
12 from datetime import datetime
13
14 # The discount rate is 10% and the sales tax rate is 6%.
15 DISC_RATE = 0.10
16 SALES_TAX_RATE = 0.06
17
18 subtotal = 0
19
20 print("Enter the price and quantity for each item.")
21 price = 1
22 while price != 0:
23     # Get the price from the user.
24     price = float(input("Please enter the price: "))
25     if price != 0:
26         # Get the quantity from the user.
27         quantity = int(input("Please enter the quantity: "))
28
29         subtotal += price * quantity
30
31     # Print a blank line.
32     print()
33
34 # Round the subtotal to two digits after
35 # the decimal and print the subtotal.
36 subtotal = round(subtotal, 2)
37 print(f"Subtotal: {subtotal:.2f}")
38 print()
39
40 # Call the now() method to get the current
41 # date and time as a datetime object from
42 # the computer's operating system.
43 current_date_and_time = datetime.now()
44
45 # Call the weekday() method to get the day of the
46 # week from the current_date_and_time object.
47 weekday = current_date_and_time.weekday()
48
49 # if the subtotal is greater than 50 and today is
50 # Tuesday or Wednesday, compute the discount amount.
51 if weekday == 1 or weekday == 2:
52     if subtotal < 50:
53         lacking = 50 - subtotal
54         print("To receive the discount, add"
55               f" {lacking:.2f} to your order.")
56     else:
57         discount = round(subtotal * DISC_RATE, 2)
```

```
58     print(f"Discount amount: {discount:.2f}")
59     subtotal -= discount
60
61 # Compute the sales tax. Notice that we compute the sales tax
62 # after computing the discount because the customer does not
63 # pay sales tax on the full price but on the discounted price.
64 sales_tax = round(subtotal * SALES_TAX_RATE, 2)
65 print(f"Sales tax amount: {sales_tax:.2f}")
66
67 # Compute the total by adding the subtotal and the sales tax.
68 total = subtotal + sales_tax
69
70 # Display the total for the user to see.
71 print(f"Total: {total:.2f}")
```

O2 Prove: Calling Functions

Purpose

Prove that you can write a Python program that calls functions and methods to get the current date and to append values to a text file.

Problem Statement

Many companies wish to understand the needs and wants of their customers more deeply so the company can create products that fill those needs and wants. One way to understand customers more deeply is to record the values entered by customers while they are using a program and then to analyze those values. One common way to record values is for a program to store them in a file.

Assignment

The previous lesson's [prove milestone](#) required you to write a program named `tire_volume.py` that computes the approximate volume of air inside a tire. Add code near the end of that program that does the following:

1. Gets the current date from the computer's operating system.
2. Opens a text file named `volumes.txt` for appending.
3. Appends to the end of the `volumes.txt` file one line of text that contains the following five values:
 - a. current date
 - b. width of the tire
 - c. aspect ratio of the tire
 - d. diameter of the wheel
 - e. volume of the tire

* For all assignments in CSE 111, please write your program in a file named as the assignment states. Also, if an assignment requires your program to read from a file or write to a file, please use the filename stated in the assignment. If you name your program or a data file differently than stated in an assignment, it will be harder for the graders to score your submitted assignment.

This assignment requires you to name your program `tire_volume.py` and requires your program to write to a file named `volumes.txt`. Please use those names.

Helpful Documentation

- The [prepare content](#) for this lesson explains how to call a function and a method.
- The [datetime.now\(\) method](#) from the standard Python `datetime` module will get the current date and time from your computer's operating system. Here is an excerpt from the official reference for the `datetime.now` method:

```
datetime.now(tz=None)
    Return the current local date and time.

    tz is optional, but if it is not None, it must be a tzinfo (time zone
    information) object
```

These two Microsoft videos explain how to use methods from the standard `datetime` module.

[Date data types](#) (8 minutes)

[Demonstration: Dates](#) (9 minutes)

The following Python code imports the `datetime` class from the `datetime` module and calls the `datetime.now` method to get the current date and time from a computer's operating system. Then it uses an f-string to format and print the current date and time.

```
1 # Import the datetime class from the datetime
2 # module so that it can be used in this program.
3 from datetime import datetime
4
5 # Call the now() method to get the current
6 # date and time as a datetime object from
7 # the computer's operating system.
8 current_date_and_time = datetime.now()
9
10 # Print only the date part of the current date and time.
11 print(f"{current_date_and_time:%Y-%m-%d}")
```

After the computer executes [line 8](#) in the above code, the variable `current_date_and_time` will hold the current date and time. Within the f-string at [line 11](#), the string sequences that begin with the percent symbol (%) are called format codes. The format codes and their meaning are listed in the official Python reference about [Basic date and time types](#). When executed, the previous example code will print the current date and time to the terminal window like this:

```
> python date_example.py
2020-07-24
```

- The built-in [open\(\) function](#) opens a file for reading or writing. Here is an excerpt from the official documentation for the `open` function:

```
open(filename, mode="rt")  
Open a file and return a corresponding file object.
```

filename is the name of the file to be opened.

mode is an optional string that specifies the mode in which the file will be opened. It defaults to "rt" which means open for reading in text mode. Other common values are "wt" for writing a text file (truncating the file if it already exists), and "at" for appending to the end of a text file that already exists (and creating and writing to a text file that doesn't exist).

- The built-in [print\(\) function](#) prints text to a terminal window or to a text file. Here is an excerpt from the official documentation for the print function:

```
print(*objects, sep=" ", end="\n", file=sys.stdout, flush=False)  
Print objects to the text stream file, separated by sep and followed by  
end. sep, end, file and flush, if present, must be given as named  
arguments.
```

- You may want to review the [lesson 11 prepare content from CSE 110](#) which includes several examples for opening and reading from a text file.

The following example code calls the open function at [line 6](#) to open a file named `cities.txt` for **appending text ("at")**. The code then calls the print function two times at [lines 9](#) and [10](#) to print two lines of text to the `cities.txt` file.

```
1 city_name = "Accra"  
2 elevation = 61  
3 population = 4200000  
4  
5 # Open a text file named cities.txt in append mode.  
6 with open("cities.txt", "at") as cities_file:  
7  
8     # Print a city's name and information to the file.  
9     print(city_name, file=cities_file)  
10    print(f"\n{elevation}, {population}", file=cities_file)
```

Notice in the previous example at [line 6](#) that the call to the open function is inside a `with` statement. Opening a file in a `with` statement, ensures that the computer will automatically close the file when the computer finishes executing the code inside the `with` block.

The variable `cities_file`, which is created at [line 6](#), is a reference to the open `cities.txt` file. At both lines [lines 9](#) and [10](#), the `cities_file` is a named argument that causes the `print` function to print to the `cities.txt` file instead of printing to the terminal window.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program using the inputs shown below. Ensure that your program's output matches the output shown below.

```
> python tire_volume.py
Enter the width of the tire in mm (ex 205): 185
Enter the aspect ratio of the tire (ex 60): 50
Enter the diameter of the wheel in inches (ex 15): 14

The approximate volume is 24.09 liters
```

2. Use VS Code to open the `volumes.txt` file and verify that the last line of text in the file looks like this, except the date will be different:
2020-03-18, 185, 50, 14, 24.09
3. Run your program using the inputs shown below. Ensure that your program's output matches the output shown below.

```
> python tire_volume.py
Enter the width of the tire in mm (ex 205): 205
Enter the aspect ratio of the tire (ex 60): 60
Enter the diameter of the wheel in inches (ex 15): 15

The approximate volume is 39.92 liters
```

4. Use VS Code to open the `volumes.txt` file and verify that the last two lines of text in the file look like this, except the dates will be different:
2020-03-18, 185, 50, 14, 24.09
2020-04-16, 205, 60, 15, 39.92

Exceeding the Requirements

If your program fulfills the requirements for this assignment as described in the previous prove milestone and the Assignment section above, your program will earn 93% of the possible points. In order to earn the remaining 7% of points, you will need to add one or more features to your program so that it exceeds the requirements. Here are a few suggestions for additional features that you could add to your program if you wish.

- Find tire prices for four or more tire sizes online. Add a set of `if ... elif ... else` statements in your program that use the tire width, tire aspect ratio, and wheel diameter that the user enters to find a price and then print the price.
- After your program prints the tire volume to the terminal window, your program should ask the user if she wants to buy tires with the dimensions that she entered.

If the user answers "yes", your program should ask for her phone number and store her phone number in the `volumes.txt` file.

Ponder

During this assignment, you wrote code that gets the current date from your computer's operating system and that writes text to a file on your computer's hard drive. The code that you wrote calls the `datetime.now()`, `open()`, and `print()` functions. Would this assignment have been more difficult if those three functions didn't exist? Now that you know what functions are and how to call them in your code, are you able to write more complex programs?

Submission

To submit your program, return to I-Learn and do these two things:

1. Upload your program (the `.py` file) for feedback.
2. Add a submission comment that specifies the grading category that best describes your program along with a one or two sentence justification for your choice. The grading criteria are:
 - a. Some attempt made
 - b. Developing but significantly deficient
 - c. Slightly deficient
 - d. Meets requirements
 - e. Exceeds requirements

03 Prepare: Writing Functions

Because most useful computer programs are very large, programmers divide their programs into parts. Dividing a program into parts makes it easier to write, debug, and understand the program. A programmer can divide a Python program into modules, classes, and functions. In this lesson and the next, you will learn how to write your own functions.

Videos

Watch the following four videos from Microsoft about writing functions:

[Introducing Functions](#) (10 minutes)

[Demonstration: Functions](#) (8 minutes)

[Parameterized Functions](#) (7 minutes)

[Demonstration: Parameterized Functions](#) (5 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is a Function?

A **function** is a group of statements that together perform one task. Broadly speaking, there are four types of functions in Python which are:

1. Built-in functions
2. Standard library functions
3. Third-party functions
4. User-defined functions

In the previous lesson, you learned how to call the first two types of functions. In lesson 5, you will learn how to install third-party modules and call third-party functions. In this lesson, you will learn how to write and call user-defined functions.

What Is a User-Defined Function?

A **user-defined function** is a function that is not a built-in function, a standard function, or a third-party function. A user-defined function is written by a programmer like yourself as part of a program. For some students the term "user-defined function" is confusing because the user of a program doesn't define the function. Instead, the programmer (you) define user-defined functions. Perhaps a more correct term is programmer-defined function. Writing user-defined functions has several advantages, including:

1. making your code more reusable
2. making your code easier to understand and debug
3. making your code easier to change and add capabilities

How to Write a User-Defined Function

To write a user-defined function in Python, simply type code that matches this template:

```
def function_name(param1, param2, ... paramN):
    """documentation string"""
    statement1
    statement2
    :
    statementN
    return value
```

The first line of a function is called the **header** or **signature**, and it includes the following:

1. the keyword `def` (which is an abbreviation for "define")
2. the function name
3. the parameter list (with the parameters separated by commas)

Here is the header for a function named `draw_circle` that takes three parameters named `x`, `y`, and `radius`:

```
def draw_circle(x, y, radius):
```

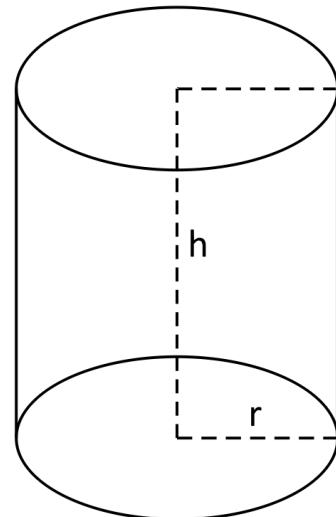
You could read the previous line of code as, "Define a function named `draw_circle` that takes three parameters named `x`, `y`, and `radius`."

The **function name** must start with a letter or the underscore (`_`). The rest of the name must be made of letters, digits (0–9), or the underscore. A function name cannot include spaces or other punctuation. A function name should be meaningful and should describe briefly what the function does. Well-named functions often start with a verb.

The statements inside a function are called the **body** of the function. Just like other block statements in Python, such as `if`, `else`, `while`, and `for`, all of which end with a colon (`:`),

you must indent the statements inside the body of a function. The body of a function should begin with a **documentation string** which is a triple quoted string that describes the function's purpose, parameters and return value. The body of a function may contain as many statements as you wish to write inside of it. However, it is a good idea to limit functions to less than 20 lines of code.

Example 1 contains a function named `print_cylinder_volume()` with no parameters that gets two numbers from the user: *radius* and *height* and uses those numbers to compute the volume of a right circular cylinder and then prints the volume for the user to see.



A right circular cylinder with radius r and height h

```
# Example 1

import math

# Define a function named print_cylinder_volume.
def print_cylinder_volume():
    """Compute and print the volume of a cylinder.
    Parameters: none
    Return: nothing
    """
    # Get the radius and height from the user.
    radius = float(input("Enter the radius of a cylinder: "))
    height = float(input("Enter the height of a cylinder: "))

    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height

    # Print the volume of the cylinder.
    print(f"Volume: {volume:.2f}")
```

Because the `print_cylinder_volume` function in example 1 doesn't accept parameters, it must be called without any arguments like this:

```
print_cylinder_volume()
```

How to Make a User-Defined Function Reusable

Because the `print_cylinder_volume` function in example 1 gets input from a user and prints its results to a terminal window, it can be used only in a program that runs when a user is present. It cannot be used in a program that runs automatically and gets input from a file or the network or a sensor. In other words, the `print_cylinder_volume` function in example 1 is not reusable in other programs. The most **reusable functions** are ones that take parameters, perform calculations, and return a result but *do not perform user input and output*.

The parameter list in a function's header contains data stored in variables that the function needs to complete its task. A **parameter** is a variable whose value comes from outside the function. One way to get input into a function is to ask the user for input by calling the built-in Python `input` function. Another way to get input into a function is through the function's parameters. Getting input through parameters is much more flexible than asking the user for input because the input through parameters can come from the user or a file on a hard drive or the network or a sensor or even another function.

Example 2 contains another version of the `print_cylinder_volume` function. This second version doesn't get the radius and height from the user. Instead, it gets input through its two parameters named `radius` and `height`.

```
# Example 2

import math

# Define a function named print_cylinder_volume.
def print_cylinder_volume(radius, height):
    """Compute and print the volume of a cylinder.
    Parameters
        radius: the radius of the cylinder
        height: the height of the cylinder
    Return: nothing
    """
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height

    # Print the volume of the cylinder.
    print(volume)
```

Because the second version of the `print_cylinder_volume` function accepts two parameters, it must be called with two arguments like this:

```
print_cylinder_volume(2.5, 4.1)
```

To **return** a result from a function, simply type the keyword `return` followed by whatever result you want returned to the calling function. Example 3 contains a third version of the cylinder volume function. Notice that the version in example 3 returns the volume instead of printing it, which makes the function more reusable. Notice also in example 3 that we changed the name of the function from `print_cylinder_volume` to

`compute_cylinder_volume` because this version doesn't print the volume but instead returns it.

```
# Example 3

import math

# Define a function named computer_cylinder_volume.
def compute_cylinder_volume(radius, height):
    """Compute and return the volume of a cylinder.

    Parameters
        radius: the radius of the cylinder
        height: the height of the cylinder
    Return: the volume of the cylinder
    """

    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height

    # Return the volume of the cylinder so that the
    # volume can be used somewhere else in the program.
    return volume
```

Many functions that you've used in the past such as `input`, `float`, and `round`, return a result. When a function returns a result, we usually write code to store that returned result in a variable to use later in the program like this:

```
text = input("Please enter your name: ")
```

Because the `compute_cylinder_volume` function in example 3 accepts two parameters and returns a result, it could be called like this:

```
volume = compute_cylinder_volume(2.5, 4.1)
```

The `main` User-Defined Function

In all previous Python programs that you wrote in CSE 110 and 111, you wrote statements that were not in a function like the simple program in example 4.

```
1 # Example 4
2
3 import math
4
5 # Get the radius and height from the user.
6 radius = float(input("Enter the radius of a cylinder: "))
7 height = float(input("Enter the height of a cylinder: "))
8
9 # Compute the volume of the cylinder.
10 volume = math.pi * radius**2 * height
11
12 # Print the volume of the cylinder.
13 print(f"Volume: {volume:.2f}")
```

```
> python example_4.py
Enter the radius in centimeters: 
Enter the height in centimeters: 
Volume: 226.19
```

Writing statements outside a function can lead to poor organization within a large program. Professional software developers write statements inside a function whenever possible. Beginning with this lesson, you will write nearly all statements inside a user-defined function. Also, in each program, you will write a user-defined function named `main`, which will contain the beginning statements of the program. In addition, in each program you will write one or more user-defined functions that have parameters, perform calculations and other useful work, and return a result to the call point. Example 5 contains the same Python program as example 4 except most of the statements are inside a user-defined function named `main`.

```
1 # Example 5
2
3 import math
4
5 # Define a function named main.
6 def main():
7     # Get the radius and height from the user.
8     radius = float(input("Enter the radius of a cylinder: "))
9     height = float(input("Enter the height of a cylinder: "))
10
11    # Compute the volume of the cylinder.
12    volume = math.pi * radius**2 * height
13
14    # Print the volume of the cylinder.
15    print(f"Volume: {volume:.2f}")
16
17 # Start this program by
18 # calling the main function.
19 main()
```

```
> python example_5.py
Enter the radius in centimeters: 
Enter the height in centimeters: 
Volume: 226.19
```

Notice the call to the `main` function at line 19 in example 5. Without that call to the `main` function, when we run the program, the program will do nothing. In all of your future programs in CSE 111 you will write a user-defined function named `main` and will have a call to `main` at the bottom of the program.

A Complete Program with User-Defined Functions

If you look closely at the code in examples 1 and 5, you will realize that both programs have the same problem, namely both the `print_cylinder_volume` function in example 1 and the `main` function in example 5 are not reusable because both of them get input from a user and print to a terminal window. A better way to write the program in examples 1

and 5 is to separate the program into two functions, one named `main` and one named `compute_cylinder_volume` as shown in example 6.

Example 6 contains a complete program with two functions, the first named `main` at line 6 and the second named `compute_cylinder_volume` at line 20. At line 13, the `main` function calls the `compute_cylinder_volume` function. Notice that the `compute_cylinder_volume` function gets its input through parameters and returns a result which makes this function reusable in other programs, including programs that run automatically without a user.

```
1 # Example 6
2
3 import math
4
5 # Define the main function.
6 def main():
7     # Get a radius and a height from the user.
8     radius = float(input("Enter the radius of a cylinder: "))
9     height = float(input("Enter the height of a cylinder: "))
10
11    # Call the compute_cylinder_volume function and store
12    # its return value in a variable to use later.
13    volume = compute_cylinder_volume(radius, height)
14
15    # Print the volume of the cylinder.
16    print(f"Volume: {volume:.2f}")
17
18
19 # Define a function that accepts two parameters.
20 def compute_cylinder_volume(radius, height):
21     """Compute and print the volume of a cylinder.
22     Parameters
23         radius: the radius of the cylinder
24         height: the height of the cylinder
25     Return: the volume of the cylinder
26     """
27     # Compute the volume of the cylinder.
28     volume = math.pi * radius**2 * height
29
30     # Return the volume of the cylinder so that the
31     # volume can be used somewhere else in the program.
32     # The returned result will be available wherever
33     # this function was called.
34     return volume
35
36
37 # Start this program by
38 # calling the main function.
39 main()
```

```
> python example_6.py
Enter the radius in centimeters: 3
Enter the height in centimeters: 8
Volume: 226.19
```

The most reusable functions are ones that have parameters, perform calculations, and return a result but *do not perform user input and output*. In the previous code example, there are two functions named `main` and `compute_cylinder_volume`. The `main` function is certainly useful in the program, but it is not reusable in other programs because it gets user input and prints the result for the user to see. The `compute_cylinder_volume` function is very reusable in another program because it doesn't get user input or print output. Instead, it takes two parameters, performs a calculation, and returns a result to the calling function. The `compute_cylinder_volume` function is so reusable that it could be included in a library of functions that compute the area and volume of 2-D and 3-D geometric shapes.

What Happens When the Computer Calls a Function?

Some students have trouble visualizing what happens when the computer calls (executes) a function. The following diagram contains the same program as example 6. The circled numbers show the order in which the events happen in the computer. The green numbers and arrows in the diagram show the order in which the computer executes statements in the program. The blue numbers and arrows show how data flows from arguments into parameters and from a returned result to a variable.

```

import math

# Define the main function.
def main():
    # Get a radius and a height from the user.
    r = float(input("Enter the radius of a cylinder: "))
    h = float(input("Enter the height of a cylinder: "))

    # Call the compute_cylinder_volume function and store
    # its return value in a variable to use later.
    v = compute_cylinder_volume(r, h)

    # Print the volume of the cylinder.
    print(f"Volume: {v:.2f}")

# Define a function that accepts two parameters.
def compute_cylinder_volume(radius, height):
    """Compute and print the volume of a cylinder.
    Parameters
        radius: the radius of the cylinder
        height: the height of the cylinder
    Return: the volume of the cylinder
    """
    # Compute the volume of the cylinder.
    volume = math.pi * radius**2 * height

    # Return the volume of the cylinder so that it
    # can be used at the call point in this program.
    return volume

# Start this program by
# calling the main function.
main()

```

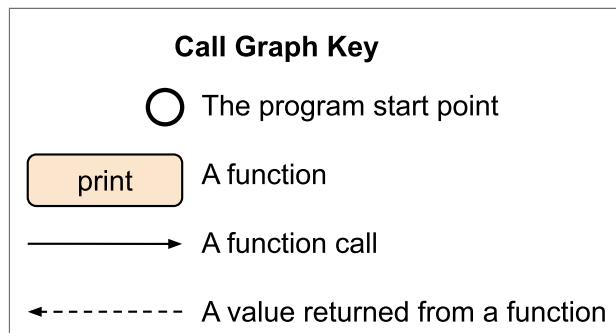
A computer will execute the statements in the previous diagram in the following order:

- The statement at (1) is not inside a function, so the computer executes it when the program begins. The statement at (1) is a call to the `main` function which causes the computer to begin executing the statements inside `main` at (2).
- At (2), the computer gets two numbers from the user.

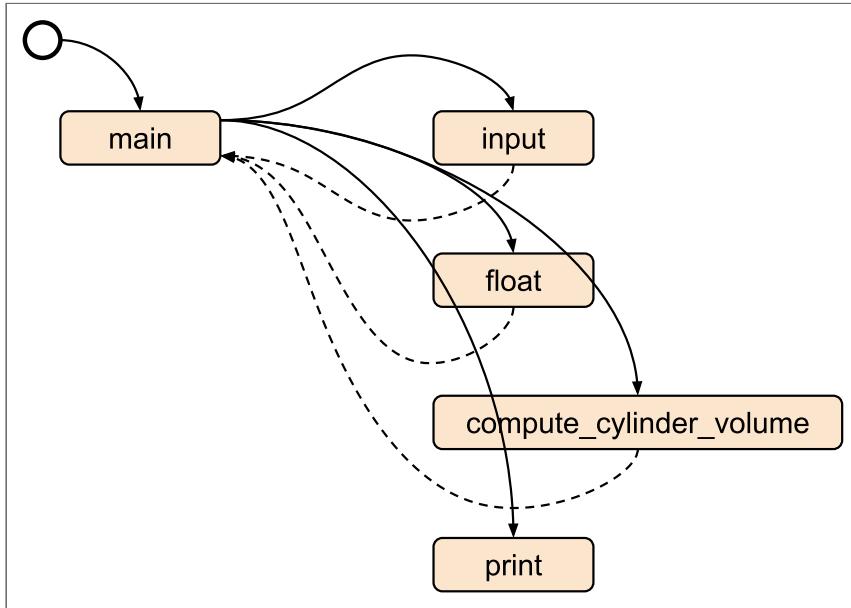
- C. The statement at (3) is a call to the `compute_cylinder_volume` function which causes the computer to copy the values in the arguments `r` and `h` into the parameters `radius` and `height` respectively and then begin executing the statements inside the `compute_cylinder_volume` function at (5).
- D. At (5), the computer computes the volume of a cylinder.
- E. The statement at (6) is a return statement which causes the computer to stop executing the `compute_cylinder_volume` function, to return the computed volume to the call point at (3), and to resume executing statements at the call point.
- F. At the call point (3), the computer stores the returned value in the variable named `v`.
- G. At (8), the computer prints the value that is in the `volume` variable for the user to see. This is the last statement in the `main` function, so after executing it, the computer resumes executing the statements after the call point (1) to `main`.
- H. At (9), there are no more statements after the call to `main`, so the computer terminates the program.

Call Graphs

A **call graph** is a diagram that shows function calls and returns within a program. A call graph can help you visualize how a program is divided into functions. Within a call graph, the unfilled circle shows where the computer begins executing a program. A rounded rectangle represents a function. A solid arrow represents a call from one function to another function. A dashed arrow represents a value returned from a called function to the calling function.



The call graph below shows the function calls and returns for the program in example 6. From the call graph, we see that the computer begins executing the program by calling the `main` function. While executing the `main` function, the computer calls the `input` and `float` functions. Then the computer calls the `compute_cylinder_volume` function. Finally the computer calls the `print` function. In the call graph we can see that the `main` and `print` functions don't return a value. The `print` function prints results for the user to see, but it doesn't return anything.



Summary

A function is a group of statements that together perform one task. A user-defined function is a function written by a programmer like you. To write a user-defined function, write code that follows this template:

```

def function_name(param1, param2, ... paramN):
    """documentation string"""
    statement1
    statement2
    :
    statementN
    return value

```

To call a user-defined function, write code that follows this template:

```

variable_name = function_name(arg1, arg2, ... argN)

```

The most reusable functions are ones that take parameters, perform calculations, and return a result but *do not perform user input and output*. All of your future programs in CSE 111 will have a user-defined function named `main` and will have a call to `main` at the bottom of the program.

It is extremely important that you can write and call functions. After watching the videos and reading this prepare content, if the concepts still seem confusing or vague to you, watch the videos and read the list of concepts **again**.

03 Checkpoint: Writing Functions

Purpose

Check your understanding of writing your own functions with parameters and then calling those functions with arguments.

Problem Statement

Many vehicle owners record the fuel efficiency of their vehicles as a way to track the health of the vehicle. If the fuel efficiency of a vehicle suddenly drops, there is probably something wrong with the engine or drive train of the vehicle. In the United States, fuel efficiency for gasoline powered vehicles is calculated as miles per gallon. In most other countries, fuel efficiency is calculated as liters per 100 kilometers.

The formula for computing fuel efficiency in miles per gallon is the following:

$$mpg = \frac{end - start}{gallons}$$

where *start* and *end* are both odometer values in miles and *gallons* is a fuel amount in U.S. gallons.

The formula for converting miles per gallon to liters per 100 kilometers is the following:

$$lp100k = \frac{235.215}{mpg}$$

Assignment

Write a Python program that asks the user for three numbers:

1. A starting odometer value in miles
2. An ending odometer value in miles
3. An amount of fuel in gallons

Your program must calculate and print fuel efficiency in both miles per gallon and liters per 100 kilometers. Your program must have three functions named as follows:

1. `main`

2. miles_per_gallon
3. lp100k_from_mpg

All user input and printing must be in the main function. In other words, the miles_per_gallon and lp100k_from_mpg functions must not call the the input or print functions.

Helpful Documentation

The [prepare content for the previous lesson](#) explains how to call a function.

The [prepare content for this lesson](#) explains how to write a function.

Steps

Copy and paste the following code into a new program named fuel_usage.py. Use the pasted code as a design as you write your program. Write code for each of the three functions.

```
def main():
    # Get an odometer value in U.S. miles from the user.

    # Get another odometer value in U.S. miles from the user.

    # Get a fuel amount in U.S. gallons from the user.

    # Call the miles_per_gallon function and store
    # the result in a variable named mpg.

    # Call the lp100k_from_mpg function to convert the
    # miles per gallon to liters per 100 kilometers and
    # store the result in a variable named lp100k.

    # Display the results for the user to see.
    pass

def miles_per_gallon(start Miles, end Miles, amount Gallons):
    """Compute and return the average number of miles
    that a vehicle traveled per gallon of fuel.

    Parameters
        start Miles: An odometer value in miles.
        end Miles: Another odometer value in miles.
        amount Gallons: A fuel amount in U.S. gallons.
    Return: Fuel efficiency in miles per gallon.
    """
    return
```

```
def lp100k_from_mpg(mpg):
    """Convert miles per gallon to liters per 100
    kilometers and return the converted value.

    Parameter mpg: A value in miles per gallon
    Return: The converted value in liters per 100km.
    """
    return

# Call the main function so that
# this program will start executing.
main()
```

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

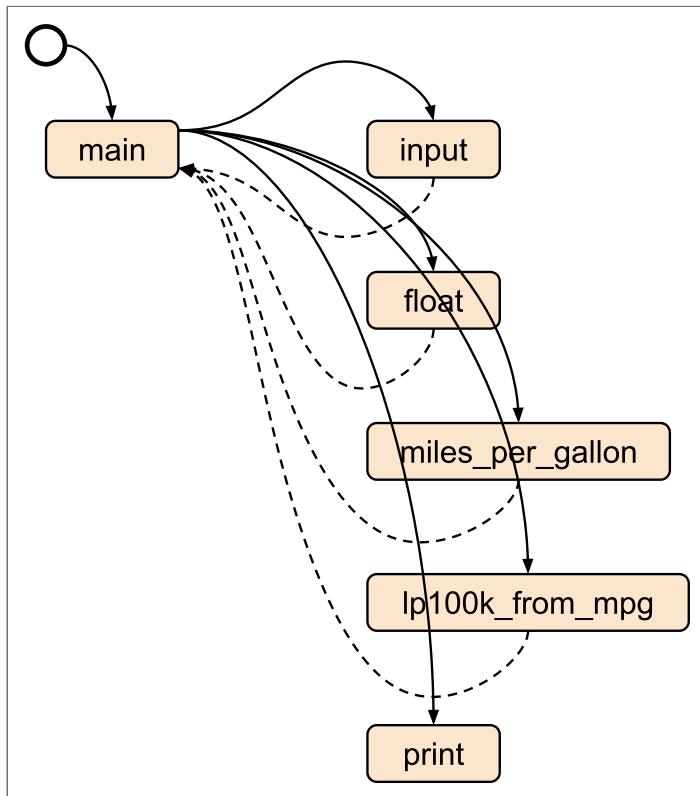
```
> python fuel_usage.py
Enter the first odometer reading (miles): 30462
Enter the second odometer reading (miles): 30810
Enter the amount of fuel used (gallons): 11.2
31.1 miles per gallon
7.57 liters per 100 kilometers
```

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample program by calling the `main` function. While executing the `main` function, the computer calls the `input` and `float` functions. Then the computer calls the `miles_per_gallon` and `lp100k_from_mpg` functions. Finally the computer calls the `print` function which is the end of the program.



Ponder

After you finish this assignment, congratulate yourself because you wrote a Python program with three user-defined functions named `main`, `miles_per_gallon`, and `lp100k_from_mpg`. Is it important that you know how to write your own functions? Why?

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson03/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 Write a Python program named fuel_usage.py that asks the user
4 for three numbers:
5 1. A starting odometer value in miles
6 2. An ending odometer value in miles
7 3. A amount of fuel in gallons
8
9 Your program must calculate and print fuel efficiency in both
10 miles per gallon and liters per 100 kilometers. Your program
11 must have three functions named as follows:
12 1. main
13 2. miles_per_gallon
14 3. lp100k_from_mpg
15
16 All user input and printing must be in the main function. In other
17 words, the miles_per_gallon and lp100k_from_mpg functions must not
18 call the the input or print functions.
19 """
20
21 def main():
22     # Get an odometer value in U.S. miles from the user.
23     start_miles = float(input(
24         "Enter the first odometer reading (miles): "))
25
26     # Get another odometer value in U.S. miles from the user.
27     end_miles = float(input(
28         "Enter the second odometer reading (miles): "))
29
30     # Get a fuel amount in U.S. gallons from the user.
31     amount_gallons = float(input(
32         "Enter the amount of fuel used (gallons): "))
33
34     # Call the miles_per_gallon function and store
35     # the result in a variable named mpg.
36     mpg = miles_per_gallon(start_miles, end_miles, amount_gallons)
37
38     # Call the lp100k_from_mpg function to convert the
39     # miles per gallon to liters per 100 kilometers and
40     # store the result in a variable named lp100k.
41     lp100k = lp100k_from_mpg(mpg)
42
43     # Display the results for the user to see.
44     print(f"{mpg:.1f} miles per gallon")
45     print(f"{lp100k:.2f} liters per 100 kilometers")
46
47
48 def miles_per_gallon(start_miles, end_miles, amount_gallons):
49     """Compute and return the average number of miles
50     that a vehicle traveled per gallon of fuel.
51     Parameters
52         start_miles: An odometer value in miles.
53         end_miles: Another odometer value in miles.
54         amount_gallons: A fuel amount in U.S. gallons.
55     Return: Fuel efficiency in miles per gallon.
56     """
57
```

```
58     mpg = abs(end_miles - start_miles) / amount_gallons
59     return mpg
60
61
62 def lp100k_from_mpg(mpg):
63     """Convert miles per gallon to liters per 100
64     kilometers and return the converted value.
65     Parameter mpg: A value in miles per gallon
66     Return: The converted value in liters per 100km.
67     """
68     lp100k = 235.215 / mpg
69     return lp100k
70
71
72 # Call the main function so that
73 # this program will start executing.
74 main()
```

03 Team Activity: Writing Functions

Instructions

Each lesson in CSE 111 contains a team activity that is designed to take about one hour to complete. You should prepare for all team activities by completing the preparation material and the individual checkpoint assignment before starting a team activity. The goal of the team activities is for students to work together and teach and learn from each other. As your team completes a team activity, instead of moving through it as quickly as you can, you should help everyone understand the concepts.

Face-to-Face Students

Face-to-face students will complete the team activities in their classroom during class time.

Online Students in a Semester Section

Online students in a semester section will arrange and participate in a one hour synchronous video meeting with your team for each team activity. As your team works through the assignment, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Online Students in a Block Section

Students in a block section complete two lessons each week. This means that online students in a block section will complete two team activities each week. For one of these two activities, your team should arrange and participate in a one hour synchronous video meeting. For the other team activity, your team may meet in a synchronous video meeting or your team may collaborate, ask and answer questions, and share code in Microsoft Teams.

As your team works through a team assignment in a synchronous video meeting, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Purpose

Boost your understanding of writing your own functions with parameters and then calling those functions with arguments.

Problem Statement

Health professionals who are helping a client achieve a healthy body weight, sometimes use two computed measures named body mass index and basal metabolic rate.

From the U.S. Centers for Disease Control and Prevention we read, "Body mass index (BMI) is a person's weight in kilograms divided by the square of their height in meters. BMI can be used to screen for weight categories such as underweight, normal, overweight, and obese that may lead to health problems. However, BMI is not diagnostic of the body fatness or health of an individual." The formula for computing BMI is

$$bmi = \frac{10,000 \text{ weight}}{\text{height}^2}$$

where *weight* is in kilograms and *height* is in centimeters.

Basal metabolic rate (BMR) is the minimum number of calories required daily to keep your body functioning at rest. BMR is different for women and men and changes with age. The revised Harris-Benedict formulas for computing BMR are

$$\text{(women)} \quad bmr = 447.593 + 9.247 \text{ weight} + 3.098 \text{ height} - 4.330 \text{ age}$$

$$\text{(men)} \quad bmr = 88.362 + 13.397 \text{ weight} + 4.799 \text{ height} - 5.677 \text{ age}$$

where *weight* is in kilograms and *height* is in centimeters.

Assignment

Work as a team to write a Python program named `fitness.py` that does the following:

1. Asks the user to enter four values:
 - a. gender
 - b. birthdate in this format: YYYY-MM-DD
 - c. weight in U.S. pounds
 - d. height in U.S. inches
2. Converts the weight from pounds to kilograms (1 lb = 0.45359237 kg).
3. Converts inches to centimeters (1 in = 2.54 cm).
4. Calculates age, BMI, and BMR.
5. Prints age, weight in kg, height in cm, BMI, and BMR.

Helpful Documentation

The [prepare content for the previous lesson](#) explains how to call functions.

The [prepare content for this lesson](#) explains how to write functions.

The official Python reference describes the built-in [round function](#).

Steps

Copy and paste the following code into a new program named `fitness.py`. Use the pasted code as a design as you write your program. Write code for each of the functions except `compute_age`. The `compute_age` function is complete and correct, and you should not change it.

```
# Import datetime so that it can be
# used to compute a person's age.
from datetime import datetime

def main():
    # Get the user's gender, birthdate, height, and weight.

    # Call the compute_age, kg_from_lb, cm_from_in,
    # body_mass_index, and basal_metabolic_rate functions
    # as needed.

    # Print the results for the user to see.
    pass

def compute_age(birth_str):
    """Compute and return a person's age in years.
    Parameter birth_str: a person's birthdate stored
        as a string in this format: YYYY-MM-DD
    Return: a person's age in years.
    """
    # Convert a person's birthdate from a string
    # to a date object.
    birthdate = datetime.strptime(birth_str, "%Y-%m-%d")
    today = datetime.now()

    # Compute the difference between today and the
    # person's birthdate in years.
    years = today.year - birthdate.year

    # If necessary, subtract one from the difference.
    if birthdate.month > today.month or \
        (birthdate.month == today.month and \
         birthdate.day > today.day):
        years -= 1
```

```

        return years

def kg_from_lb(pounds):
    """Convert a mass in pounds to kilograms.
    Parameter pounds: a mass in U.S. pounds.
    Return: the mass in kilograms.
    """
    return

def cm_from_in(inches):
    """Convert a length in inches to centimeters.
    Parameter inches: a length in inches.
    Return: the length in centimeters.
    """
    return

def body_mass_index(weight, height):
    """Compute and return a person's body mass index.
    Parameters
        weight: a person's weight in kilograms.
        height: a person's height in centimeters.
    Return: a person's body mass index.
    """
    return

def basal_metabolic_rate(gender, weight, height, age):
    """Compute and return a person's basal metabolic rate.
    Parameters
        weight: a person's weight in kilograms.
        height: a person's height in centimeters.
        age: a person's age in years.
    Return: a person's basal metabolic rate in kcals per day.
    """
    return

# Call the main function so that
# this program will start executing.

```

Core Requirements

1. Your program contains complete and correct functions named `compute_age`, `kg_from_lb`, and `cm_from_in`.
2. Your program contains complete and correct functions named `body_mass_index` and `basal_metabolic_rate`. To be correct, the `basal_metabolic_rate` function must compute BMR differently for males and females.
3. Your program contains a function named `main` which gets four values from the user, calls the other functions, and prints the results for the user to see.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Modify your program to print the height values in meters instead of centimeters.
2. Modify your program to allow the user to enter weight in British stones and add a function named `kg_from_stone`.
3. Modify your program to allow the user to enter height as U.S. feet and inches.
4. Add something or change something in your program that you think would make your program better, easier for the user, more elegant, or more fun. Be creative.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output is similar* to the output below.

```
> python fitness.py
Please enter your gender (M or F): 
Enter your birthdate (YYYY-MM-DD): 
Enter your weight in U.S. pounds: 
Enter your height in U.S. inches: 
Age (years): 19
Weight (kg): 56.70
Height (cm): 137.2
Body mass index: 30.1
Basal metabolic rate (kcal/day): 1315

> python fitness.py
Please enter your gender (M or F): 
Enter your birthdate (YYYY-MM-DD): 
Enter your weight in U.S. pounds: 
Enter your height in U.S. inches: 
Age (years): 17
Weight (kg): 65.77
Height (cm): 147.3
Body mass index: 30.3
Basal metabolic rate (kcal/day): 1580
```

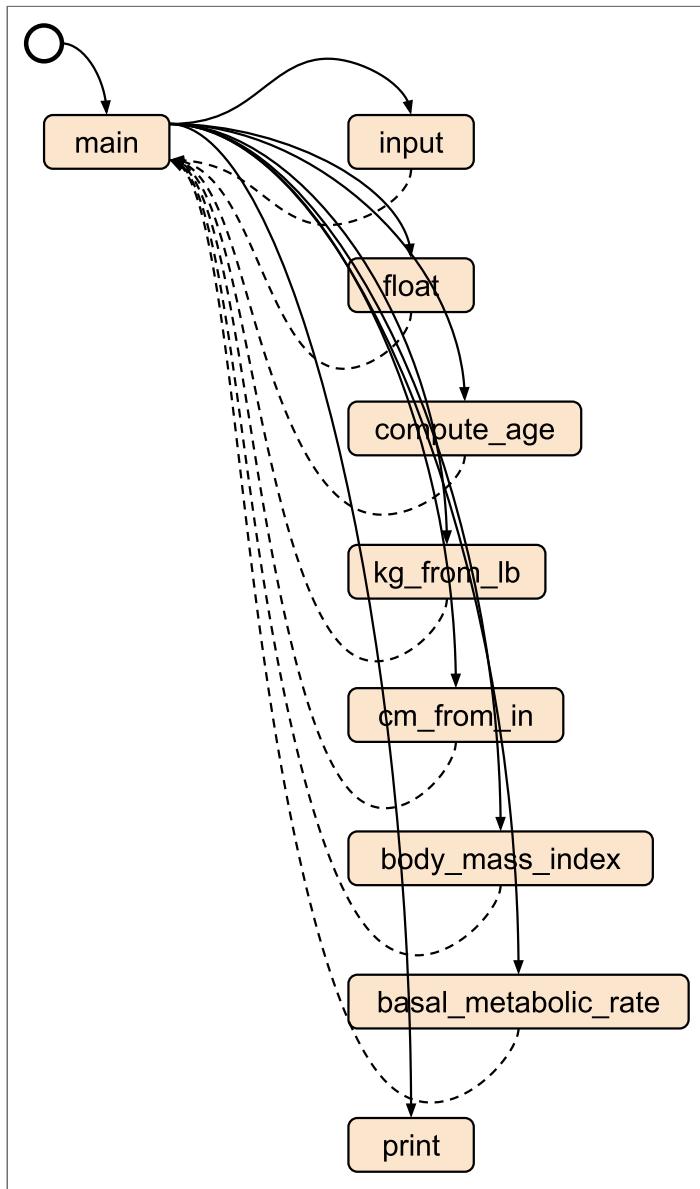
* Note that the output of your program will not be the same for the age and BMR values because a computer executed these sample runs before you executed your program. Therefore, when you run your program, the people will be older than they were when the computer completed these sample runs.

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#). Please ***do not look at the sample solution*** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample program by calling the `main` function. While executing the `main` function, the computer calls the `input` and `float` functions. Then the computer calls the `compute_age`, `kg_from_lb`, `cm_from_in`, `body_mass_index`, and `basal_metabolic_rate` functions. Finally the computer calls the `print` function which is the end of the program.



Ponder

After you finish this assignment, congratulate yourself because you wrote a Python program with six user-defined functions named `main`, `compute_age`, `kg_from_lb`, `cm_from_in`, `body_mass_index`, and `basal_metabolic_rate`. You also wrote code to call those six functions. Why is it important that you know how to write your own functions?

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson03/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 """
4 Write a Python program named fitness.py that does the following:
5 1. Asks the user to enter four values:
6     a. gender
7     b. birthdate in this format: YYYY-MM-DD
8     c. weight in U.S. pounds
9     d. height in U.S. inches
10 2. Converts the weight from pounds to kilograms (1 lb = 0.45359237 kg)
11 3. Converts inches to centimeters (1 in = 2.54 cm).
12 4. Calculates age, BMI, and BMR.
13 5. Prints age, weight in kg, height in cm, BMI, and BMR.
14 """
15
16 # Import datetime so that it can be
17 # used to compute a person's age.
18 from datetime import datetime
19
20
21 def main():
22     # Get the user's gender, birthdate, height, and weight.
23     gender = input("Please enter your gender (M or F): ")
24     birth_str = input("Enter your birthdate (YYYY-MM-DD): ")
25     pounds = float(input("Enter your weight in U.S. pounds: "))
26     inches = float(input("Enter your height in U.S. inches: "))
27
28     # Call the compute_age function to
29     # compute the user's age in years.
30     years = compute_age(birth_str)
31
32     # Call the kg_from_lb function to
33     # convert from pounds to kilograms.
34     kg = kg_from_lb(pounds)
35
36     # Call the cm_from_in function to
37     # convert from inches to centimeters.
38     cm = cm_from_in(inches)
39
40     # Call the body_mass_index function.
41     bmi = body_mass_index(kg, cm)
42
43     # Call the basal_metabolic_rate function.
44     bmr = basal_metabolic_rate(gender, kg, cm, years)
45
46     # Print the results for the user to see.
47     print(f"Age (years): {years}")
48     print(f"Weight (kg): {kg:.2f}")
49     print(f"Height (cm): {cm:.1f}")
50     print(f"Body mass index: {bmi:.1f}")
51     print(f"Basal metabolic rate (kcal/day): {bmr:.0f}")
52
53
54 def compute_age(birth_str):
55     """Compute and return a person's age in years.
56     Parameter birth_str: a person's birthdate stored
57         as a string in this format: YYYY-MM-DD
```

```

58     Return: a person's age in years.
59     """
60     # Convert a person's birthdate from a string
61     # to a date object.
62     birthdate = datetime.strptime(birth_str, "%Y-%m-%d")
63
64     # Compute the difference between today and the
65     # person's birthdate in years.
66     today = datetime.now()
67     years = today.year - birthdate.year
68
69     # If necessary, subtract one from the difference.
70     if birthdate.month > today.month or \
71         (birthdate.month == today.month and \
72          birthdate.day > today.day):
73         years -= 1
74
75     return years
76
77
78 def kg_from_lb(pounds):
79     """Convert a mass in pounds to kilograms.
80     Parameter pounds: a mass in U.S. pounds.
81     Return: the mass in kilograms.
82     """
83     kg = pounds * 0.45359237
84     return kg
85
86
87 def cm_from_in(inches):
88     """Convert a length in inches to centimeters.
89     Parameter inches: a length in inches.
90     Return: the length in centimeters.
91     """
92     cm = inches * 2.54
93     return cm
94
95
96 def body_mass_index(weight, height):
97     """Compute and return a person's body mass index.
98     Parameters
99         weight: a person's weight in kilograms.
100        height: a person's height in centimeters.
101    Return: a person's body mass index.
102    """
103    bmi = weight / (height ** 2) * 10000
104    return bmi
105
106
107 def basal_metabolic_rate(gender, weight, height, age):
108     """Compute and return a person's basal metabolic rate.
109     Parameters
110         weight: a person's weight in kilograms.
111         height: a person's height in centimeters.
112         age: a person's age in years.
113     Return: a person's basal metabolic rate in kcals per day.
114     """
115     if gender.upper() == "F":
116         bmr = 447.593 + 9.247 * weight + 3.098 * height - 4.330 * age
117     else:
118         bmr = 88.362 + 13.397 * weight + 4.799 * height - 5.677 * age

```

```
119     return bmr
120
121
122 # Call the main function so that
123 # this program will start executing.
124 main()
```

03 Prove Milestone: Writing Functions

Purpose

Prove that you can write functions with parameters and call those functions multiple times with arguments.

Problem Statement

Modern computers are capable of performing all sorts of calculations to produce numbers. However, they are also capable of performing calculations to produce art, illustrations, animations, movies, and music.

Assignment

During this prove milestone and the next prove assignment, you will write a Python program that draws a semi-realistic outdoor scene in a computer window. Your program can draw any outdoor scene that you like as long as it meets these requirements:

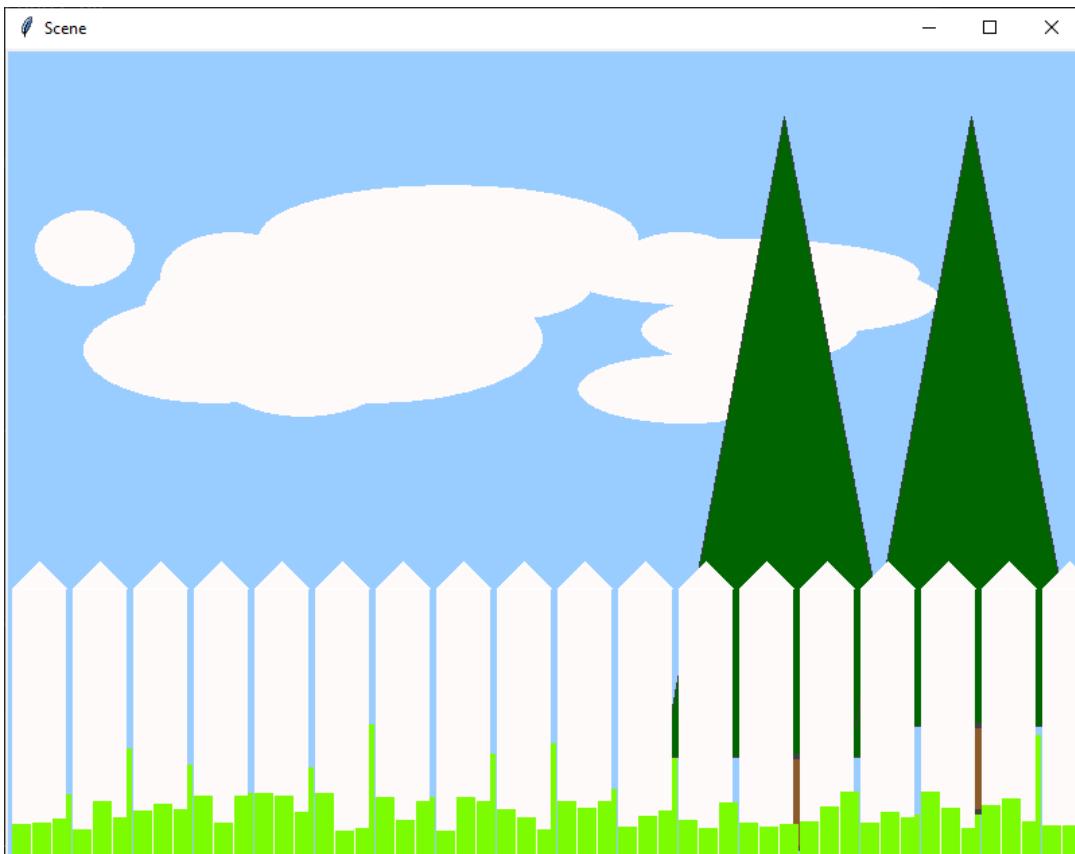
1. The scene must be outdoor and include part of the sky.
2. The sky must have clouds.
3. The scene must include repetitive objects, such as blades of grass, trees, leaves on a tree, birds, flowers, insects, fish, pickets in a fence, dashed lines on a road, buildings, bales of hay, snowmen, snowflakes, or icicles.

Your program must be divided into functions such as `draw_sky`, `draw_cloud`, `draw_ground`, `draw_bird`, `draw_flower`, `draw_insect`, `draw_fish`, or `draw_snowman`. Each of the objects in your scene should be drawn in its own function. To draw the shapes in the scene, your program will call functions in a Python library named Draw 2-D.

During this milestone, you will write code that draws the sky, the ground, and clouds in your scene. During the next prove assignment, you will write code that completes your scene. As you write your program, write it so that it draws objects in the order of farthest away to nearest. For example, your program should draw the sky first, then clouds, then the ground, then trees, then insects in the trees. Be creative.

Scene Gallery

The following example scene was drawn by a student's Python program and fulfills the requirements of this assignment.



If you would like ideas about what your program can draw, look at this [scene gallery](#). All the scenes in the gallery were drawn by programs written by former CSE 111 students.

Helpful Documentation

- The [prepare content for the previous lesson](#) explains how to call functions.
- The [prepare content for this lesson](#) explains how to write functions.
- The [documentation for the Draw 2-D library](#) includes an overview, example programs, a function reference, and a table of supported colors.
- The following videos walk through examples of using functions to draw with the Draw 2-D library.

[Drawing with Functions, part 1](#) (14 minutes)

[Drawing with Functions, part 2](#) (18 minutes)

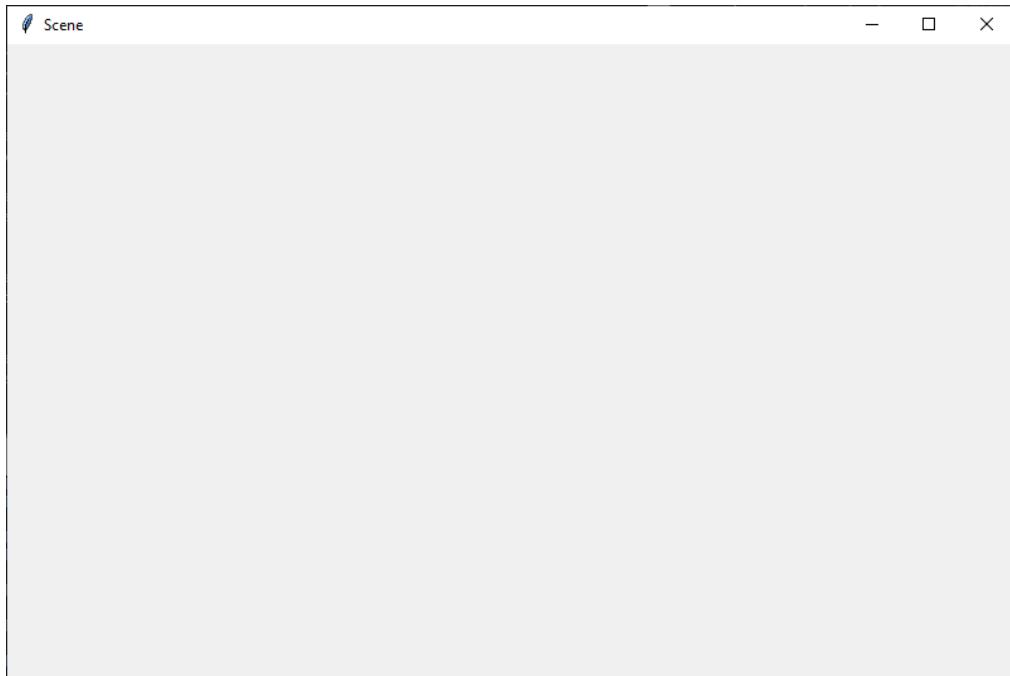
Steps

Do the following:

1. Download the [draw2d.py](#) file and save it in the same folder where you will save your program.
2. Using VS Code, create a new file and copy and paste the following code into your new file. This beginning code imports the functions from the Draw 2-D library and creates a window and a canvas that your program will draw into.

```
1 # Import the functions from the Draw 2-D library
2 # so that they can be used in this program.
3 from draw2d import \
4     start_drawing, draw_line, draw_oval, draw_arc, \
5     draw_rectangle, draw_polygon, draw_text, finish_drawing
6
7
8 def main():
9     # Width and height of the scene in pixels
10    scene_width = 800
11    scene_height = 500
12
13    # Call the start_drawing function in the draw2d.py
14    # library which will open a window and create a canvas.
15    canvas = start_drawing("Scene", scene_width, scene_height)
16
17    # Call your drawing functions such
18    # as draw_sky and draw_ground here.
19
20
21
22    # Call the finish_drawing function
23    # in the draw2d.py library.
24    finish_drawing(canvas)
25
26
27    # Define your functions such as
28    # draw_sky and draw_ground here.
29
30
31
32    # Call the main function so that
33    # this program will start executing.
34 main()
```

3. Save your new file as `scene.py` and run your new program. When you run it, it should open a window that contains an empty canvas as shown in the following figure.



4. Read the [documentation for the Draw 2-D library](#).
5. Starting at [line.31](#) of your new program, write your functions to draw the sky, ground, and clouds.
6. Starting at [line.21](#), write calls to your drawing functions.

Intent

The intent of this assignment is not to train you to become a world-famous cloud artist. Instead, the intent is to teach you how to apply the following principles to a real programming problem:

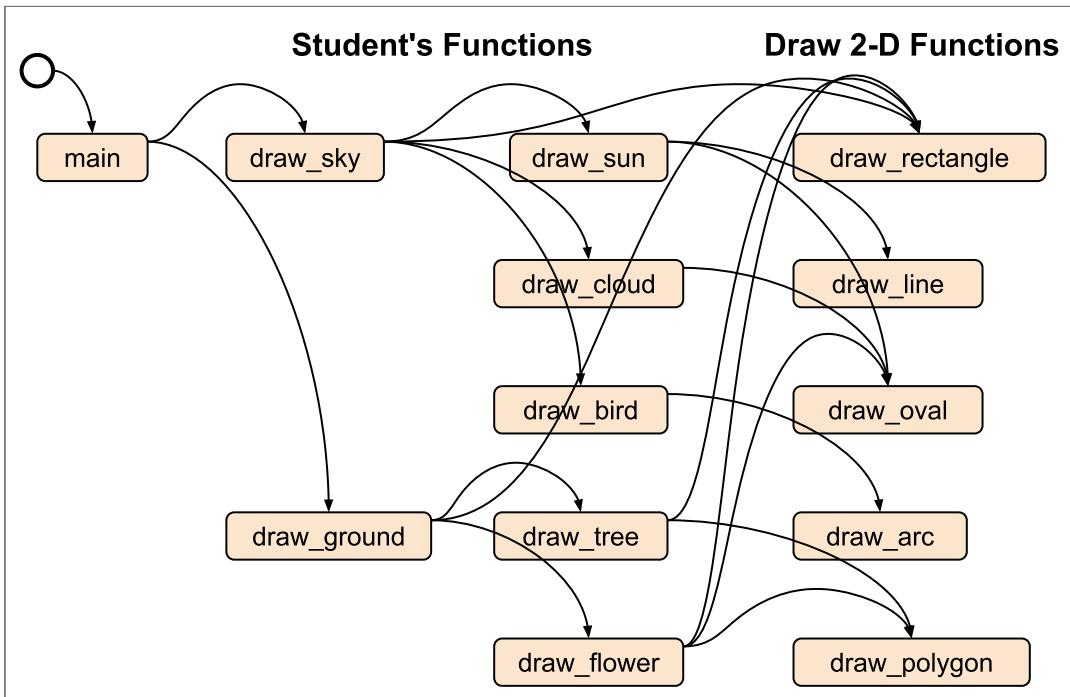
- Dividing a large program into functions
- Writing a function
- Deciding what needs to be a parameter in a function definition
- Calling a function with different argument values to produce different results
- Understanding how to separate the parts of a function that should stay the same and the parts that depend on parameters, in order to make the function as reusable as possible.

Your program must be divided into functions such as `draw_sky`, `draw_cloud`, `draw_ground`, `draw_bird`, `draw_grass_stem`, or `draw_insect`. The headers for the `draw_sky`, `draw_ground`, and `draw_pine_tree` functions in the [example program](#) of the

Draw 2-D documentation are good examples for the headers of all your draw_* functions. Below are a few other examples of headers for functions that you might want to include in your program. However, these are simply examples, and you can change them in your program or even exclude them from your program.

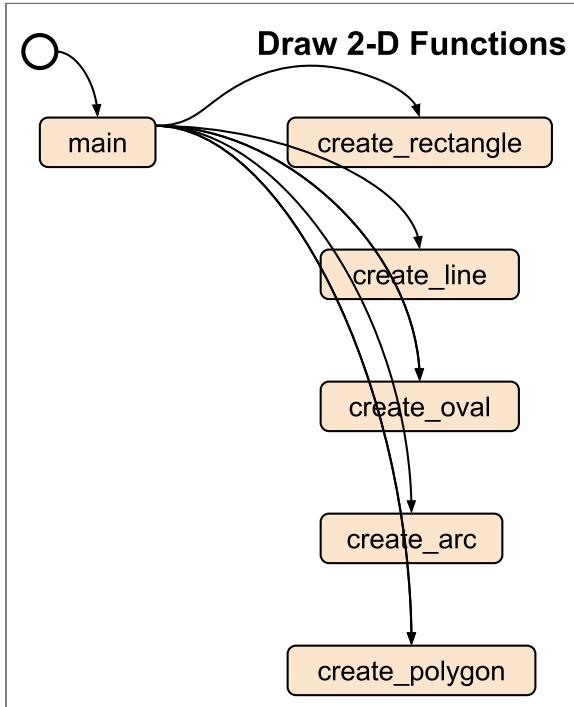
```
def draw_house(canvas, house_left, house_bottom):  
    pass  
  
def draw_bird(canvas, bird_center, bird_top):  
    pass  
  
def draw_pebble(canvas, pebble_left, pebble_top, pebble_radius):  
    pass  
  
def draw_picket(canvas, picket_left, picket_bottom):  
    pass  
  
def draw_grass_stem(canvas, stem_left, stem_top, stem_height):  
    pass
```

The following call graph shows the functions for a well-designed scene drawing program. From the call graph we see that this program is divided into small functions (draw_sky, draw_ground, draw_sun, draw_cloud, etc.). We see that the main function calls the draw_sky and draw_ground functions. The draw_sky function calls the draw_sun, draw_cloud, and draw_bird functions, and so on. This is a good design because each function is relatively small and performs just one task. Also, because the draw_flower function has parameters (not shown in the call graph), a programmer can write multiple calls to the draw_flower function, which will easily draw multiple flowers in different locations in his scene. The draw_cloud, draw_bird, and draw_tree functions also have parameters, and a programmer can write multiple calls to them, which will easily draw multiple clouds, birds, and trees in different locations. This is a good design.



The call graph for a well-designed program. Because the program is divided into small functions that each perform a single task, the program is well-designed.

The next call graph shows the functions for a different scene drawing program. Notice that this program isn't separated into small functions. Instead, all the code to draw an outdoor scene is inside the `main` function. The scene that this program draws may look very nice and have just as many objects as a well-designed program. However, the `main` function in this second program will be very long, complex, and confusing. This is a poor design.



The call graph for a poorly designed program. Because the program is not divided into small functions, the `main` function calls the Draw 2-D functions to draw all the objects in the scene. The `main` function will be very long and confusing.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and verify that it correctly opens a window and draws within that window an outdoor scene that contains at least the sky, clouds, and the ground.

Submission

On or before the due date, return to I-Learn and report your progress on this milestone.

Draw 2-D Library

Overview

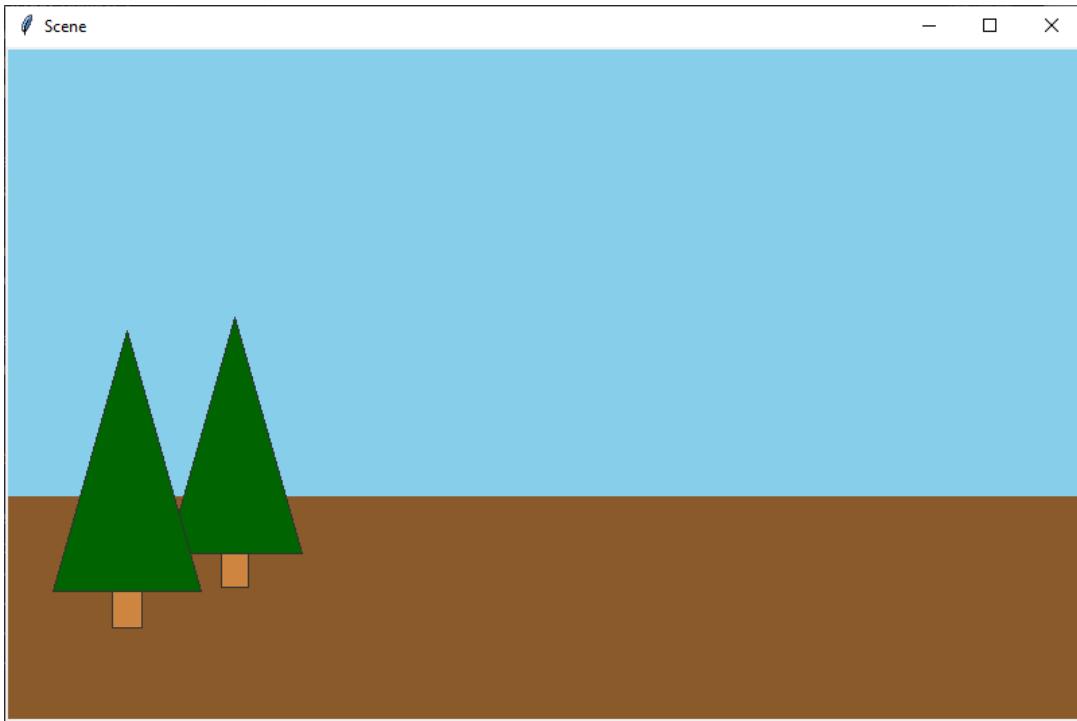
The Draw 2-D library contains functions that draw shapes on a canvas in a computer window. To use the Draw 2-D library, your program must do the following:

1. Import the library's functions
2. Call the `start_drawing` function once. The `start_drawing` function returns a canvas, that your program must pass to each of the `draw_*` functions.
3. Call the `draw_line`, `draw_oval`, `draw_arc`, `draw_rectangle`, `draw_polygon`, and `draw_text` functions as many times as needed. The `draw_*` functions accept Euclidean coordinates that determine where a shape will be drawn on the canvas. The origin of the canvas (0, 0) is in the lower left corner of the canvas.
4. Call the `finish_drawing` function once.

Examples

Outdoor Scene Example

Example program 1 contains all four of the steps needed to use the Draw 2-D library and draws an outdoor scene with two pine trees as shown in this image.



The computer draws the two pine trees because of the following:

1. The code at [line 90](#) calls the `main` function.
2. At [line 20](#), the `main` function calls the `draw_ground` function.
3. At [lines 42 and 49](#), the `draw_ground` function calls the `draw_pine_tree` function.

```
1 # Example 1
2
3 # Import the functions from the Draw 2-D library
4 # so that they can be used in this program.
5 from draw2d import \
6     start_drawing, draw_line, draw_oval, draw_arc, \
7     draw_rectangle, draw_polygon, draw_text, finish_drawing
8
9
10 def main():
11     scene_width = 800
12     scene_height = 500
13
14     # Call the start_drawing function in the draw2d.py
15     # library which will open a window and create a canvas.
16     canvas = start_drawing("Scene", scene_width, scene_height)
17
18     # Call the draw_sky and draw_ground functions in this file.
19     draw_sky(canvas, scene_width, scene_height)
20     draw_ground(canvas, scene_width, scene_height)
21
22     # Call the finish_drawing function
23     # in the draw2d.py library.
24     finish_drawing(canvas)
25
```

```

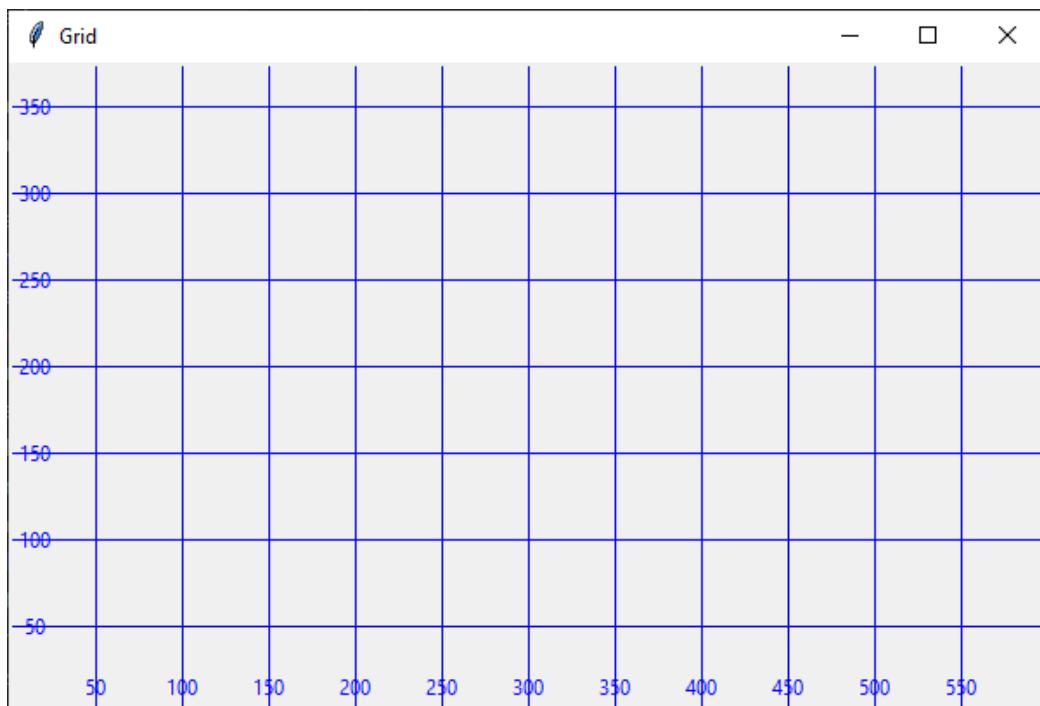
26
27 def draw_sky(canvas, scene_width, scene_height):
28     """Draw the sky and all the objects in the sky."""
29     draw_rectangle(canvas, 0, scene_height / 3,
30                    scene_width, scene_height, width=0, fill="sky blue")
31
32
33 def draw_ground(canvas, scene_width, scene_height):
34     """Draw the ground and all the objects on the ground."""
35     draw_rectangle(canvas, 0, 0,
36                    scene_width, scene_height / 3, width=0, fill="tan4")
37
38 # Draw a pine tree.
39 tree_center_x = 170
40 tree_bottom = 100
41 tree_height = 200
42 draw_pine_tree(canvas, tree_center_x,
43                 tree_bottom, tree_height)
44
45 # Draw another pine tree.
46 tree_center_x = 90
47 tree_bottom = 70
48 tree_height = 220
49 draw_pine_tree(canvas, tree_center_x,
50                 tree_bottom, tree_height)
51
52
53 def draw_pine_tree(canvas, center_x, bottom, height):
54     """Draw a single pine tree.
55     Parameters
56         canvas: The canvas where this function
57             will draw a pine tree.
58         center_x, bottom: The x and y location in pixels where
59             this function will draw the bottom of a pine tree.
60         height: The height in pixels of the pine tree that
61             this function will draw.
62     Return: nothing
63     """
64     trunk_width = height / 10
65     trunk_height = height / 8
66     trunk_left = center_x - trunk_width / 2
67     trunk_right = center_x + trunk_width / 2
68     trunk_top = bottom + trunk_height
69
70     # Draw the trunk of the pine tree.
71     draw_rectangle(canvas,
72                   trunk_left, trunk_top, trunk_right, bottom,
73                   outline="gray20", width=1, fill="tan3")
74
75     skirt_width = height / 2
76     skirt_height = height - trunk_height
77     skirt_left = center_x - skirt_width / 2
78     skirt_right = center_x + skirt_width / 2
79     skirt_top = bottom + height
80
81     # Draw the crown (also called skirt) of the pine tree.
82     draw_polygon(canvas, center_x, skirt_top,
83                  skirt_right, trunk_top,
84                  skirt_left, trunk_top,
85                  outline="gray20", width=1, fill="dark green")
86

```

```
87  
88 # Call the main function so that  
89 # this program will start executing.  
90 main()
```

Coordinate Grid

Example program 2 draws a grid that can help a programmer determine the correct coordinates for an object in a scene.



In example 2, the `for` loop at lines 24–26 causes the computer to draw the vertical lines in the grid. The `for` loop at lines 30–32 causes the computer to draw the vertical lines in the grid.

```
1 # Example 2  
2  
3 from draw2d import \  
4     start_drawing, draw_line, draw_text, finish_drawing  
5  
6 def main():  
7     scene_width = 600  
8     scene_height = 375  
9  
10    # Call the start_drawing function in the draw2d.py  
11    # library which will open a window and create a canvas.  
12    canvas = start_drawing("Grid", scene_width, scene_height)  
13  
14    draw_grid(canvas, scene_width, scene_height, 50)
```

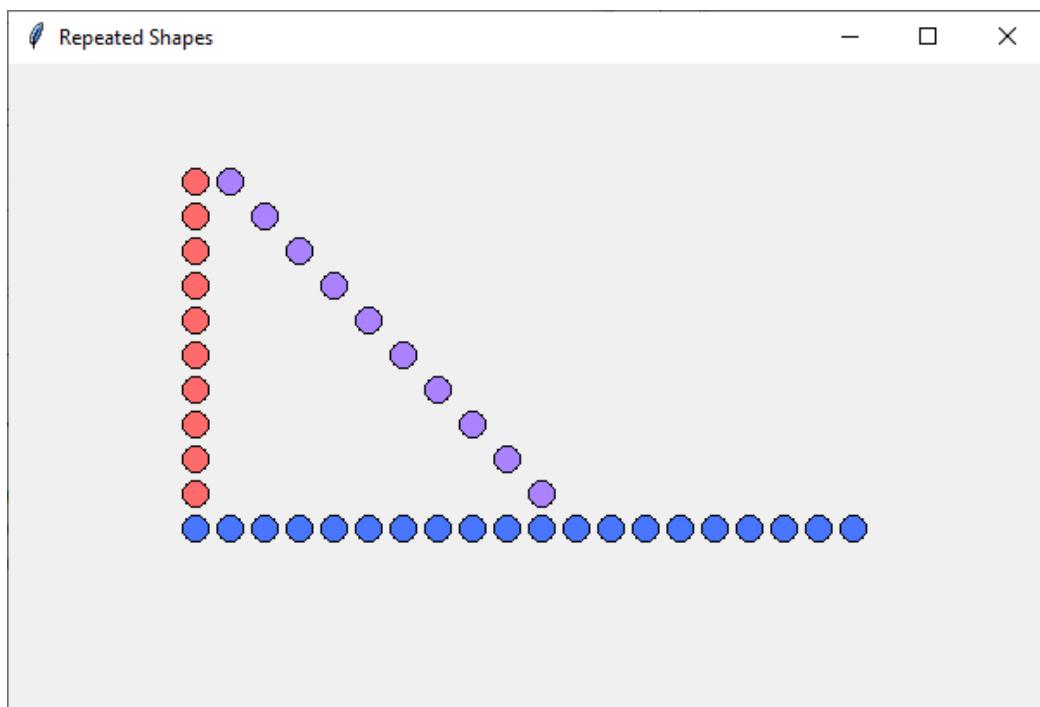
```

15
16     # Call the finish_drawing function
17     # in the draw2d.py library.
18     finish_drawing(canvas)
19
20
21 def draw_grid(canvas, width, height, interval, color="blue"):
22     # Draw a vertical line at every x interval.
23     label_y = 15
24     for x in range(interval, width, interval):
25         draw_line(canvas, x, 0, x, height, fill=color)
26         draw_text(canvas, x, label_y, f"{x}", fill=color)
27
28     # Draw a horizontal line at every y interval.
29     label_x = 15
30     for y in range(interval, height, interval):
31         draw_line(canvas, 0, y, width, y, fill=color)
32         draw_text(canvas, label_x, y, f"{y}", fill=color)
33
34
35 # Call the main function so that
36 # this program will start executing.
37 main()

```

Row, Column, and Diagonal of Shapes

The program in example 3 uses three for loops and the draw_oval function to draw circles in a row, a column, and a diagonal.

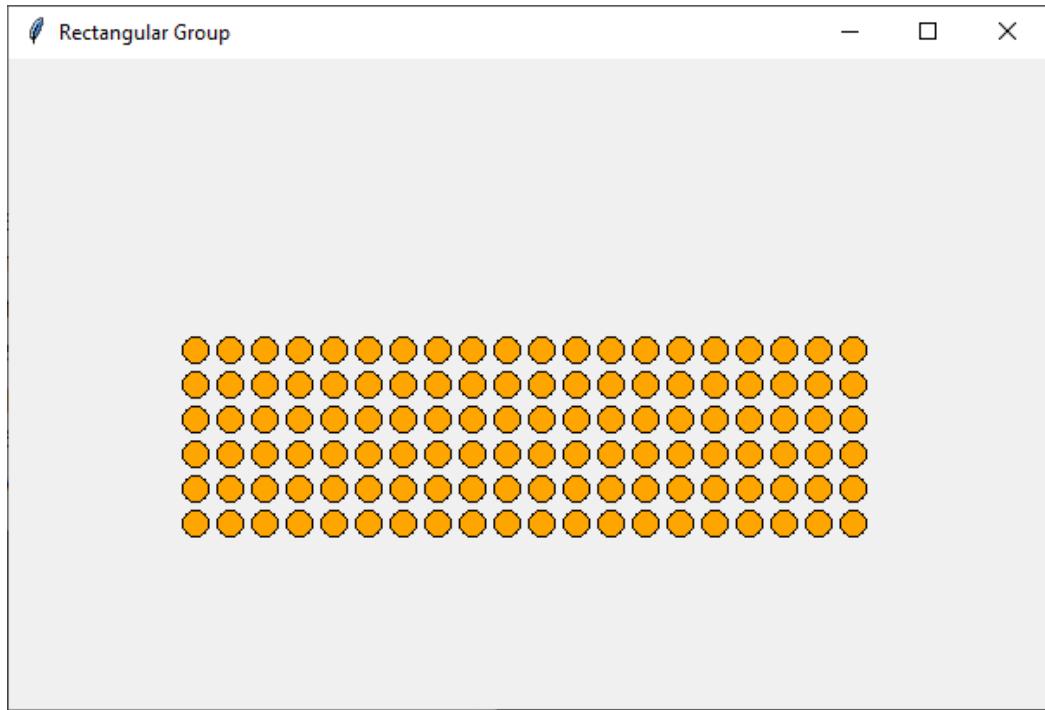


Comparing the three for loops at lines 17–20, 25–28, and 33–37 will help you understand how you can use a for loop to draw many shapes. Of course, your program can call any draw function within a for loop and draw any shape you like.

```
1 # Example 3
2
3 from draw2d import start_drawing, draw_oval, finish_drawing
4
5 def main():
6     # Call the start_drawing function in the draw2d.py
7     # library which will open a window and create a canvas.
8     canvas = start_drawing("Repeated Shapes", 600, 375)
9
10    diameter = 15
11    space = 5
12    interval = diameter + space
13
14    # Draw a row of 20 circles.
15    x = 100
16    y = 100
17    for i in range(20):
18        draw_oval(canvas, x, y, x + diameter, y + diameter,
19                  fill="royalBlue1")
20        x += interval
21
22    # Draw a column of 10 circles.
23    x = 100
24    y = 120
25    for i in range(10):
26        draw_oval(canvas, x, y, x + diameter, y + diameter,
27                  fill="indianRed1")
28        y += interval
29
30    # Draw a diagonal of 10 circles.
31    x = 120
32    y = 300
33    for i in range(10):
34        draw_oval(canvas, x, y, x + diameter, y + diameter,
35                  fill="mediumPurple1")
36        x += interval
37        y -= interval
38
39    # Call the finish_drawing function
40    # in the draw2d.py library.
41    finish_drawing(canvas)
42
43
44 # Call the main function so that
45 # this program will start executing.
46 main()
```

Rectangular Group of Shapes

The program in example 4 uses two for loops and the draw_oval function to draw a rectangular group of circles.



The first `for` loop starts at line 16. The second `for` loop is nested inside the first at lines 18–21. The computer executes the first `for` loop once for each row in the group. The computer executes the second `for` loop once for each cell in the group. Because the loops are nested, the computer executes the statements inside the second loop at lines 19–21 120 times (6×20). In other words, the program in example 4 will draw six rows with 20 circles in each row, which makes 120 circles.

```
1 # Example 4
2
3 from draw2d import start_drawing, draw_oval, finish_drawing
4
5 def main():
6     # Call the start_drawing function in the draw2d.py
7     # library which will open a window and create a canvas.
8     canvas = start_drawing("Rectangular Group", 600, 375)
9
10    diameter = 15
11    space = 5
12    interval = diameter + space
13
14    # Draw a rectangular series of circles.
15    y = 100
16    for row in range(6):
17        x = 100
18        for cell in range(20):
19            draw_oval(canvas, x, y,
20                      x + diameter, y + diameter, fill="orange")
21            x += interval
22        y += interval
23
24    # Call the finish_drawing function
```

```
25     # in the draw2d.py library.
26     finish_drawing(canvas)
27
28
29 # Call the main function so that
30 # this program will start executing.
31 main()
```

Randomly Located Shapes

The program in example 5 draws a row of circles but some of the circles are missing because the computer randomly chose not to draw some of the circles. It also draws 100 circles, each with a random diameter at a randomly chosen location.



In example 5 at line 24, the call to the `random.randint` function causes the computer to calculate a random number between 1 and 5 inclusive. The if statement at line 25 allows the computer to draw a circle if the random number is greater than 1. In other words, if the random number is 1, the computer will not draw a circle, which means about 20% ($1 / 5$) of the circles in the row will not be drawn.

In example 5, the statements at lines 36–41, cause the computer to draw 100 circles, each with a random location and diameter. At lines 37–39, the computer calculates a random location (x and y) and a random diameter. Then at lines 40–41 the computer uses those random numbers to draw a circle.

```
1 # Example 5
```

```

2
3 from draw2d import start_drawing, draw_oval, finish_drawing
4 import random
5
6 def main():
7     scene_width = 600
8     scene_height = 375
9
10    # Call the start_drawing function in the draw2d.py
11    # library which will open a window and create a canvas.
12    canvas = start_drawing("Random Locations",
13                           scene_width, scene_height)
14
15    diameter = 15
16    space = 5
17    interval = diameter + space
18
19    # Draw a row of circles with
20    # some of the circles missing.
21    x = 100
22    y = 300
23    for i in range(20):
24        number = random.randint(1, 5)
25        if number > 1:
26            draw_oval(canvas, x, y,
27                      x + diameter, y + diameter, fill="yellow2")
28            x += interval
29
30    half_height = round(scene_height / 2)
31    min_diam = 15
32    max_diam = 30
33
34    # Draw 100 circles, each with
35    # a random location and diameter.
36    for i in range(100):
37        x = random.randint(0, scene_width - max_diam)
38        y = random.randint(0, half_height)
39        diameter = random.randint(min_diam, max_diam)
40        draw_oval(canvas, x, y, x + diameter, y + diameter,
41                  fill="mediumOrchid1")
42
43    # Call the finish_drawing function
44    # in the draw2d.py library.
45    finish_drawing(canvas)
46
47
48 # Call the main function so that
49 # this program will start executing.
50 main()

```

Function Reference

The following functions are contained in the Draw 2-D library.

start_drawing(title, width, height)

Create a window with a canvas where a program can draw 2-dimensional shapes.

Parameters

- **title**: the title that will appear in the window's title bar
- **width**: the width in pixels of the canvas
- **height**: the height in pixels of the canvas

Return

- the new canvas

Example

```
canvas = start_drawing("Scene", 800, 500)
```

```
draw_line(canvas, x0, y0, x1, y1, ... xn, yn, width=1, fill="black")
```

Draw a line that goes through the series of points $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$

Parameters

- **canvas**: the canvas returned from the `start_drawing` function
- **x0, y0, x1, y1, ... xn, yn**: the coordinates for the segments of a line that this function will draw
- **width**: the line's width. The default is 1 pixel.
- **fill**: the line's color. The default is black.

Return

- nothing

Example

```
draw_line(canvas, 0, 5, 100, 50, width=10, fill="blue")
```

```
draw_oval(canvas, x0, y0, x1, y1, width=1, outline="black", fill="")
```

Draw an oval (ellipse) inside the bounding box defined by the coordinates $(x_0, y_0), (x_1, y_1)$

Parameters

- **canvas**: the canvas returned from the `start_drawing` function
- **x0, y0, x1, y1**: the coordinates of a bounding box for the oval that this function will draw

- **width**: the width of the oval's outline. The default is 1 pixel. A width of 0 will draw an oval without an outline.
- **outline**: the color of the oval's outline. The default is black.
- **fill**: the color of the oval's interior. The default is "" which means transparent.

Return

- nothing

Example

```
draw_oval(canvas, 100, 100, 300, 200, fill="pink")
```

```
draw_arc(canvas, x0, y0, x1, y1, start=0, extent=90,
         width=1, outline="black", fill="")
```

Draw a wedge shaped slice taken from an oval (ellipse) defined by the bounding box with coordinates (x_0, y_0) , (x_1, y_1) .

Parameters

- **canvas**: the canvas returned from the `start_drawing` function
- **x0, y0, x1, y1**: the bounding box for the oval from which this function will draw a wedge
- **width**: the width of the oval's outline; default is 1 pixel
- **outline**: the color of the oval's outline; default is black
- **fill**: the color of the oval's interior; default is "" which means transparent

Return

- nothing

Example

```
draw_arc(canvas, 100, 100, 300, 200, start=180, extent=270)
```

```
draw_rectangle(canvas, x0, y0, x1, y1,
               width=1, outline="black", fill="")
```

Draw a rectangle with two of its corners at (x_0, y_0) , (x_1, y_1) .

Parameters

- **canvas**: the canvas returned from the `start_drawing` function
- **x0, y0, x1, y1**: the bounding box for the rectangle that this function will draw

- **width**: the width of the rectangle's outline; default is 1 pixel
- **outline**: the color of the rectangle's outline; default is black
- **fill**: the color of the rectangle's interior; default is "" which means transparent

Return

- nothing

Example

```
draw_rectangle(canvas, 100, 100, 300, 200, width=5)
```

```
draw_polygon(canvas, x0, y0, x1, y1, x2, y2, ... xn, yn,  
width=1, outline="black", fill="")
```

Draw a polygon with vertices $(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)$. The polygon is always a closed polygon with the same quantity of segments as vertices. In other words, the segments are defined as follows:
 $(x_0, y_0) -> (x_1, y_1) -> \dots -> (x_n, y_n) -> (x_0, y_0)$

Parameters

- **canvas**: the canvas returned from the `start_drawing` function
- **x0, y0, x1, y1, x2, y2, ... xn, yn**: the coordinates for the vertices of the polygon that this function will draw
- **width**: the width of the polygon's outline; default is 1 pixel
- **outline**: the color of the polygon's outline; default is black
- **fill**: the color of the polygon's interior; default is "" which means transparent

Return

- nothing

Example

```
draw_polygon(canvas,  
            100, 100, 200, 100, 200, 150, fill="green")
```

```
draw_text(canvas, center_x, center_y, text, fill="black")
```

Draw text with the center of the text at $(center_x, center_y)$.

Parameters

- **canvas**: the canvas returned from the `start_drawing` function

- **center_x, center_y**: the coordinates for the center point where this function will draw text
- **text**: the text to be drawn. To force a line break, include a newline character ("\n").

Return

- nothing

Example

```
draw_text(canvas, 250, 200, "Olá", fill="purple")
```

```
finish_drawing(canvas)
```

Call all functions that are necessary to display the drawing on the computer's monitor.

Parameters

- **canvas**: the canvas returned from the start_drawing function

Return

- nothing

Example

```
finish_drawing(canvas)
```

Colors

The Draw 2-D library supports the following named colors.

lightCoral	chocolate1	orange2	chartreuse	turquoise4	dodgerBlue1	violet	snow1	gray51
indianRed1	chocolate2	orange3	chartreuse2	darkSlateGray1	dodgerBlue2	magenta	snow2	gray50
indianRed2	chocolate	orange4	chartreuse3	darkSlateGray2	dodgerBlue3	magenta2	snow3	gray49
indianRed	chocolate3	darkOrange	chartreuse4	darkSlateGray3	dodgerBlue4	magenta3	white	gray48
indianRed3	saddleBrown	darkOrange1	mintCream	darkSlateGray4	cornflowerBlue	magenta4	gray99	gray47
indianRed4	seashell	darkOrange2	honeydew1	darkSlateGray	royalBlue	orchid1	gray98	gray46
firebrick1	seashell2	darkOrange3	honeydew2	lightCyan1	royalBlue1	orchid2	gray97	gray45
firebrick2	seashell3	darkOrange4	honeydew3	lightCyan2	royalBlue2	orchid	gray96	gray44
firebrick3	seashell4	paleGoldenrod	honeydew4	lightCyan3	royalBlue3	orchid3	gray95	gray43
firebrick	peachPuff	goldenrod1	seaGreen1	lightCyan4	royalBlue4	orchid4	gray94	gray42
firebrick4	peachPuff2	goldenrod2	seaGreen2	cyan1	blue1	maroon1	gray93	gray41
red1	peachPuff3	goldenrod	seaGreen3	cyan2	blue2	maroon2	gray92	gray40
red2	peachPuff4	goldenrod3	mediumSeaGreen	cyan3	blue3	maroon3	gray91	gray39
red3	tan	goldenrod4	seaGreen	cyan4	blue4	maroon	gray90	gray38
red4	tan1	darkGoldenrod1	darkSeaGreen1	cadetBlue1	navyBlue	maroon4	gray89	gray37
rosyBrown1	sandyBrown	darkGoldenrod2	darkSeaGreen2	cadetBlue2	midnightBlue	deepPink1	gray88	gray36
rosyBrown2	tan2	darkGoldenrod3	darkSeaGreen3	cadetBlue3	lavender	deepPink2	gray87	gray35
rosyBrown3	tan3	darkGoldenrod	darkSeaGreen	cadetBlue	slateBlue1	deepPink3	gray86	gray34
rosyBrown	tan4	darkGoldenrod4	darkSeaGreen4	cadetBlue4	slateBlue2	deepPink4	gray85	gray33
rosyBrown4	floralWhite	gold1	paleGreen1	ghostWhite	slateBlue3	hotPink1	gray84	gray32
brown1	linen	gold2	paleGreen	aliceBlue	slateBlue4	hotPink	gray83	gray31
brown2	oldLace	gold3	paleGreen2	powderBlue	mediumPurple1	hotPink2	gray82	gray30
brown3	antiqueWhite	gold4	paleGreen3	lightBlue1	mediumPurple2	hotPink3	gray81	gray29
brown	antiqueWhite1	lemonChiffon1	paleGreen4	lightBlue	mediumPurple	hotPink4	gray80	gray28
brown4	antiqueWhite2	lemonChiffon2	forestGreen	lightBlue2	mediumPurple3	paleVioletRed1	gray79	gray27
tomato1	antiqueWhite3	lemonChiffon3	green1	lightBlue3	mediumPurple4	paleVioletRed	gray78	gray26
tomato2	antiqueWhite4	lemonChiffon4	green2	lightBlue4	purple1	paleVioletRed2	gray77	gray25
tomato3	cornsilk	khaki1	green3	deepSkyBlue	purple2	paleVioletRed3	gray76	gray24
tomato4	cornsilk2	khaki	green4	deepSkyBlue2	purple3	paleVioletRed4	gray75	gray23
coral	cornsilk3	khaki2	green	deepSkyBlue3	purple4	violetRed1	gray74	gray22
coral1	cornsilk4	khaki3	darkGreen	deepSkyBlue4	purple	violetRed2	gray73	gray21
coral2	bisque1	darkKhaki	springGreen1	lightSkyBlue1	blueViolet	violetRed	gray72	gray20
coral3	bisque2	khaki4	springGreen2	lightSkyBlue2	darkViolet	violetRed3	gray71	gray19
coral4	bisque3	ivory1	springGreen3	lightSkyBlue3	mediumOrchid1	violetRed4	gray70	gray18
salmon	bisque4	ivory2	springGreen4	lightSkyBlue4	mediumOrchid2	lavenderBlush1	gray69	gray17
salmon1	burlywood1	ivory3	aquamarine1	skyBlue	mediumOrchid	lavenderBlush2	gray68	gray16
salmon2	burlywood2	ivory4	aquamarine2	skyBlue1	mediumOrchid3	lavenderBlush3	gray67	gray15
salmon3	burlywood3	beige	aquamarine3	lightSkyBlue	mediumOrchid4	lavenderBlush4	gray66	gray14
salmon4	burlywood4	lightYellow1	aquamarine4	skyBlue2	darkOrchid1	pink	gray65	gray13
lightSalmon1	burlywood	lightYellow2	azure1	skyBlue3	darkOrchid2	pink1	gray64	gray12
lightSalmon2	blanchedAlmond	lightYellow3	azure2	skyBlue4	darkOrchid3	pink2	gray63	gray11
darkSalmon	papayaWhip	yellow1	azure3	lightSteelBlue	darkOrchid4	pink3	gray62	gray10
lightSalmon3	moccasin	yellow2	azure4	steelBlue1	thistle1	pink4	gray61	gray9
lightSalmon4	navajoWhite	yellow3	paleTurquoise1	steelBlue2	thistle2	lightPink	gray60	gray8
orangeRed1	navajoWhite2	yellow4	paleTurquoise2	steelBlue3	thistle	lightPink1	gray59	gray7
orangeRed2	navajoWhite3	oliveDrab1	paleTurquoise3	steelBlue	thistle3	lightPink2	gray58	gray6
orangeRed3	navajoWhite4	oliveDrab2	paleTurquoise4	steelBlue4	thistle4	lightPink3	gray57	gray5
orangeRed4	wheat	oliveDrab3	lightSeaGreen	slateGray1	plum1	lightPink4	gray56	gray4
sienna1	wheat1	oliveDrab4	turquoise	slateGray2	plum2	mistyRose1	gray55	gray3
sienna2	wheat2	darkOliveGreen	turquoise1	slateGray3	plum	mistyRose2	gray54	gray2
sienna3	wheat3	greenYellow	turquoise2	lightSlateGray	plum3	mistyRose3	gray53	gray1

sienna	wheat4	lawnGreen	darkTurquoise	slateGray	plum4	mistyRose4	gray52	black
sienna4	orange	limeGreen	turquoise3	slateGray4				

04 Prepare: Function Details

During this lesson, you will learn additional details about writing and calling functions. These details include variable scope, default parameter values, and optional arguments and will help you understand functions better and write them more effectively.

Variable Scope

The **scope** of a variable determines how long that variable exists and where it can be used. Within a Python program, there are two categories of scope: local and global. A variable has **local scope** when it is defined (assigned a value) inside a function. A variable has **global scope** when it is defined outside of all functions. Here is a small Python program that has two variables: `g` and `x`. `g` is defined outside of all functions and therefore has global scope. `x` is defined inside the `main` function and therefore has local scope.

```
# g is a global variable because it
# is defined outside of all functions.
g = 25

def main():
    # x is a local variable because
    # it is defined inside a function.
    x = 1
```

As shown in the following table, a local variable (a variable with local scope) is defined inside a function, exists for as long as its containing function is executing, and can be used within its containing function but nowhere else. A global variable (a variable with global scope) is defined outside all functions, exists for as long as its containing Python program is executing, and can be used within all functions in its containing Python program.

Python Variable Scope		
	Local	Global
Where to Define	Inside a function	Outside all functions
Owner	The function where the variable is defined	The Python file where the variable is defined
Lifetime	Only as long as its containing function is executing	As long as its containing program is executing
Where Usable	Only inside the function where it is defined	In all functions of the Python program

The following Python code example contains parameters and variables. Parameters have local scope because they are defined within a function, specifically within a function's header and exist for as long as their containing function is executing. The variable

nShapes is global because it is defined outside of all functions. Because it is a global variable, the code in the body of all functions may use the variable *nShapes*. Within the *square_area* function, the parameter named *length* and the variable named *area* both have local scope. Within the *rectangle_area* function, the parameters named *width* and *length* and the variable named *area* have local scope.

```
nShapes = 0

def square_area(length):
    area = length * length
    return area

def rectangle_area(width, length):
    area = width * length
    return area
```

Because local variables are visible only within the function where they are defined, a programmer can define two variables with the same name as long as he defines them in different functions. In the previous example, both of the *square_area* and *rectangle_area* functions contain a parameter named *length* and a variable named *area*. All four of these variables are entirely separate and do not conflict with each other in any way because the scope of each variable is local to the function where it is defined.

Common Mistake

A common mistake that many programmers make is to assume that a local variable can be used inside other functions. For example, the Python program in example 3 includes two functions named *main* and *circle_area*. Line 6 in *main* defines a variable named *radius*. Some programmers assume that the variable *radius* that is defined in *main* (and is therefore local to *main* only) can be used in the *circle_area* function. However, local variables from one function cannot be used inside another function. The local variables from *main* cannot be used inside *circle_area*.

```
1 # Example 3
2
3 import math
4
5 def main():
6     radius = float(input("Enter the radius of a circle: "))
7     area = circle_area()
8     print(f"area: {area:.1f}")
9
10 def circle_area():
11     # Mistake! There is no variable named radius
12     # defined inside this function, so the variable
13     # radius cannot be used in this function.
14     area = math.pi * radius * radius
15     return area
16
17 main()
```

```

> python example_3.py
Enter the radius of a circle: 4.17
Traceback (most recent call last):
  File "c:\Users\cse111\example_3.py", line 17, in <module>
    main()
  File "c:\Users\cse111\example_3.py", line 7, in main
    area = circle_area()
  File "c:\Users\cse111\example_3.py", line 14, in circle_area
    area = math.pi * radius * radius
NameError: name 'radius' is not defined

```

The correct way to fix the mistake in example 3 is to add a parameter to the `circle_area` function as shown at [line 10](#) and pass the radius from the `main` function to the `circle_area` function as shown at [line 7](#) in example 4.

```

1 # Example 4
2
3 import math
4
5 def main():
6     radius = float(input("Enter the radius of a circle: "))
7     area = circle_area(radius)
8     print(f"area: {area:.1f}")
9
10 def circle_area(radius):
11     area = math.pi * radius * radius
12     return area
13
14 main()

```

```

> python example_4.py
Enter the radius of a circle: 4.17
area: 54.6

```

Default Parameter Values and Optional Arguments

Python allows function parameters to have default values. If a parameter has a default value, then its corresponding argument is optional. If a function is called without an argument, the corresponding parameter gets its default value.

Consider the program in example 5. Notice at [line 19](#) in the header for the `arc_length` function, that the parameter `radius` does not have a default value but the parameter `degrees` has a default value of 360. This means that when a programmer writes code to call the `arc_length` function, the programmer must pass a value for `radius` but is not required to pass a value for `degrees`. At [line 8](#), the programmer wrote code to call the `arc_length` function and passed 4.7 for the `radius` parameter but did not pass a value for the `degrees`, so during that call to `arc_length`, the value of `degrees` will be the default value from [line 19](#), which is 360. At [line 13](#), the programmer wrote code to call the `arc_length` function again and passed two arguments: 4.7 and 270, so during that call to `arc_length`, the value of `degrees` will be 270.

```

1 # Example 5
2
3 import math
4
5 def main():
6     # Call the arc_length function with only one argument
7     # even though the arc_length function has two parameters.
8     len1 = arc_length(4.7)
9     print(f"len1: {len1:.1f}")
10
11    # Call the arc_length function again but
12    # this time with two arguments.
13    len2 = arc_length(4.7, 270)
14    print(f"len2: {len2:.1f}")
15
16
17    # Define a function with two parameters. The
18    # second parameter has a default value of 360.
19    def arc_length(radius, degrees=360):
20        """Compute and return the length of an arc of a circle."""
21        circumference = 2 * math.pi * radius
22        length = circumference * degrees / 360
23        return length
24
25
26 main()

```

```

> python example_5.py
len1: 29.5
len2: 22.1

```

Function Design

What are the properties of a good function?

There are many things to consider when writing a function, and many authors have written about design concepts that make functions easier to understand and less error prone. In future courses at BYU-Idaho, you will study some of these design concepts. For CSE 111, the following list contains a few properties that you should incorporate into your functions.

- A good function is understandable by other programmers. One way to make a function understandable is to write a documentation string at the top of the function that describes the function, its parameters, and its return value. Another way to make a function understandable is to write comments in the body of the function as needed.
- A good function performs a single task that the programmer can describe, and the function's name matches its task.
- A good function is relatively short, perhaps fewer than 20 lines of code.

- A good function has as few decision points (`if` statements and loops) as possible. Too many decision points in a function make a function error prone and difficult to test.
- A good function is as reusable as possible. Functions that use parameters and return a result are more reusable than functions that get input from a user and print results to a terminal window.

As you read the sample code in CSE 111, observe how the sample functions fit these good properties, and as you write programs for CSE 111, do your best to write functions that have these good properties.

Summary

During this lesson, you are studying variable scope, default parameter values, and optional arguments. A variable that is defined (assigned a value) inside a function has local scope, and it can be used inside only the function where it is defined. Parameters always have local scope.

A parameter may have a default value like the parameter `degrees` in this function header:

```
def arc_length(radius, degrees=360):  
    :
```

When a function's parameter has a default value, you can write a call to that function without passing an argument for the parameter like this:

```
length = arc_length(3.5)
```

In other words, the argument for a parameter that has a default value is optional.

04 Checkpoint: Variable Scope

Purpose

Check your understanding of parameters, arguments, and local variable scope by fixing a program that won't run because the programmer tried to use variables that were defined in a different function.

Helpful Documentation

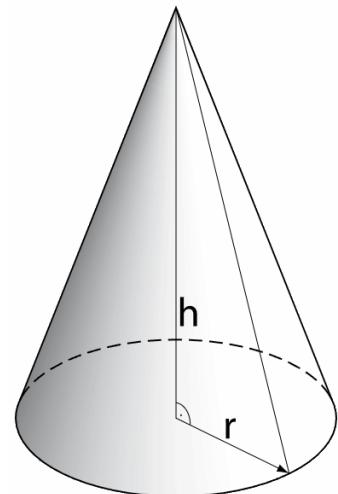
The prepare content for lesson 3 explains what a [parameter](#) is.

The prepare content for lesson 2 explains what an [argument](#) is.

The prepare content for this lesson contains an [example program](#) with a mistake similar to the mistake that is in the example program below.

Problem Statement

The following example Python program is supposed to ask the user for the radius and height of a right circular cone and then compute and print the volume of that cone. The code is almost correct, but it contains a common mistake. The programmer who wrote it assumed that the `cone_volume` function could use the `radius` and `height` variables that are defined in the `main` function. Of course, this assumption is incorrect because as the [prepare content](#) for this lesson explains, variables defined inside a function have local scope and can be used only inside their containing function.



A right circular cone with radius r and height h

```
1  """Compute and print the volume of a right circular cone."""
2
3  # Import the standard math module so that
4  # math.pi can be used in this program.
```

```

5 import math
6
7
8 def main():
9     # Call the cone_volume function to compute
10    # the volume of an example cone.
11    ex_radius = 2.8
12    ex_height = 3.2
13    ex_vol = cone_volume()
14
15    # Print several lines that describe this program.
16    print("This program computes the volume of a right")
17    print("circular cone. For example, if the radius of a")
18    print(f"cone is {ex_radius} and the height is {ex_height}")
19    print(f"then the volume is {ex_vol:.1f}")
20    print()
21
22    # Get the radius and height of the cone from the user.
23    radius = float(input("Please enter the radius of the cone: "))
24    height = float(input("Please enter the height of the cone: "))
25
26    # Call the cone_volume function to compute the volume
27    # for the radius and height that came from the user.
28    vol = cone_volume()
29
30    # Print the radius, height, and
31    # volume for the user to see.
32    print(f"Radius: {radius}")
33    print(f"Height: {height}")
34    print(f"Volume: {vol:.1f}")
35
36
37 def cone_volume():
38     """Compute and return the volume of a right circular cone."""
39     volume = math.pi * radius**2 * height / 3
40     return volume
41
42
43 # Start this program by
44 # calling the main function.
45 main()

```

Assignment

Do the following:

1. Using VS Code, open a new file. Copy and paste all of the example program above into the new file. Save the new file as `cone_volume.py`
2. Run your `cone_volume.py` program. What error message do you see in the terminal frame?
3. Fix the `cone_volume` function which begins on line 36 by adding two parameters to its header. What parameter names should you use?

Hint: the parameter names at [line 36](#) and the variable names at [line 38](#) must be the same.

4. Fix the call to the `cone_volume` function that is in `main` at [line 13](#) by adding two arguments. Because the `cone_volume` function has two parameters (you added the two parameters in the previous step), each call to the `cone_volume` function must have two arguments. What variable names should you use for the arguments?

Hint: the only variables that you can use must be defined inside `main` above line 13.

5. Fix the call to the `cone_volume` function that is in `main` at [line 27](#) by adding two arguments. What variable names should you use for the arguments?

Hint: the only variables that you can use must be defined inside `main` above line 27.

6. Save your program and run it again. Did it run correctly? If not, fix the mistakes until it runs correctly.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. After you finish the steps of this assignment, run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python cone_volume.py
This program computes the volume of a right circular cone.
For example, if the radius of a cone is 2.8 and
the height is 3.2, then the volume is 26.3

Please enter the radius of the cone: 5
Please enter the height of the cone: 8.2
Radius: 5.0
Height: 8.2
Volume: 214.7
```

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Ponder

The original program that you copied and pasted to start this assignment didn't work because the programmer tried to use the *radius* and *height* variables inside the `cone_volume` function. However, the *radius* and *height* variables are defined inside the `main` function and therefore have local scope and cannot be used outside of `main`. To fix the program, you added two parameters to the `cone_volume` function and then added two arguments to each call of the `cone_volume` function.

Why was local variable scope invented by computer scientists? How does local variable scope make a program easier to write and understand?

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson04/check_solution.py

```
1 """Compute and print the volume of a right circular cone."""
2
3 # Import the standard math module so that
4 # math.pi can be used in this program.
5 import math
6
7
8 def main():
9     # Call the cone_volume function to compute
10    # the volume of an example cone.
11    ex_radius = 2.8
12    ex_height = 3.2
13    ex_vol = cone_volume(ex_radius, ex_height)
14
15    # Print several lines that describe this program.
16    print("This program computes the volume of a right circular cone.")
17    print(f"For example, if the radius of a cone is {ex_radius} and")
18    print(f"the height is {ex_height}, then the volume is {ex_vol}")
19    print()
20
21    # Get the radius and height of the cone from the user.
22    radius = float(input("Please enter the radius of the cone: "))
23    height = float(input("Please enter the height of the cone: "))
24
25    # Call the cone_volume function to compute the volume
26    # for the radius and height that came from the user.
27    vol = cone_volume(radius, height)
28
29    # Print the radius, height, and
30    # volume for the user to see.
31    print(f"Radius: {radius}")
32    print(f"Height: {height}")
33    print(f"Volume: {vol}")
34
35
36 def cone_volume(radius, height):
37     """Compute and return the volume of a right circular cone."""
38     volume = math.pi * radius**2 * height / 3
39     return volume
40
41
42 # Start this program by
43 # calling the main function.
44 main()
```

04 Team Activity: Writing Functions

Instructions

Each lesson in CSE 111 contains a team activity that is designed to take about one hour to complete. You should prepare for all team activities by completing the preparation material and the individual checkpoint assignment before starting a team activity. The goal of the team activities is for students to work together and teach and learn from each other. As your team completes a team activity, instead of moving through it as quickly as you can, you should help everyone understand the concepts.

Face-to-Face Students

Face-to-face students will complete the team activities in their classroom during class time.

Online Students in a Semester Section

Online students in a semester section will arrange and participate in a one hour synchronous video meeting with your team for each team activity. As your team works through the assignment, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Online Students in a Block Section

Students in a block section complete two lessons each week. This means that online students in a block section will complete two team activities each week. For one of these two activities, your team should arrange and participate in a one hour synchronous video meeting. For the other team activity, your team may meet in a synchronous video meeting or your team may collaborate, ask and answer questions, and share code in Microsoft Teams.

As your team works through a team assignment in a synchronous video meeting, be certain that each student can see the code that is being typed. When your team completes the video meeting, share the final Python file with everyone on your team.

Purpose

Strengthen your understanding of user-defined functions, parameters, arguments, and local variable scope by writing a program that has three or more functions.

Problem Statement

In many countries, food is stored in steel cans (also known as tin cans) that are shaped like cylinders. There are many different sizes of steel cans. The **storage efficiency** of a can tells us how much a can stores versus how much steel is required to make the can. Some sizes of cans require a lot of steel to store a small amount of food. Other sizes of cans require less steel and store more food. A can size with a large storage efficiency is considered more friendly to the environment than a can size with a small storage efficiency.

The storage efficiency of a steel can is computed by dividing the volume of a can by its surface area.

$$\text{storage_efficiency} = \frac{\text{volume}}{\text{surface_area}}$$

In other words, the storage efficiency of a can is the space inside the can divided by the amount of steel required to make the can. The formulas for the volume and surface area of a cylinder are:

$$\text{volume} = \pi \text{ radius}^2 \text{ height}$$

$$\text{surface_area} = 2\pi \text{ radius} (\text{radius} + \text{height})$$

- π is the constant PI, the ratio of the circumference of a circle divided by its diameter (use `math.pi`)
- radius is the radius of the cylinder
- height is the height of the cylinder

Helpful Documentation

The prepare content for the previous lesson explains [how to write functions](#).

The prepare content for this lesson explains [local variable scope](#).

The Python [math module](#) contains mathematical constants and functions including [math.pi](#).

Assignment

Work as a team to write a Python program named `can_sizes.py` that computes and prints the storage efficiency for each of the following 12 steel can sizes. Then visually examine the output and answer this question, "Which can size has the highest storage efficiency?"

Name	Radius (centimeters)	Height (centimeters)	Cost per Can (U.S. dollars)
#1 Picnic	6.83	10.16	\$0.28
#1 Tall	7.78	11.91	\$0.43
#2	8.73	11.59	\$0.45
#2.5	10.32	11.91	\$0.61
#3 Cylinder	10.79	17.78	\$0.86
#5	13.02	14.29	\$0.83
#6Z	5.40	8.89	\$0.22
#8Z short	6.83	7.62	\$0.26
#10	15.72	17.78	\$1.53
#211	6.83	12.38	\$0.34
#300	7.62	11.27	\$0.38
#303	8.10	11.11	\$0.42

If you separate your program into functions, this problem will be much easier to solve than if you don't separate it into functions. You are free to write any functions that you choose in your program, but we *strongly* suggest that your program include at least these three functions:

- `main`
- `compute_volume`
- `compute_surface_area`

Core Requirements

1. Your program must compute the volume of all 12 can sizes.
2. Your program must compute the surface area of all 12 can sizes.
3. Your program must compute and print the storage efficiency of all 12 can sizes.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add another function named `compute_storage_efficiency` to your program. This function should call the `compute_volume` and `compute_surface_area` functions and then compute and return the storage efficiency of a steel can size. Replace code in the `main` function with a call to the `compute_storage_efficiency` function as appropriate. Did adding and calling the `compute_storage_efficiency` function reduce the number of lines of code in your program?
2. The table of can sizes that appears in the Assignment section above includes a column that contains the cost per can of each steel can size. Add another function to your program named `compute_cost_efficiency` that computes and returns the volume of a steel can divided by its cost. Write code to call the `compute_cost_efficiency` function and print the cost efficiency for each can size. Then visually examine the output and answer this question, "Which can size has the highest cost efficiency?"
3. If you remember how to use lists and a for loop from CSE 110, rewrite your `main` function so that it uses a list that contains the can size names and dimensions. Then write a loop that processes the values in the list.
4. Add `if` statements inside the loop to automatically determine which can size has the best storage efficiency and which can size has the best cost efficiency.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and verify that it prints the correct results, rounded and formatted as shown below.

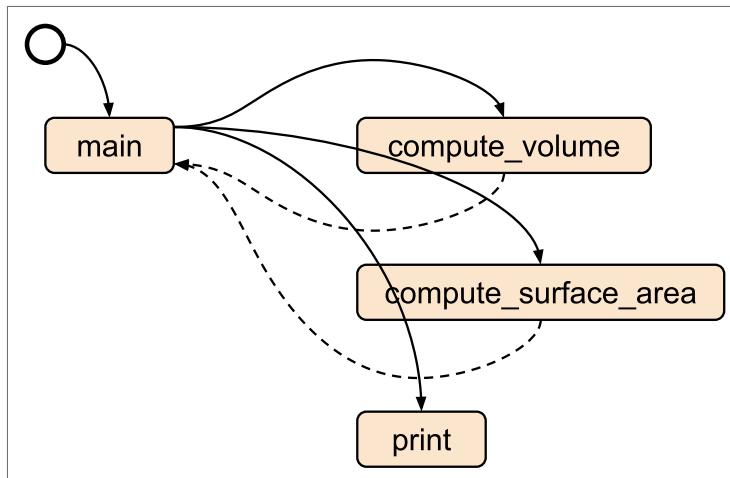
```
> python can_sizes.py
#1 Picnic 2.0
#1 Tall 2.4
#2 2.5
#2.5 2.8
#3 Cylinder 3.4
#5 3.4
#6Z 1.7
#8Z short 1.8
#10 4.2
#211 2.2
#300 2.3
#303 2.3
```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#) or the [stretch solution](#). Please ***do not look at the sample solution*** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample program by calling the `main` function. While executing the `main` function, the computer calls the `compute_volume`, `compute_surface_area`, and `print` functions.



Ponder

After you finish this assignment, congratulate yourself because you wrote a Python program with at least three user-defined functions. As you wrote your program, what did you learn about organizing a program into functions?

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson04/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 # Import the standard math module so that
4 # math.pi can be used in this program.
5 import math
6
7
8 def main():
9     name = "#1 Picnic"
10    radius = 6.83
11    height = 10.16
12    volume = compute_volume(radius, height)
13    surf_area = compute_surface_area(radius, height)
14    storage_efficiency = volume / surf_area
15    print(f"{name} {storage_efficiency:.1f}")
16
17    name = "#1 Tall"
18    radius = 7.78
19    height = 11.91
20    volume = compute_volume(radius, height)
21    surf_area = compute_surface_area(radius, height)
22    storage_efficiency = volume / surf_area
23    print(f"{name} {storage_efficiency:.1f}")
24
25    name = "#2"
26    radius = 8.73
27    height = 11.59
28    volume = compute_volume(radius, height)
29    surf_area = compute_surface_area(radius, height)
30    storage_efficiency = volume / surf_area
31    print(f"{name} {storage_efficiency:.1f}")
32
33    name = "#2.5"
34    radius = 10.32
35    height = 11.91
36    volume = compute_volume(radius, height)
37    surf_area = compute_surface_area(radius, height)
38    storage_efficiency = volume / surf_area
39    print(f"{name} {storage_efficiency:.1f}")
40
41    name = "#3 Cylinder"
42    radius = 10.79
43    height = 17.78
44    volume = compute_volume(radius, height)
45    surf_area = compute_surface_area(radius, height)
46    storage_efficiency = volume / surf_area
47    print(f"{name} {storage_efficiency:.1f}")
48
49    name = "#5"
50    radius = 13.02
51    height = 14.29
52    volume = compute_volume(radius, height)
53    surf_area = compute_surface_area(radius, height)
54    storage_efficiency = volume / surf_area
55    print(f"{name} {storage_efficiency:.1f}")
56
57    name = "#6Z"
```

```

58     radius = 5.4
59     height = 8.89
60     volume = compute_volume(radius, height)
61     surf_area = compute_surface_area(radius, height)
62     storage_efficiency = volume / surf_area
63     print(f"{name} {storage_efficiency:.1f}")
64
65     name = "#8Z short"
66     radius = 6.83
67     height = 7.62
68     volume = compute_volume(radius, height)
69     surf_area = compute_surface_area(radius, height)
70     storage_efficiency = volume / surf_area
71     print(f"{name} {storage_efficiency:.1f}")
72
73     name = "#10"
74     radius = 15.72
75     height = 17.78
76     volume = compute_volume(radius, height)
77     surf_area = compute_surface_area(radius, height)
78     storage_efficiency = volume / surf_area
79     print(f"{name} {storage_efficiency:.1f}")
80
81     name = "#211"
82     radius = 6.83
83     height = 12.38
84     volume = compute_volume(radius, height)
85     surf_area = compute_surface_area(radius, height)
86     storage_efficiency = volume / surf_area
87     print(f"{name} {storage_efficiency:.1f}")
88
89     name = "#300"
90     radius = 7.62
91     height = 11.27
92     volume = compute_volume(radius, height)
93     surf_area = compute_surface_area(radius, height)
94     storage_efficiency = volume / surf_area
95     print(f"{name} {storage_efficiency:.1f}")
96
97     name = "#303"
98     radius = 8.1
99     height = 11.11
100    volume = compute_volume(radius, height)
101    surf_area = compute_surface_area(radius, height)
102    storage_efficiency = volume / surf_area
103    print(f"{name} {storage_efficiency:.1f}")
104
105
106 def compute_volume(radius, height):
107     """Compute and return the volume of a cylinder.
108
109     Parameters
110         radius: the radius of the cylinder
111         height: the height of the cylinder
112     Return: the volume of the cylinder
113     """
114     volume = math.pi * radius**2 * height
115     return volume
116
117
118 def compute_surface_area(radius, height):

```

```
119     """Compute and return the surface area of a cylinder.
120
121     Parameters
122         radius: the radius of the cylinder
123         height: the height of the cylinder
124     Return: the surface area of the cylinder
125     """
126     surf_area = 2 * math.pi * radius * (radius + height)
127     return surf_area
128
129
130 # Start this program by
131 # calling the main function.
132 main()
```

lesson04/teach_stretch.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 # Import the standard math module so that
4 # math.pi can be used in this program.
5 import math
6
7
8 def main():
9     can_sizes = [
10         ["#1 Picnic", 6.83, 10.16, 0.28],
11         ["#1 Tall", 7.78, 11.91, 0.43],
12         ["#2", 8.73, 11.59, 0.45],
13         ["#2.5", 10.32, 11.91, 0.61],
14         ["#3 Cylinder", 10.79, 17.78, 0.86],
15         ["#5", 13.02, 14.29, 0.83],
16         ["#6Z", 5.4, 8.89, 0.22],
17         ["#8Z short", 6.83, 7.62, 0.26],
18         ["#10", 15.72, 17.78, 1.53],
19         ["#211", 6.83, 12.38, 0.34],
20         ["#300", 7.62, 11.27, 0.38],
21         ["#303", 8.1, 11.11, 0.42]
22     ]
23
24     best_store = None
25     best_cost = None
26     max_store_eff = -1
27     max_cost_eff = -1
28
29     # For each can in the can_sizes list, unpack the values
30     # into the variables name, radius, height, and cost.
31     for name, radius, height, cost in can_sizes:
32
33         # Call the compute_storage_efficiency and
34         # compute_cost_efficiency functions.
35         store_eff = compute_storage_efficiency(radius, height)
36         cost_eff = compute_cost_efficiency(radius, height, cost)
37
38         # Print the can size name, storage
39         # efficiency, and cost efficiency.
40         print(f"{name} {store_eff:.1f} {cost_eff:.0f}")
41
42         # If the storage efficiency of the current can size is
43         # greater than the maximum storage efficiency, save then
44         # the current can size name and its storage efficiency.
45         if store_eff > max_store_eff:
46             best_store = name
47             max_store_eff = store_eff
48
49         # If the cost efficiency of the current can size is
50         # greater than the maximum cost efficiency, then save
51         # the current can size name and its cost efficiency.
52         if cost_eff > max_cost_eff:
53             best_cost = name
54             max_cost_eff = cost_eff
55
56     # Print the best storage and cost efficiencies.
57     print("Best can size in storage efficiency:", best_store)
```

```

58     print("Best can size in cost efficiency:", best_cost)
59
60
61 def compute_storage_efficiency(radius, height):
62     """Compute and return the storage efficiency of a steel can size.
63     The storage efficiency is the volume of the can divided by its
64     surface area.
65
66     Parameters
67         radius: the radius of the steel can
68         height: the height of the steel can
69     Return: the storage efficiency of the steel can
69     """
69
71     volume = compute_volume(radius, height)
72     surf_area = compute_surface_area(radius, height)
73     efficiency = volume / surf_area
74     return efficiency
75
76
77 def compute_cost_efficiency(radius, height, cost):
78     """Compute and return the cost efficiency of a steel can size.
79     The cost efficiency is the volume of the can divided by its cost.
80
81     Parameters
82         radius: the radius of the steel can
83         height: the height of the steel can
84         cost: the cost of the steel can
85     Return: the cost efficiency of the steel can
85     """
85
87     volume = compute_volume(radius, height)
88     efficiency = volume / cost
89     return efficiency
90
91
92 def compute_volume(radius, height):
93     """Compute and return the volume of a cylinder.
94
95     Parameters
96         radius: the radius of the cylinder
97         height: the height of the cylinder
98     Return: the volume of the cylinder
98     """
98
100    volume = math.pi * radius**2 * height
101    return volume
102
103
104 def compute_surface_area(radius, height):
105     """Compute and return the surface area of a cylinder.
106
107     Parameters
108         radius: the radius of the cylinder
109         height: the height of the cylinder
110     Return: the surface area of the cylinder
110     """
110
112    surf_area = 2 * math.pi * radius * (radius + height)
113    return surf_area
114
115
116 # Start this program by
117 # calling the main function.
118 main()

```

O4 Prove Assignment: Writing Functions

Purpose

Prove that you can write functions with parameters and call those functions multiple times with arguments.

Problem Statement

Modern computers are capable of performing all sorts of calculations to produce numbers. However, they are also capable of performing calculations to produce art, illustrations, animations, movies, and music.

Assignment

During the previous lesson's milestone, you wrote code to draw at least the sky, clouds, and ground of an outdoor scene. During this lesson, you will write code that draws the remaining objects in your scene. Your program can draw any outdoor scene that you like as long as it meets these requirements:

1. The scene must be outdoor and include part of the sky.
2. The sky must have clouds.
3. The scene must include repetitive objects, such as blades of grass, trees, leaves on a tree, birds, flowers, insects, fish, pickets in a fence, dashed lines on a road, buildings, bales of hay, snowmen, snowflakes, or icicles.

Your program must be divided into functions such as `draw_sky`, `draw_cloud`, `draw_ground`, `draw_bird`, `draw_flower`, `draw_insect`, `draw_fish`, or `draw_snowman`. Each repetitive object in your scene should be drawn by a function that your program calls repeatedly, once for each repeated object. For example, your program could include a function named `draw_leaf` that your program repeatedly calls to draw each leaf on a tree at a different location.

As you write your program, write it so that it draws objects in the order of farthest away to nearest. For example, your program should draw the sky first, then clouds, then the ground, then trees, then insects in the trees. Be creative.

Helpful Documentation

- The prepare content for lesson 2 explains [how to call functions](#).
- The prepare content for the previous lesson explains [how to write functions](#).
- The prepare content for this lesson lesson explains the [properties of a good function](#).
- The [prove milestone](#) of the previous lesson describes this assignment in more detail and contains additional helpful documentation.
- The documentation for the Draw 2D library includes [example programs](#), a [function reference](#), and a table of [supported colors](#).

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and verify that it correctly opens a window and draws within that window a complete outdoor scene that contains the sky, clouds, the ground, and other elements.

Exceeding the Requirements

If your program fulfills the requirements for this assignment as described in the previous prove milestone and the Assignment section above, your program will earn 93% of the possible points. In order to earn the remaining 7% of points, you will need to add one or more features to your program so that it exceeds the requirements. If you wish to exceed the requirements of this assignment, you could write code to add more items to your scene. The most efficient way to add more items to your scene is to write a function that draws something, such as `draw_pebble`, `draw_snowman`, or `draw_fish` and then call that function many times in your `draw_scene` function.

Ponder

After you finish this assignment, congratulate yourself because you wrote a Python program with many user-defined functions. As you wrote your program, what did you learn about organizing a program into functions? Did you learn that you can work more efficiently by writing a function once and calling it many times with different arguments?

Submission

To submit your program, return to I-Learn and do these two things:

1. Upload your program (the .py file) for feedback.
2. Add a submission comment that specifies the grading category that best describes your program along with a one or two sentence justification for your choice. The grading criteria are:
 - a. Some attempt made
 - b. Developing but significantly deficient
 - c. Slightly deficient
 - d. Meets requirements
 - e. Exceeds requirements

05 Prepare: Testing Functions

During this lesson, you will learn to use a more systematic approach to developing code. Specifically, you will learn how to write test functions that automatically verify that program functions are correct. You will learn how to use a Python module named `pytest` to run your test functions, and you will learn how to read the output of `pytest` to help you find and fix mistakes in your code.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

Inefficient Testing

During previous lessons, you tested your programs by running them, typing user input, reading the program's output, and verifying that the output was correct. This is a valid way to test a program. However, it is time consuming, tedious, and error prone. A much better way to test a program is to test its functions individually and to write separate **test functions** that *automatically* verify that the program's functions are correct.

In this course, you will write test functions in a Python file that is separate from your Python program. In other words, you will keep normal program code and test code in separate files.

Assert Statements

In a computer program, an **assertion** is a statement that causes the computer to check if a comparison is true. When the computer checks the comparison, if the comparison is true, the computer will continue to execute the code in the program. However, if the comparison is false, the computer will raise an `AssertionError`, which will likely cause the program to terminate. (In [lesson 10](#) you will learn how to write code to handle errors so that a program won't terminate when the computer raises an error.)

A programmer writes assertions in a program to inform the computer of comparisons that must be true in order for the program to run successfully. The Python keyword to write an assertion is `assert`. Imagine a program used by a bank to track account balances, deposits, and withdrawals. A programmer might write the first few lines of the `deposit` function like this:

```
1 def deposit(amount):  
2     # In order for this program to work correctly and
```

```
3     # for the bank records to be correct, we must not
4     # allow someone to deposit a zero or negative amount.
5     assert amount > 0
6     :
```

The `assert` statement at [line 5](#) in the previous example will cause the computer to check if the `amount` is greater than zero (0). If the `amount` is greater than zero, the computer will continue to execute the program. However, if the `amount` is zero or less (negative), the computer will raise an `AssertionError`, which will likely cause the program to terminate.

A programmer can write any valid Python comparison in an `assert` statement. Here are a few examples from various unrelated programs:

```
assert temperature < 0
assert len(given_name) > 0
assert balance == 0
assert school_year != "senior"
```

The pytest Module

[pytest](#) is a third-party Python module that makes it easy to write and run test functions. There are other Python testing modules besides `pytest`, but `pytest` seems to be the easiest to use. `pytest` is not a standard Python module. It is a third-party module. This means that when you installed Python on your computer, `pytest` was not installed, and you will need to install `pytest` in order to use it. During the checkpoint of this lesson, you will use a standard Python module named `pip` to install `pytest`.

`pytest` allows a programmer to write simple test functions that use the Python `assert` statement to verify that a function returns a correct result. For example, if we want to verify that the built-in `min` function works correctly, we could write a test function like this:

```
def test_min():
    assert min(7, -3, 0, 2) == -3
```

In the previous function, the `assert` statement will cause the computer to first call the `min` function and pass 7, -3, 0, and 2 as arguments to the `min` function. The `min` function will find the minimum value of its parameters and return that minimum value. Then the `assert` statement will compare the returned minimum value to -3. If the returned value is not -3, the `assert` statement will raise an exception which will cause `pytest` to print an error message.

Comparing Floating Point Numbers

Within a computer's memory, everything (all numbers, text, sound, pictures, movies, everything) is stored using the binary number system. While executing a Python program, a computer stores integers in binary in a way that exactly represents the integers. For example, a computer stores the integer 23 as 00010111 in binary which is an exact representation of decimal 23. However, a computer approximates floating point numbers (numbers with digits after the decimal place). For example, while executing a Python program, a computer stores the floating point number 23.7 as binary 01000000001101110110011001100110011001100110011001100110011001100110011. This binary number is actually 23.6999999999999928945726424 in decimal which is an approximation to 23.7

Because computers approximate floating point numbers, we must carefully compare floating point numbers in our test functions. It is bad practice to check if floating point numbers are equal using just the equality operator (`==`). The `pytest` module contains a function named `approx` to help us compare floating point numbers. For example, to test the `math.sqrt` function, we could write a test function like this:

```
def test_sqrt():
    assert math.sqrt(5) == approx(2.24, 0.01)
```

The assert statement in the previous test function verifies that the value returned from `math.sqrt(5)` is within 1% (0.01) of 2.24.

How to Test a Function

To test a function you should do the following:

1. Write a function that is part of your normal Python program.
 2. Think about different parameter values that will cause the computer to execute all the code in your function and will possibly cause your function to fail or return an incorrect result.
 3. In a separate Python file, write a test function that calls your program function and uses an `assert` statement to *automatically* verify that the value returned from your program function is correct.
 4. Use `pytest` to run the test function.
 5. Read the output of `pytest` and use that output to help you find and fix mistakes in both your program function and test function.

Example

Below is a simple function named `cels_from_fahr` that converts a temperature in Fahrenheit to Celsius and returns the Celsius temperature. The `cels_from_fahr` function

is part of a larger Python program in a file named `weather.py`.

```
1 # weather.py
2
3 def cels_from_fahr(fahr):
4     """Convert a temperature in Fahrenheit to
5     Celsius and return the Celsius temperature.
6     """
7     cels = (fahr - 32) * 5 / 9
8     return cels
```

We want to test the `cels_from_fahr` function. From the function header at [line 3](#) in `weather.py`, we see that `cels_from_fahr` takes one parameter named `fahr`. To adequately test this function, we should call it at least three times with the following arguments.

- a negative number
- zero
- a positive number

In a separate file named `test_weather.py` we write a test function named `test_cels_from_fahr` as follows:

```
1 # test_weather.py
2
3 from weather import cels_from_fahr
4 from pytest import approx
5 import pytest
6
7 def test_cels_from_fahr():
8     """Test the cels_from_fahr function by calling it and
9     comparing the values it returns to the expected values.
10    Notice this test function uses pytest.approx to compare
11    floating point numbers.
12    """
13    assert cels_from_fahr(-25) == approx(-31.66667)
14    assert cels_from_fahr(0) == approx(-17.77778)
15    assert cels_from_fahr(32) == approx(0)
16    assert cels_from_fahr(70) == approx(21.1111)
17
18    # Call the main function that is part of pytest so that the
19    # computer will execute the test functions in this file.
20    pytest.main(["-v", "--tb=line", "-rN", __file__])
```

Notice in `test_weather.py` at [lines 13–16](#) that the test function `test_cels_from_fahr` calls the program function `cels_from_fahr` four times: once with a negative number, once with zero, and twice with positive numbers. Notice also that the test function uses `assert` and `approx`.

After writing the test function, we use `pytest` to run the test function. At [line 20](#), instead of writing a call to the `main` function, as we do in program files, we write a call to the `pytest.main` function. In CSE 111, at the bottom of all test files, we will write a call to `pytest.main` exactly as shown on [line 20](#). This call to `pytest.main` will cause the `pytest`

module to run our test functions. When pytest runs our test functions, it will produce output that tells us if the tests passed or failed like this:

```
> python test_weather.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.1
rootdir: C:\Users\cse111\lesson05
collected 1 item

test_weather.py::test_cels_from_fahr PASSED [100%]

===== 1 passed in 0.10s =====
```

As shown above, pytest runs the `test_cels_from_fahr` function which calls the `cels_from_fahr` function four times and verifies that `cels_from_fahr` returns the correct value each time. We can see from the output of pytest, "PASSED [100%]" and "1 passed", that the `cels_from_fahr` function returned the expected (correct) result all four times.

Separating Program Code from Test Code

In CSE 111, we will write test functions in a file separate from program functions. It is a good idea to separate test functions and program functions because the separation makes it easy to release a program to users without releasing the test functions to them. In general, users of a program don't want the test functions. One consequence of writing program functions and test functions in separate files is that we must add an import statement at the top of the test file that imports all the program functions that will be tested.

Line 3 from `test_weather.py` above is an example of an import statement that imports functions from a program file. Line 3 matches this template:

```
from file_name import function_1, function_2, ... function_N
```

When the computer imports functions from a file, the computer immediately executes all statements that are not written inside a function. This includes the statement to call the `main` function:

```
# Start this program by
# calling the main function.
main()
```

This means that when we run our test functions, the computer will import our program functions and at the same time, will execute the call to `main()` which will start the program executing. However, we don't want the computer to execute the program while it is executing the test functions, so we have a problem. How can we get the computer to import the program functions without executing the `main` function? Fortunately, the developers of Python gave us a solution to this problem. Instead of writing the following code to start our program running:

```
# Start this program by
# calling the main function.
main()
```

We write an `if` statement above the call to `main()` like this:

```
# If this file is executed like this:
# > python program.py
# then call the main function. However, if this file is simply
# imported (e.g. into a test file), then skip the call to main.
if __name__ == "__main__":
    main()
```

Writing the `if` statement above the call to `main()` is the correct way to write code to start a program. The Python programming language guarantees that when the computer imports the program functions (in order to test them), the comparison in the `if` statement will be false, so the computer will skip the call to `main()`. At another time, when the computer executes the program (not the test functions), the comparison in the `if` statement will be true, which will cause the computer to call the `main` function and start the program.

Which Program Functions Should We Test?

Because we are responsible computer programmers and want to ensure that all of our program functions work correctly, we would like to test all program functions. In other words, we would like to write at least one test function for each program function. However, this may not always be possible. The easiest program functions to test are the functions that have parameters and return a value. The hardest program functions to test are the functions that get user input, print results to a terminal window, or draw something to a window. During the next eight lessons in CSE 111, we will usually write one test function for each program function that is easy to test, meaning each function that does not get user input and does not print to a terminal window. This means that you won't write a test function for your program's `main` function because `main` usually gets user input and prints to a terminal window.

Video

Watch the following video that shows a BYU-Idaho faculty member writing two test functions and using `pytest` to run them.

[Writing a Test Function](#) (20 minutes)

Documentation

The official online documentation for pytest contains much more information about using pytest. The following pages are the most applicable to CSE 111.

[Create your first test](#)

[assert](#)

[pytest.approx](#)

[pytest.main](#)

Summary

During this lesson, you are learning to write test functions that automatically verify that program functions are working correctly. In CSE 111, you will write test functions in a Python file that is separate from your program file. At the top of the test file, you will import the program functions. Then you will write one test function for each program function, except `main`. Within a test function, you will write `assert` statements that compare the value returned from a program function to the expected value. You will use a standard Python module named `pytest` to run your test functions. When a test fails, you will use the output of `pytest` to help you find and fix the mistakes in your code.

05 Checkpoint: Testing Functions

Purpose

Improve your ability to verify the correctness of functions by writing a test function and running it with pytest.

Assignment

Write a test function that tests a previously written function. Then use pytest to run test functions.

Helpful Documentation

[pip](#) is a standard Python module that you can use to download and install third-party modules. During the checkpoint of this lesson, you will use pip to download and install pytest, so that you can use pytest in your test code.

This [video about the pip module](#) (16 minutes) shows a BYU-Idaho faculty member using pip to install other Python modules.

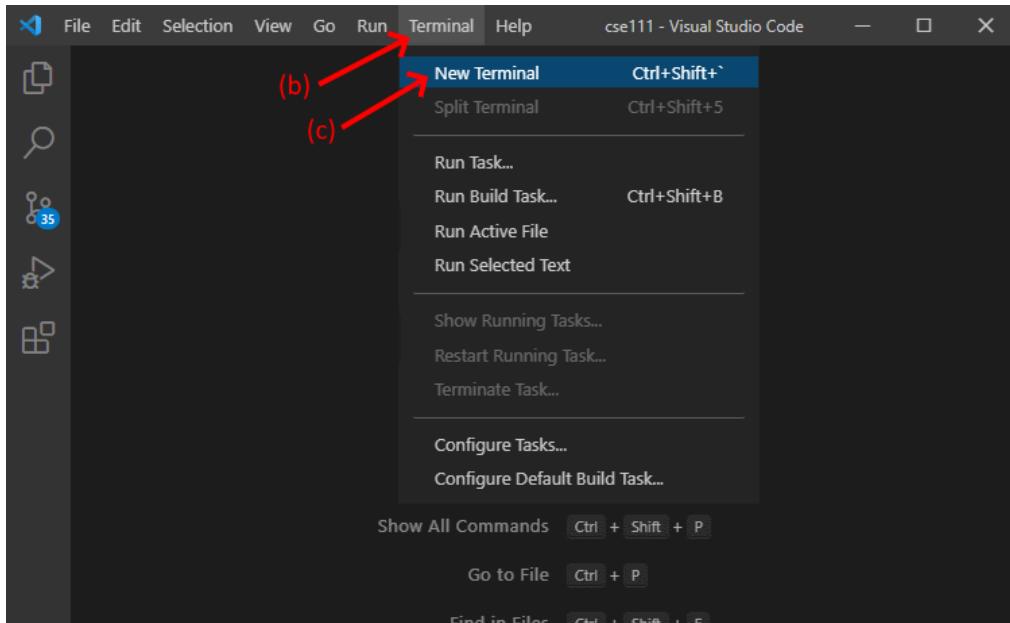
The [prepare content](#) for this lesson explains how to use pytest, assert, and approx to automatically verify that functions are correct. It also contains an [example test function](#) and links to additional documentation about pytest.

This [video about test functions](#) (20 minutes) shows a BYU-Idaho faculty member writing two test functions and using pytest to run them.

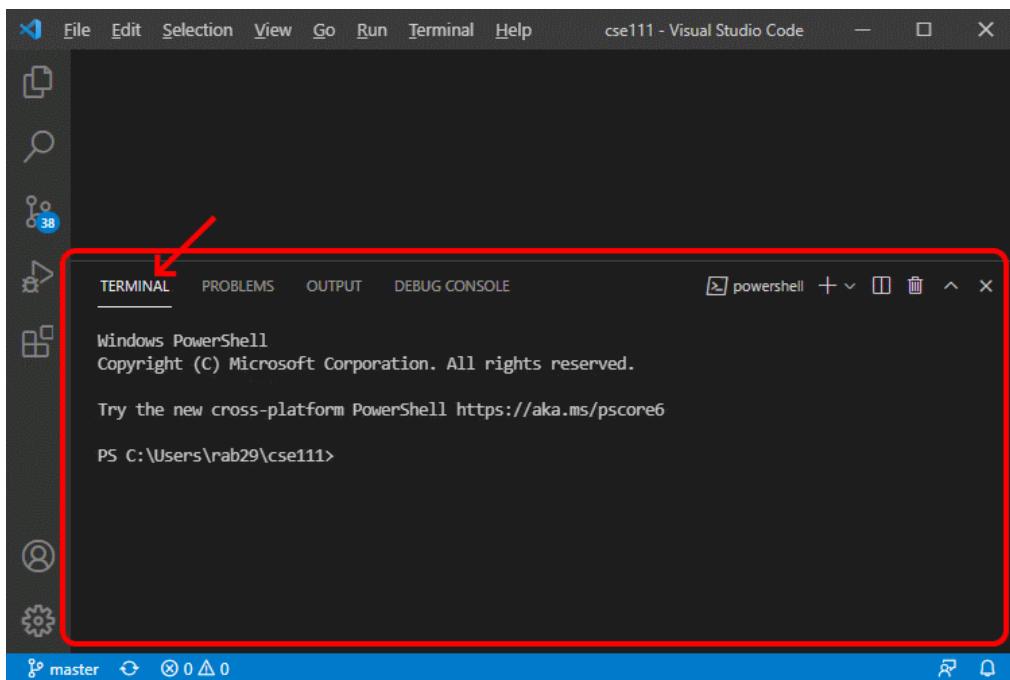
Steps

Do the following:

1. Open a new terminal frame in VS Code by doing the following:
 - a. Open VS Code
 - b. On the menu bar for VS Code, click "Terminal"
 - c. On the menu, click "New Terminal"



This will open a terminal frame at the bottom of the VS Code window. A terminal is a window or frame where a user can type and execute computer commands.



2. Copy and paste the following command into the terminal frame and execute the command by pressing the Enter key. This command will upgrade pip and several other parts of the Python installation modules so that pip will work correctly.
 - o Mac OS users:

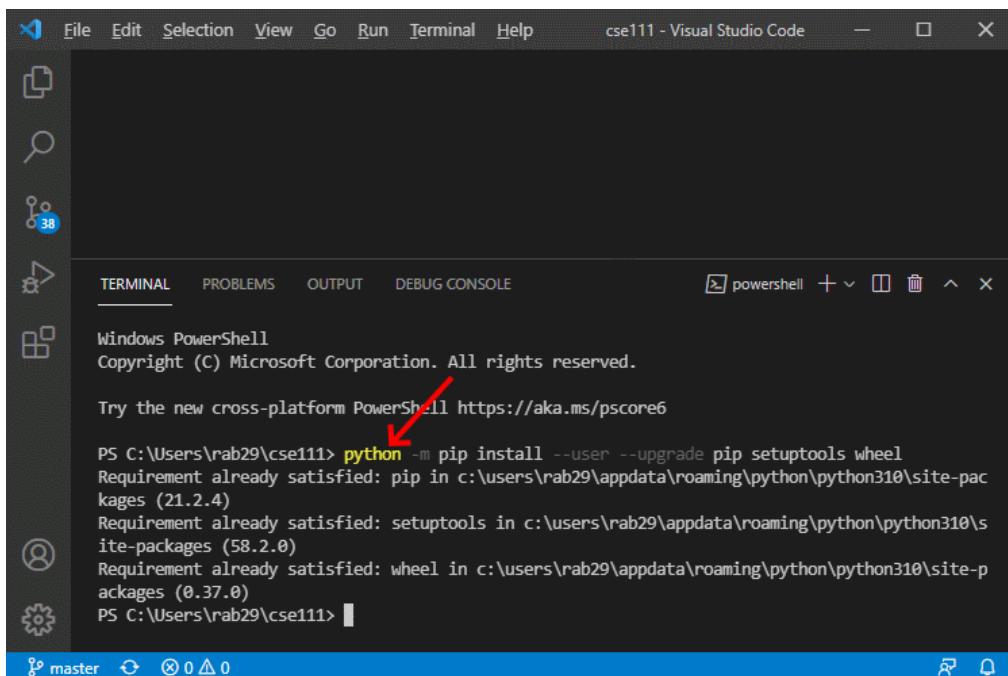
```
python3 -m pip install --user --upgrade pip setuptools wheel
```

- o Windows users:

```
python -m pip install --user --upgrade pip setuptools wheel
```

* If your computer is running the Windows operating system, and the above command doesn't work on your computer, try the py command instead of the python command like this:

```
py -m pip install --user --upgrade pip setuptools wheel
```



3. Install the pytest module by copying, pasting, and executing the following command in the terminal frame.

- o Mac OS users:

```
python3 -m pip install --user pytest
```

- o Windows users:

```
python -m pip install --user pytest
```

* If your computer is running the Windows operating system, and the above command doesn't work on your computer, try the py command instead of the python command like this:

```
py -m pip install --user pytest
```

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Users\rab29\cse111> python -m pip install --user pytest
Collecting pytest
  Using cached pytest-6.2.5-py3-none-any.whl (280 kB)
Requirement already satisfied: attrs>=19.2.0 in c:\users\rab29\appdata\roaming\python\python310\site-packages (from pytest) (21.2.0)
Requirement already satisfied: py>=1.8.2 in c:\users\rab29\appdata\roaming\python\python310\site-packages (from pytest) (1.10.0)
Requirement already satisfied: iniconfig in c:\users\rab29\appdata\roaming\python\python310\site-packages (from pytest) (1.1.1)

```

4. Download these two Python files: [words.py](#) and [test_words.py](#) and save them in the same folder.
5. Open the downloaded words.py file in VS Code. Notice the words.py file contains two small functions named prefix and suffix. Notice also that each function has a documentation string (a triple quoted string immediately below a function header) that describes what the function does. Read the documentation strings for both functions.
6. Open the downloaded test_words.py file in VS Code. In test_words.py examine the test_prefix function. Notice that it takes no parameters and contains nine assert statements. Each assert statement calls the prefix function and then compares the value returned from the prefix function to the expected value.
7. In test_words.py write a function named test_suffix that is similar to the test_prefix function. The test_suffix function should take no parameters and contain nine assert statements that call the suffix function with these parameters:

Arguments		Expected Return
s1	s2	Value
""	""	""
""	"correct"	""
"clear"	""	""
"angelic"	"awesome"	""

Arguments		Expected Return
s1	s2	Value
"found"	"profound"	"found"
"ditch"	"itch"	"itch"
"happy"	"funny"	"y"
"tired"	"fatigued"	"ed"
"swimming"	"FLYING"	"ing"

- Save your `test_words.py` file and run it by clicking the green run icon in VS Code.

Testing Procedure

Verify that your test program works correctly by following each step in this procedure:

- Run your test program and ensure that your test program's output is similar to the sample run output below.

```
> python test_words.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson05
collected 2 items

test_words.py::test_prefix PASSED [ 50%]
test_words.py::test_suffix PASSED [100%]

===== 2 passed in 0.09s =====
```

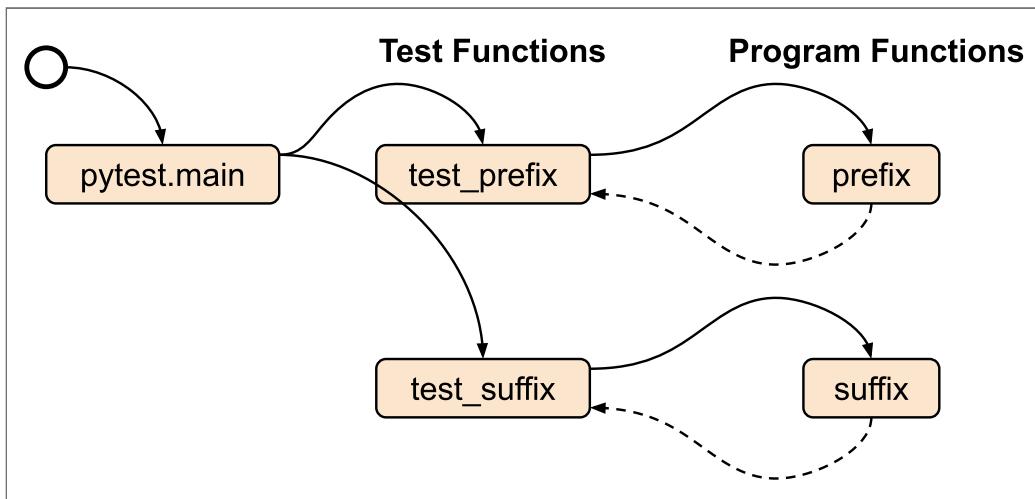
Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample test functions by calling the `pytest.main` function. While executing the `pytest.main` function, the computer calls the `test_prefix` function. While executing the `test_prefix` function, the computer calls the `prefix` function. Then while still executing

the `pytest.main` function, the computer calls the `test_suffix` function. While executing the `test_suffix` function, the computer calls the `suffix` function.



Ponder

During this assignment, you downloaded a Python file that contains two program functions named `prefix` and `suffix`. You wrote a test function named `test_suffix` that is similar to the `test_prefix` function that was given to you. You used `pytest` to run both test functions and examined the output of `pytest` to verify that the test functions passed. Because the test functions called `prefix` and `suffix` with many different arguments and verified (using `assert`) that the values returned from `prefix` and `suffix` were correct, we can assume that the `prefix` and `suffix` functions work correctly. Do you think writing and running test functions will help you write better programs?

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson05/words.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def prefix(string1, string2):
4     """Return the prefix, if any, that appears in both string1 and
5     string2. In other words, return a string of the characters
6     that appear at the beginning of both string1 and string2. For
7     example, if string1 is "inconceivable" and string2 is
8     "inconvenient", this function will return "incon".
9
10    Parameters
11        string1: a string of text
12        string2: another string of text
13    Return: a string
14    """
15
16    # Convert both strings to lower case.
17    string1 = string1.lower()
18    string2 = string2.lower()
19
20    # Start at the beginning of both strings.
21    i = 0
22
23    # Repeat until the computer finds two
24    # characters that are not the same.
25    limit = min(len(string1), len(string2))
26    while i < limit:
27        if string1[i] != string2[i]:
28            break
29        i += 1
30
31    # Extract a substring from string1 and return it.
32    pre = string1[0 : i]
33    return pre
34
35 def suffix(string1, string2):
36     """Return the suffix, if any, that appears in both string1 and
37     string2. In other words, return a string of the characters
38     that appear at the end of both string1 and string2. For
39     example, if string1 is "hilarious" and string2 is "nefarious",
40     this function will return "arious".
41
42    Parameters
43        string1: a string of text
44        string2: another string of text
45    Return: a string
46    """
47
48    # Convert both strings to lower case.
49    string1 = string1.lower()
50    string2 = string2.lower()
51
52    # Start at the end of both strings.
53    i1 = len(string1) - 1
54    i2 = len(string2) - 1
55
56    # Repeat until the computer finds two
57    # characters that are not the same.
58    limit = min(len(string1), len(string2))
```

```
58     for _ in range(limit):
59         if string1[i1] != string2[i2]:
60             break
61         i1 -= 1
62         i2 -= 1
63
64     # Extract a substring from string1 and return it.
65     suf = string1[i1+1 : ]
66
67     return suf
```

lesson05/test_words.py

```
1 """Verify that the prefix and suffix functions work correctly."""
2
3 from words import prefix, suffix
4 import pytest
5
6
7 def test_prefix():
8     """Verify that the prefix function works correctly.
9     Parameters: none
10    Return: nothing
11    """
12     pre = prefix("upbeat", "upgrade")
13     assert isinstance(pre, str), "prefix function must return a string"
14
15     assert prefix("cat", "catalog") == "cat"
16     assert prefix("", "") == ""
17     assert prefix("", "correct") == ""
18     assert prefix("clear", "") == ""
19     assert prefix("happy", "funny") == ""
20     assert prefix("cat", "catalog") == "cat"
21     assert prefix("dogmatic", "dog") == "dog"
22     assert prefix("jump", "joyous") == "j"
23     assert prefix("upbeat", "upgrade") == "up"
24     assert prefix("Disable", "dIstasteful") == "dis"
25
26
27 # Call the main function that is part of pytest so that the
28 # computer will execute the test functions in this file.
29 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

lesson05/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 """Verify that the prefix and suffix functions work correctly."""
4
5 from words import prefix, suffix
6 import pytest
7
8
9 def test_prefix():
10     """Verify that the prefix function works correctly.
11     Parameters: none
12     Return: nothing
13     """
14     assert prefix("", "") == ""
15     assert prefix("", "correct") == ""
16     assert prefix("clear", "") == ""
17     assert prefix("happy", "funny") == ""
18     assert prefix("cat", "catalog") == "cat"
19     assert prefix("dogmatic", "dog") == "dog"
20     assert prefix("jump", "joyous") == "j"
21     assert prefix("unwise", "ungrateful") == "un"
22     assert prefix("Disable", "dIstasteful") == "dis"
23
24
25 def test_suffix():
26     """Verify that the suffix function works correctly.
27     Parameters: none
28     Return: nothing
29     """
30     assert suffix("", "") == ""
31     assert suffix("", "correct") == ""
32     assert suffix("clear", "") == ""
33     assert suffix("angelic", "awesome") == ""
34     assert suffix("found", "profound") == "found"
35     assert suffix("ditch", "itch") == "itch"
36     assert suffix("happy", "funny") == "y"
37     assert suffix("tired", "fatigued") == "ed"
38     assert suffix("swimming", "FLYING") == "ing"
39
40
41 # Call the main function that is part of pytest so that
42 # the test functions in this file will start executing.
43 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

05 Team Activity: Testing and Fixing Functions

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Purpose

Writing and running test functions often help a software developer find mistakes in code. During this assignment, you will write three test functions. Then use pytest to run the test functions and use the output of pytest to help you find and fix errors in some program functions.

Problem Statement

Most people around the world today have at least two names, a family name and a given name. In the United States, we usually write a person's given name followed by his family name. However, when a computer lists names in alphabetical order, it is convenient to list the family name first and then the given name like this:

- Toussaint; Marie
- Toussaint; Olivier
- Washington; George
- Washington; Martha

When writing a program that alphabetizes names, it is often helpful to have the following three functions.

```
make_full_name
    Combines a person's given name and family name into one string with the
    family name first

extract_family_name
    Extracts a person's family name from his full name

extract_given_name
    Extracts a person's given name from his full name
```

A programmer has already written those three functions. However, there are mistakes in at least two of the three functions.

Assignment

As a team, write three test functions named `test_make_full_name`, `test_extract_family_name`, and `test_extract_given_name`. Each of the test functions will test one of three previously written program functions. Use pytest to run the test functions and find and fix the mistakes, if any, that are in program functions.

Helpful Documentation

The [prepare content](#) for this lesson explains how to use pytest, assert, and approx to automatically verify that functions are correct. It also contains an [example test function](#) and links to additional documentation about pytest.

This [video about test functions](#) (20 minutes) shows a BYU-Idaho faculty member writing two test functions and using pytest to run them.

Steps

Do the following:

1. Download the [names.py](#) Python file and save it in the same folder where you will save your Python test program. Then open the downloaded file in VS Code. Notice that the downloaded file contains three small functions named: `make_full_name`, `extract_family_name`, and `extract_given_name`. Notice also that each function has a documentation string (a triple quoted string immediately below the function header) that describes what the function does. Read the documentation strings for all three functions. The code in the functions may contain some mistakes.
2. Using VS Code, open a new Python file and copy and paste the following import statements at the top of the file.

```
from names import make_full_name, \
    extract_family_name, extract_given_name
import pytest
```

Save the file with the name `test_names.py` in the same folder where you downloaded and saved the `names.py` file.

3. In `test_names.py`, write a function named `test_make_full_name` that takes no parameters. Write assert statements inside this function to test the value returned from the `make_full_name` function. If you are not sure what the `make_full_name` function does or how to test it, read the documentation string that is at the top of the `make_full_name` function in the `names.py` file.

4. In `test_names.py` write a function named `test_extract_family_name` that takes no parameters. Write assert statements inside this function to test the value returned from the `extract_family_name` function.
5. In `test_names.py` write a function named `test_extract_given_name` that takes no parameters. Write assert statements inside this function to test the value returned from the `extract_given_name` function.
6. At the bottom of your `test_names.py` file, write a call to the `pytest.main` function like this:

```
# Call the main function that is part of pytest so that the
# computer will execute the test functions in this file.
pytest.main(["-v", "--tb=line", "-rN", __file__])
```
7. Save your `test_names.py` file and run it by clicking the green run icon in VS Code.
8. Read the output from pytest and use the output to help you find and fix any errors that are in your test functions or the `make_full_name`, `extract_family_name`, and `extract_given_name` functions.
9. Repeat steps 7 and 8 until you have found and fixed all the mistakes and your three test functions pass.

Core Requirements

1. Write `test_make_full_name` so that it tests `make_full_name` with various names, including long names, short names, and hyphenated names. Fix the mistake in the `make_full_name` function.
2. Write `test_extract_family_name` so that it tests `extract_family_name` with various names, including long names, short names, and hyphenated names.
3. Write `test_extract_given_name` so that it tests `extract_given_name` with various names, including long names, short names, and hyphenated names. Fix the mistake in the `extract_given_name` function.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. In the United States, mailing addresses are supposed to be written in this form: *number street, city, state zipcode*
For example: 525 S Center St, Rexburg, ID 83460

Download and save this Python file named [address.py](#). Open a new Python file named test_address.py and write a test function named test_extract_city that verifies that the extract_city function works correctly.

2. Write a test function named test_extract_state that verifies that the extract_state function works correctly.
3. Write a test function named test_extract_zipcode that verifies that the extract_zipcode function works correctly.

Testing Procedure

Before you fix the mistakes in the make_full_name and extract_given_name functions, pytest will print output similar to this:

```
> python test_names.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.1
rootdir: C:\Users\cse111\lesson05
collected 3 items

test_names.py::test_make_full_name FAILED [ 33%]
test_names.py::test_extract_family_name PASSED [ 66%]
test_names.py::test_extract_given_name FAILED [100%]

===== FAILURES =====
C:\Users\cse111\early-functions\docs\Lesson05\teach_solution.py:
AssertionError: assert 'Smith-Jones;Ava' == 'Smith-Jones; Ava'
C:\Users\cse111\early-functions\docs\lesson05\names.py:31:
ValueError: substring not found
===== 2 failed, 1 passed in 0.14s =====
```

After you fix the mistakes in the make_full_name and extract_given_name functions, pytest will print output similar to this:

```
> python test_names.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy-0.1
rootdir: C:\Users\cse111\lesson05
collected 3 items

test_names.py::test_make_full_name PASSED [ 33%]
test_names.py::test_extract_family_name PASSED [ 66%]
test_names.py::test_extract_given_name PASSED [100%]

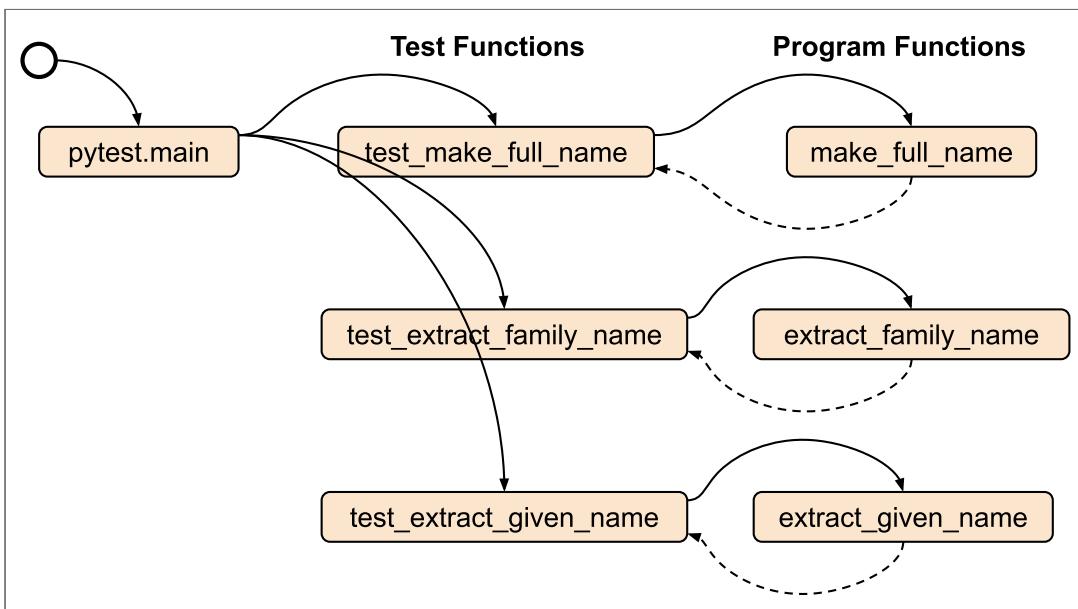
===== 3 passed in 0.12s =====
```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#) and the [stretch solution](#). Please **do not look at the sample solutions** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Call Graph

The following call graph shows the function calls and returns in the sample solution for this assignment. From this call graph we see that the computer starts executing the sample test functions by calling the `pytest.main` function. While executing the `pytest.main` function, the computer calls the `test_make_full_name` function which calls the `make_full_name` function. Then while still executing `pytest.main`, the computer calls the `test_extract_family_name` function which calls the `extract_family_name` function. Then while still executing `pytest.main`, the computer calls the `test_extract_given_name` function which calls the `extract_given_name` function.



Ponder

During this assignment, you downloaded three program functions that work with a person's name. You wrote three test functions that each test one of the program functions. You used `pytest` to run your test functions and used the output of `pytest` to help you find and fix the mistakes in the program functions. How will writing and running test functions help you write better programs?

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson05/names.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def make_full_name(given_name, family_name):
4     """Return a string in this form "family_name; given_name". For
5     example, if this function were called like this:
6     make_full_name("Sally", "Brown"), it would return "Brown; Sally".
7
8     Parameters
9         given_name: a string that contains a person's given name
10        family_name: a string that contains a person's family name
11    Return: a string in the form "family_name; given_name"
12    """
13    full_name = f"{family_name};{given_name}"
14    return full_name
15
16
17 def extract_family_name(full_name):
18     """Extract and return the family name from a string in this form:
19     "family_name; given_name". For example, if this function were
20     called like this:
21     extract_family_name("Brown; Sally"), it would return "Brown".
22
23     Parameter:
24         full_name: a string in the form "family_name; given_name"
25     Return: a string that contains a person's family name
26     """
27     # Find the index where ";" appears within the full name string.
28     semicolon_index = full_name.index("; ")
29
30     # Extract a substring from the full name and return it.
31     family_name = full_name[0 : semicolon_index]
32     return family_name
33
34
35 def extract_given_name(full_name):
36     """Extract and return the given name from a string in this form:
37     "family_name; given_name". For example, if this function were
38     called like this:
39     extract_given_name("Brown; Sally"), it would return "Sally".
40
41     Parameter:
42         full_name: a string in the form "family_name; given_name"
43     Return: a string that contains a person's given name
44     """
45     # Find the index where ";" appears within the full name string.
46     semicolon_index = full_name.index("/ ")
47
48     # Extract a substring from the full name and return it.
49     given_name = full_name[semicolon_index + 2 : ]
50     return given_name
```

lesson05/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 Verify that the make_full_name, extract_family_name,
4 and extract_given_name functions work correctly.
5 """
6
7 from names import make_full_name,
8     extract_given_name, extract_family_name
9 import pytest
10
11
12 def test_make_full_name():
13     """Verify that the make_full_name
14     function returns correct results.
15
16     Parameters: none
17     Return: nothing
18     """
19     assert make_full_name("Ava", "Smith-Jones") == "Smith-Jones; Ava"
20     assert make_full_name("James", "Madison") == "Madison; James"
21     assert make_full_name("J", "Ng") == "Ng; J"
22     assert make_full_name("", "") == "; "
23
24
25 def test_extract_family_name():
26     """Verify that the extract_family_name
27     function returns correct results.
28
29     Parameters: none
30     Return: nothing
31     """
32     assert extract_family_name("Smith-Jones; Ava") == "Smith-Jones"
33     assert extract_family_name("Madison; James") == "Madison"
34     assert extract_family_name("Ng; J") == "Ng"
35     assert extract_family_name("; ") == ""
36
37
38 def test_extract_given_name():
39     """Verify that the extract_given_name
40     function returns correct results.
41
42     Parameters: none
43     Return: nothing
44     """
45     assert extract_given_name("Smith-Jones; Ava") == "Ava"
46     assert extract_given_name("Madison; James") == "James"
47     assert extract_given_name("Ng; J") == "J"
48     assert extract_given_name("; ") == ""
49
50
51 # Call the main function that is part of pytest so that
52 # the test functions in this file will start executing.
53 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

lesson05/address.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def extract_city(full_address):
4     """Extract and return the name of a city from
5     a properly formatted U.S. mailing address.
6     Parameter
7         full_address: a U.S. mailing address in this format:
8             number and street, city, state zipcode
9     Return: the city name
10    """
11    full_address = full_address.strip()
12    last_comma_index = full_address.rindex(",")
13    mid_comma_index = full_address.rindex(",", 0, last_comma_index)
14    city = full_address[mid_comma_index + 1 : last_comma_index]
15    city = city.strip()
16    return city
17
18
19 def extract_state(full_address):
20     """Extract and return the two letter abbreviation for
21     a state from a properly formatted U.S. mailing address.
22     Parameter
23         full_address: a U.S. mailing address in this format:
24             number and street, city, state zipcode
25     Return: the two letter state abbreviation
26    """
27    full_address = full_address.strip()
28    last_comma_index = full_address.rindex(",")
29    last_space_index = full_address.rindex(" ")
30    state = full_address[last_comma_index + 1 : last_space_index]
31    state = state.strip()
32    return state
33
34
35 def extract_zipcode(full_address):
36     """Extract and return the ZIP code from
37     a properly formatted U.S. mailing address.
38     Parameter
39         full_address: a U.S. mailing address in this format:
40             number and street, city, state zipcode
41     Return: the ZIP code
42    """
43    full_address = full_address.strip()
44    last_space_index = full_address.rindex(" ")
45    zipcode = full_address[last_space_index + 1 : ]
46    return zipcode
```

lesson05/teach_stretch.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 Verify that the extract_city, extract_state,
4 and extract_zipcode functions work correctly.
5 """
6
7
8 from address import extract_city, extract_state, extract_zipcode
9 import pytest
10
11 def test_extract_city():
12     """Verify that the extract_city function returns correct results.
13     Parameters: none
14     Return: nothing
15     """
16     assert extract_city("123 W Main, Rexburg, ID 83440") == "Rexburg"
17     assert extract_city("78 Pine St, Avon Park, FL 33825") == "Avon Par
18
19
20 def test_extract_state():
21     """Verify that the extract_state function returns correct results.
22     Parameters: none
23     Return: nothing
24     """
25     assert extract_state("123 W Main, Rexburg, ID 83440") == "ID"
26     assert extract_state("78 Pine St, Avon Park, FL 33825") == "FL"
27
28
29 def test_extract_zipcode():
30     """Verify that the extract_zipcode
31     function returns correct results.
32
33     Parameters: none
34     Return: nothing
35     """
36     assert extract_zipcode("123 W Main, Rexburg, ID 83440") == "83440"
37     assert extract_zipcode("78 Pine St, Avon Park, FL 33825") == "33825
38
39 # Call the main function that is part of pytest so that
40 # the test functions in this file will start executing.
41 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

05 Prove Milestone: Testing and Fixing Functions

Purpose

Prove that you can write a Python program and write and run test functions to help you find and fix mistakes.

Problem Statement

The **Turing test**, named after Alan Turing, is a test of a computer's ability to make conversation that is indistinguishable from human conversation. A computer that could pass the Turing test would need to understand sentences typed by a human and respond with sentences that make sense.

In English and many other languages, **grammatical quantity** (also known as grammatical number) is an attribute of nouns, pronouns, adjectives, and verbs that expresses count distinctions, such as "one", "two", "some", or "many". The grammatical quantity of the words in a sentence must match. In English, there are two categories of grammatical quantity: single and plural. For example, here are three English sentences that contain **single** nouns, pronouns, adjectives, and verbs:

The boy laughs.
One dog eats.
She drinks water.

Here are three English sentences that contain **plural** nouns, pronouns, adjectives, and verbs:

Two birds fly.
Some animals eat grass.
Many cars are red.

Grammatical tense is an attribute of verbs that expresses when an action happened. Many languages include past, present, and future tenses. For example, here are three English sentences, the first with past tense, the second with present tense, and the third with future tense:

The cat walked.
The cat walks.
The cat will walk.

Assignment

Write a Python program named `sentences.py` that generates simple English sentences. During this prove milestone, you will write and test functions that generate sentences with three parts:

1. a determiner (sometimes known as an article)
2. a noun
3. a verb

For example:

```
A cat laughed.  
One man eats.  
The woman will think.  
Some girls thought.  
Many dogs run.  
Many men will write.
```

For this milestone , your program must include at least these four functions:

1. `main`
2. `get_determiner`
3. `get_noun`
4. `get_verb`

You may add other program functions if you want. The functions `get_determiner`, `get_noun`, and `get_verb`, must randomly choose a word from a list of words and return the randomly chosen word.

In addition, for this milestone assignment, you must write a file named `test_senectences.py` that contains three functions named as follows:

1. `test_get_determiner`
2. `test_get_noun`
3. `test_get_verb`

All the functions that you must write for this milestone assignment are described in the Steps section below.

Helpful Documentation

- In CSE 110, you studied Python lists. You should recall that we create a Python list with square brackets and commas like this list of strings:

```
# Create a list of strings and assign
```

```
# the list to a variable named words.  
words = ["boy", "girl", "cat", "dog", "bird", "house"]
```

- The standard Python `random` module includes a [function named `choice`](#) that will randomly choose one element from a list and return that element. The `choice` function is easy to call like this:

```
import random  
  
# Create a list of strings and assign  
# the list to a variable named words.  
words = ["boy", "girl", "cat", "dog", "bird", "house"]  
  
# Call the random.choice function which will choose  
# one string from the words list. Store the chosen  
# string in a variable named word.  
word = random.choice(words)
```

- The Python [str.capitalize method](#) will capitalize the first letter in a word. The `capitalize` method is easy to call like this:

```
# This could be any word from any source.  
word = "horse"  
  
# Call the capitalize method which will  
# capitalize the first letter of the word.  
cap_word = word.capitalize()
```

- The [prepare content](#) for this lesson explains how to use `pytest` and `assert` to automatically verify that functions are correct. It also contains an [example test function](#) and links to additional documentation about `pytest`.
- This [video about test functions](#) (20 minutes) shows a BYU-Idaho faculty member writing two test functions and using `pytest` to run them.

Steps

Do the following:

- Using VS Code, create a new file, import the `random` module at the top of the file, and save the file as `sentences.py`
- Copy and paste the following `get_determiner` function into your program.

```
def get_determiner(quantity):  
    """Return a randomly chosen determiner. A determiner is  
    a word like "the", "a", "one", "two", "some", "many".  
    If quantity == 1, this function will return either "a",  
    "one", or "the". Otherwise this function will return  
    either "two", "some", "many", or "the".  
  
    Parameter  
    quantity: an integer.
```

```

    If quantity == 1, this function will return a
    determiner for a single noun. Otherwise this
    function will return a determiner for a plural
    noun.

Return: a randomly chosen determiner.

"""
if quantity == 1:
    words = ["a", "one", "the"]
else:
    words = ["two", "some", "many", "the"]

# Randomly choose and return a determiner.
word = random.choice(words)
return word

```

3. Use the `get_determiner` function as an example to help you write the `get_noun` function. The `get_noun` function must have the following header and fulfill the requirements of the following documentation string.

```

def get_noun(quantity):
    """Return a randomly chosen noun.
    If quantity == 1, this function will
    return one of these ten single nouns:
        "bird", "boy", "car", "cat", "child",
        "dog", "girl", "man", "rabbit", "woman"
    Otherwise, this function will return one of
    these ten plural nouns:
        "birds", "boys", "cars", "cats", "children",
        "dogs", "girls", "men", "rabbits", "women"

    Parameter
        quantity: an integer that determines if
            the returned noun is single or plural.
    Return: a randomly chosen noun.
"""

```

4. Use the `get_determiner` function as an example to help you write the `get_verb` function. The `get_verb` function must have the following header and fulfill the requirements of the following documentation string.

```

def get_verb(quantity, tense):
    """Return a randomly chosen verb. If tense is "past",
    this function will return one of these ten verbs:
        "drank", "ate", "grew", "laughed", "thought",
        "ran", "slept", "talked", "walked", "wrote"
    If tense is "present" and quantity is 1, this
    function will return one of these ten verbs:
        "drinks", "eats", "grows", "laughs", "thinks",
        "runs", "sleeps", "talks", "walks", "writes"
    If tense is "present" and quantity is NOT 1,
    this function will return one of these ten verbs:
        "drink", "eat", "grow", "laugh", "think",
        "run", "sleep", "talk", "walk", "write"
    If tense is "future", this function will return one of
    these ten verbs:
        "will drink", "will eat", "will grow", "will laugh",
        "will think", "will run", "will sleep", "will talk",
        "will walk", "will write"

    Parameters

```

```

    quantity: an integer that determines if the
    returned verb is single or plural.
    tense: a string that determines the verb conjugation,
    either "past", "present" or "future".
    Return: a randomly chosen verb.
"""

```

5. Write the `main` function and any other functions that you think are necessary for your program to generate and print six sentences with these characteristics:

Quantity Verb Tense

- a. single past
- b. single present
- c. single future
- d. plural past
- e. plural present
- f. plural future

6. In a new file named `test_sentences.py`, write three test functions named `test_get_determiner`, `test_get_noun`, and `test_get_verb`. Each of these three test functions must test one of the program functions: `get_determiner`, `get_noun`, and `get_verb`.

Recall that each of the three program functions (`get_determiner`, `get_noun`, and `get_verb`) returns a word that the computer chooses randomly from a list of words. The random nature of the program functions makes the corresponding test functions a little difficult to implement. How do we write code to verify that a word chosen randomly is correct? One possible way is to verify that the chosen word is from the correct list. For example, if we write a call to the `get_determiner` function and pass 1 as the quantity like this:

```
determ = get_determiner(1)
```

Then the word in the `determ` variable should be in this list:

```
["a", "one", "the"]
```

We can use the `assert` keyword and the Python membership operator, which is the keyword `in`, to verify that the word in the `determ` variable is in the list, like this:

```

def test_get_determiner():
    # Call the get_determiner function.
    determ = get_determiner(1)

    # Verify that the word stored in the variable
    # determ is in the list of single determiners.
    words = ["a", "one", "the"]
    assert determ in words

```

The following example contains code for the test_get_determiner function. Copy and paste the code into your test_sentences.py file and use it as an example to help you write test_get_noun and test_get_verb.

```
from sentences import get_determiner, get_noun, get_verb
import random
import pytest

def test_get_determiner():
    # 1. Test the single determiners.

    single_determiners = ["a", "one", "the"]

    # This loop will repeat the statements inside it 4 times.
    # If a loop's counting variable is not used inside the
    # body of the loop, many programmers will use underscore
    # (_) as the variable name for the counting variable.
    for _ in range(4):

        # Call the get_determiner function which
        # should return a single determiner.
        word = get_determiner(1)

        # Verify that the word returned from get_determiner
        # is one of the words in the single_determiners list.
        assert word in single_determiners

    # 2. Test the plural determiners.

    plural_determiners = ["two", "some", "many", "the"]

    # This loop will repeat the statements inside it 4 times.
    for _ in range(4):

        # Call the get_determiner function which
        # should return a plural determiner.
        word = get_determiner(2)

        # Verify that the word returned from get_determiner
        # is one of the words in the plural_determiners list.
        assert word in plural_determiners
```

- At the bottom of your test_sentences.py file, write a call to the pytest.main function like this:

```
# Call the main function that is part of pytest so that the
# computer will execute the test functions in this file.
pytest.main(["-v", "--tb=line", "-rN", __file__])
```

Testing Procedure

Verify that your test program works correctly by following each step in this procedure:

1. Run your `test_sentences.py` program and verify that all three of the test functions pass. If one or more of the tests don't pass, find and fix the mistakes in your program functions or test functions until the tests pass as shown in this output:

```
> python test_sentences.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson05
collected 3 items

test_sentences.py::test_get_determiner PASSED [ 33%]
test_sentences.py::test_get_noun PASSED [ 66%]
test_sentences.py::test_get_verb PASSED [100%]

===== 3 passed in 0.10s =====
```

2. Run your `sentences.py` program and ensure that your program outputs six sentences with the following characteristics:

Quantity	Verb Tense
a. single	past
b. single	present
c. single	future
d. plural	past
e. plural	present
f. plural	future

Your program's output should be similar to the sample run output shown here. However, because your program randomly chooses the determiners, nouns, and verbs, your program will generate different sentences than the ones shown here.

```
> python sentences.py
The cat laughed.
Some girls thought.
One man eats.
Many dogs run.
The woman will think.
Many men will write.
```

Ponder

During this assignment, you wrote four program functions named `main`, `get_determiner`, `get_noun`, and `get_verb`. Also, in a separate file, you wrote three test functions named `test_get_determiner`, `test_get_noun`, and `test_get_verb`. Each of the test functions called one of the program functions and automatically verified that the value returned from the program function was correct. If you worked as a software developer on a large project with four other software developers, how would test functions help you and the other developers write better code?

Submission

On or before the due date, return to I-Learn and report your progress on this milestone.

06 Prepare: Troubleshooting Functions

What should you do when your program isn't working? If your program is small, you could examine each line of the program. However, if your program is large or contains multiple functions, there are better ways to find and fix the problems, including writing and running test functions, writing print statements, and using a debugger.

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

Types of Mistakes

Broadly speaking, there are two types of mistakes or errors that a programmer might make when writing a program: syntax errors and logic errors. A **syntax error** is a mistake made by a programmer that violates the rules of a programming language such as misspelling a keyword, forgetting to type a closing parenthesis, or forgetting to type a colon (:) at the end of an `if` statement. A syntax error will cause the computer to terminate a Python program and print an error message to the terminal window. A **logic error** is a mistake made by a programmer that causes the computer to produce the wrong results. Often, a logic error will not cause the computer to terminate a program or to print an error message. It will simply cause the computer to produce incorrect results.

Error Messages

Regardless of the type of error (syntax or logic), if the computer prints an error message while executing a program, the first thing a programmer should do is read and understand the error message. Example 1 shows a simple Python program that contains a syntax error. The message that the computer printed because of the syntax error is shown below example 1.

```
1 # Example 1
2
3 def main():
4     print("Are you surprised, Clark?")
5
6 # Start this program by
7 # calling the main function.
8 if __name__ == "__main__":
9     main()
```

```
> python surprise.py
  File "C:\Users\cse111\lesson06\surprise.py", line 4
    print("Are you surprised, Clark?")
                           ^
SyntaxError: EOL while scanning string literal
```

From the error message, we read that example 1 contains a syntax error and that the error is on line 4 of the program where there is something wrong with the string. By examining the error message and the program at line 4, we learn that the programmer forgot to type the closing double quote at the end of the string. By the way, **EOL** that appears in the error message is an acronym for "end of line." You might also see the acronyms **EOF** and **EOT** which mean "end of file" and "end of transmission."

If the computer prints an error message that you don't understand, you can search the internet for its meaning. Simply copy and paste the error message into the search bar of your browser. Here are two of the search results from Google for the error message "SyntaxError: EOL while scanning string literal."

About 27,500 results (0.49 seconds)

[SyntaxError- EOL while scanning string literal](#)

SyntaxError: EOL while scanning string literal "EOL" stands for "end of line". An EOL error means that Python hit the end of a line while going through a string.

[Syntax Error: EOL while scanning string literal - AskPython](#)

EOL stands for "End of Line". The error means that the Python Interpreter reached the end of the line when it tried to scan the string literal. The string literals (constants) must be enclosed in single and double quotation marks.

Print Statements

If the computer doesn't print an error message, but your program is producing incorrect results, you could add print statements to your program in strategic locations to help you find the mistakes. These print statements should print the value of the variables in your program so that you can examine the values to ensure they are correct. Example 2 contains a program with a complex calculation for computing the remaining balance of a loan. Notice the print statements at lines 28, 32, and 35 and also at lines 43, 47, and 50. The programmer wrote these print statements at important points in the program, specifically near the beginning of each function and in the middle of the calculations.

```
1 # Example 2
2
3 def main():
4     print("This program computes and prints the remaining")
5     print("balance for a loan with a fixed annual percentage")
6     print("rate and a fixed number of payments per year.")
7     print()
8     print("Please enter the following five values.")
```

```

9
10 principal = float(input("Principal amount: "))
11 annual_rate = float(input("Annual percentage rate: "))
12 years = int(input("Number of years in the life of the loan: "))
13 payments_per_year = int(input("Number of payments per year: "))
14 number_paid = int(input("Number of payments already paid: "))
15
16 balance = compute_balance(principal, annual_rate, years,
17                           payments_per_year, number_paid)
18
19 print()
20 print(f"Balance remaining: {balance}")
21
22
23 def compute_balance(princ, ar, years, ppy, ptd):
24     """Compute and return the balance remaining for a loan."""
25     payment = compute_payment(princ, ar, years, ppy)
26
27     print()
28     print(f"compute_balance({princ}, {ar}, {years}, {ppy}, {ptd})")
29
30     rate = ar / ppy
31     power = (1 + rate) ** ptd
32     print(f"    payment: {payment}  rate: {rate}  power: {power}")
33
34     balance = princ * power - payment * (power - 1) / rate
35     print(f"    balance: {balance:.2f}")
36
37     return round(balance, 2)
38
39
40 def compute_payment(princ, ar, years, ppy):
41     """Compute and return the payment per period for a loan."""
42     print()
43     print(f"compute_payment({princ}, {ar}, {years}, {ppy})")
44
45     rate = ar / ppy
46     n = years * ppy
47     print(f"    rate: {rate}  n: {n}")
48
49     payment = princ * rate / (1 - (1 + rate) ** -n)
50     print(f"    payment: {payment:.2f}")
51
52     return round(payment, 2)
53
54
55 # Start this program by
56 # calling the main function.
57 if __name__ == "__main__":
58     main()

```

```
> python balance.py
This program computes and prints the remaining
balance for a loan with a fixed annual percentage
rate and a fixed number of payments per year.

Please enter the following five values.
Principal amount: 80000
Annual percentage rate: 0.06
Number of years in the life of the loan: 15
Number of payments per year: 12
Number of payments already paid: 45

compute_payment(80000.0, 0.06, 15, 12)
    rate: 0.005  n: 180
    payment: 675.09

compute_balance(80000.0, 0.06, 15, 12, 45)
    payment: 675.09  rate: 0.005  power: 1.2516208207696773
    balance: 66156.33

Balance remaining: 66156.33
```

Although print statements are simple to understand and add to a program and often helpful, in many situations they are not the most effective way to find logic errors.

Test Functions

Many programmers underestimate the effectiveness of writing and running test functions to find logic errors in a program. Many programmers will write a complete program with multiple functions and never pause to test any part of it. Instead, they will write the entire program and then do something similar to the following steps to test it.

1. Run the program and enter input like a user would.
2. Examine the program's output and discover that the output is incorrect.
3. Review all the program's code, make some small changes, and add print statements.
4. Repeat steps 1–3 again and again and again, spending lots of time trying to find and fix the errors.

This method for finding and fixing mistakes is time consuming because the programmer is trying to test the whole program at once. Instead, the programmer should test each individual function by writing and running test functions as explained in the [previous lesson](#).

Using a Debugger

A **debugger** is a software development tool that allows a programmer to watch the computer execute the statements in a program. While using a debugger, a programmer can examine the values stored in a program's variables as the computer executes each

line of the program. A debugger is a very effective tool for finding mistakes in a program. Nearly all programming languages and environments include a debugger. Professional software developers use a debugger nearly every single work day. Most companies will not use a programming language unless that language includes a debugger.

The Python programming language includes a debugger that you can use while writing a program in VS Code. It is much easier to learn how to use a debugger by watching someone use it than by reading about a debugger. Watch the following video that shows a BYU-Idaho faculty member using a debugger in VS Code to find mistakes in a program.

How to Use the [Python Debugger in VS Code](#) (18 minutes)

A debugger is also a great learning tool. Using a debugger to step through the statements of your program one at a time and examining the values of the variables after each step will show you exactly how Python works. Try it! If you don't completely understand `if-elif-else` statements, `while` loops, `for` loops, passing parameters, or returning a value from a function, then put one or more breakpoints in a program that contains those elements, start the program in the debugger, and step through it one line at a time. After the computer executes each statement, examine the values of the variables and predict in your mind how the next statement will change the values of the variables.

Summary

During this lesson, you are learning the difference between a syntax error and a logic error. A syntax error is a mistake made by a programmer that violates the rules of a programming language and prevents a program from running. Usually, you must fix all syntax errors before your program will run. After you fix all syntax errors, your program will run but may contain logic errors. A logic error is a mistake made by a programmer that causes the computer to produce the wrong results. Some tools that you can use to find and fix the logic errors in your programs are test functions, print statements, and a debugger.

06 Checkpoint: Troubleshooting Functions

Purpose

Practice using the Python debugger in VS Code.

Assignment

Do the following:

1. Download and save the [example.py](#) file.
2. Open example.py in VS Code.
3. Follow along with this [video about the Python debugger](#) (17 minutes) from the prepare content and use the debugger in VS Code to step through the example.py program.

Ponder

As you used the debugger to step through the example.py program, did you learn anything new about Python and how functions work? Do you think the debugger could help you find mistakes in your code?

Submission

When complete, report your progress in the associated I-Learn quiz.

06 Team Activity: Troubleshooting Functions

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Purpose

Write a program with several functions. Use a debugger while writing your program or after writing it to step through your code.

Problem Statement

The Rosenberg self-esteem scale is a self-esteem measure developed by the sociologist Morris Rosenberg in 1965. It is still used in social-science research today. To complete the measure, a person completes a survey that contains the following ten statements.

1. I feel that I am a person of worth, at least on an equal plane with others.
2. I feel that I have a number of good qualities.
3. All in all, I am inclined to feel that I am a failure.
4. I am able to do things as well as most other people.
5. I feel I do not have much to be proud of.
6. I take a positive attitude toward myself.
7. On the whole, I am satisfied with myself.
8. I wish I could have more respect for myself.
9. I certainly feel useless at times.
10. At times I think I am no good at all.

The person responds to each statement by choosing one of these four options: strongly disagree, disagree, agree, or strongly agree. The person's response to each statement is worth 0–3 points, meaning the highest possible score is $10 * 3 = 30$ points. If a person scores lower than 15 points, the person may have a problematic low self-esteem.

Notice that five of the statements (numbers 1, 2, 4, 6, 7) are positive and are scored like this:

Choice	Points
Strongly Disagree	0
Disagree	1
Agree	2

Choice	Points
Strongly Agree	3

The other five statements (numbers 3, 5, 8, 9, 10) are negative and are scored like this:

Choice	Points
Strongly Disagree	3
Disagree	2
Agree	1
Strongly Agree	0

Assignment

As a team, write a Python program named `esteem.py` that implements the Rosenberg self-esteem scale. Your program must ask the user to respond to each of the ten statements with D, d, a, or A which mean strongly disagree, disagree, agree, and strongly agree. Your program must compute the score for each answer and sum and display the person's total score. You should think about how you will separate this program into functions before you begin writing the program.

Core Requirements

1. Your program prints the introductory text as shown in the Testing Procedure section below.
2. Your program prints each of the ten statements and gets a response from the user.
3. Your program computes the score for each response and sums all the scores and displays the total score.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Write a program that implements the [Nature Relatedness Scale](#)

Helpful Documentation

This [video about the Python debugger](#) (17 minutes) from the prepare content for this lesson shows how to use the debugger in VS Code to find mistakes in a program.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Use the debugger in VS Code to step through your program and verify that it works correctly.
2. Run your `esteem.py` program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python esteem.py
This program is an implementation of the Rosenberg
Self-Esteem Scale. This program will show you ten
statements that you could possibly apply to yourself.
Please rate how much you agree with each of the
statements by responding with one of these four letters:

D means you strongly disagree with the statement.
d means you disagree with the statement.
a means you agree with the statement.
A means you strongly agree with the statement.

1. I feel that I am a person of worth, at least on an
equal plane with others.
Enter D, d, a, or A: 
2. I feel that I have a number of good qualities.
Enter D, d, a, or A: 
3. All in all, I am inclined to feel that I am a failure.
Enter D, d, a, or A: 
4. I am able to do things as well as most other people.
Enter D, d, a, or A: 
5. I feel I do not have much to be proud of.
Enter D, d, a, or A: 
6. I take a positive attitude toward myself.
Enter D, d, a, or A: 
7. On the whole, I am satisfied with myself.
Enter D, d, a, or A: 
8. I wish I could have more respect for myself.
Enter D, d, a, or A: 
9. I certainly feel useless at times.
Enter D, d, a, or A: 
10. At times I think I am no good at all.
Enter D, d, a, or A: 
```

Your score is 21.
A score below 15 may indicate problematic low self-esteem.

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#). Please **do not look at the sample solution** until you have either finished the program or diligently worked for at

least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Ponder

Did you use the debugger to step through your program? What is the difference between the "Step Over" () and "Step Into" () buttons?

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson06/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 NEGATIVE = -1
4 POSITIVE = 1
5
6
7 def main():
8     print("This program is an implementaiton of the Rosenberg")
9     print("Self-Esteem Scale. This program will show you ten")
10    print("statements that you could possibly apply to yourself.")
11    print("Please rate how much you agree with each of the")
12    print("statements by responding with one of these four letters:")
13    print()
14    print("D means you strongly disagree with the statement.")
15    print("d means you disagree with the statement.")
16    print("a means you agree with the statement.")
17    print("A means you strongly agree with the statement.")
18    print()
19
20    score = 0
21    score += ask_question(
22        "1. I feel that I am a person of worth,"
23        " at least on an equal plane with others.", POSITIVE)
24    score += ask_question(
25        "2. I feel that I have a number of good qualities.", POSITIVE)
26    score += ask_question(
27        "3. All in all, I am inclined to feel that I am a failure.",
28        NEGATIVE)
29    score += ask_question(
30        "4. I am able to do things as well as most other people.",
31        POSITIVE)
32    score += ask_question(
33        "5. I feel I do not have much to be proud of.", NEGATIVE)
34    score += ask_question(
35        "6. I take a positive attitude toward myself.", POSITIVE)
36    score += ask_question(
37        "7. On the whole, I am satisfied with myself.", POSITIVE)
38    score += ask_question(
39        "8. I wish I could have more respect for myself.", NEGATIVE)
40    score += ask_question(
41        "9. I certainly feel useless at times.", NEGATIVE)
42    score += ask_question(
43        "10. At times I think I am no good at all.", NEGATIVE)
44
45    print()
46    print(f"Your score is {score}.")
47    print("A score below 15 may indicate problematic low self-esteem.")
48
49
50 def ask_question(statement, pos_or_neg):
51     """Display one statement to the user and get the user's response.
52     Then determine the score for the response and return the score.
53
54     Parameters
55         statement: The statement to show the user.
56         pos_or_neg: Either the constant POSITIVE or NEGATIVE.
57     Return: the score from the user's response to the statement.
```

```
58     """
59     print(statement)
60     answer = input("Enter D, d, a, or A: ")
61     score = 0
62     if answer == 'D':
63         score = 0
64     elif answer == 'd':
65         score = 1
66     elif answer == 'a':
67         score = 2
68     elif answer == 'A':
69         score = 3
70
71     if pos_or_neg == NEGATIVE:
72         score = 3 - score
73
74     return score
75
76
77 # If this file was executed like this:
78 # > python esteem.py
79 # then call the main function. However, if this file
80 # was simply imported, then skip the call to main.
81 if __name__ == "__main__":
82     main()
```

06 Prove Assignment: Troubleshooting Functions

Purpose

Prove that you can write a Python program and write and run test functions to help you find and fix mistakes.

Problem Statement

The **Turing test**, named after Alan Turing, is a test of a computer's ability to make conversation that is indistinguishable from human conversation. A computer that could pass the Turing test would need to understand sentences typed by a human and respond with sentences that make sense.

In English, a **preposition** is a word used to express spatial or temporal relations, such as "in", "over", and "before". A **prepositional phrase** is group of words that begins with a preposition and includes a noun. For example:

above the water
in the kitchen
after the meeting

Assignment

Write the second half of the Python program that you began in the previous lesson's [prove milestone](#), a program that generates simple English sentences. During this prove assignment, you will write and test functions that generate sentences with four parts:

1. a determiner
2. a noun
3. a verb
4. a prepositional phrase

For example:

One girl talked for the car.
A bird drinks off one child.
The child will run on the car.
Some dogs drank above many rabbits.
Some children laugh at many dogs.
Some rabbits will talk about some cats.

To complete this prove assignment, your program must include at least these six functions:

1. main
2. get_determiner
3. get_noun
4. get_verb
5. get_preposition
6. get_prepositional_phrase

You may add other functions if you find them helpful. The `get_preposition` function must randomly choose a preposition from a list and return the randomly chosen preposition. The `get_prepositional_phrase` function must make a prepositional phrase by calling the `get_preposition`, `get_determiner`, and `get_noun` functions.

In addition, to complete this prove assignment, you must write and submit a file named `test_senetences.py` that contains five functions named as follows:

1. `test_get_determiner`
2. `test_get_noun`
3. `test_get_verb`
4. `test_get_preposition`
5. `test_get_prepositional_phrase`

Helpful Documentation

The [prepare content](#) for this lesson explains how to troubleshoot functions and entire programs that are not working correctly.

The string [split method](#) separates a string of text into words and returns a list of strings.

Steps

Do the following:

1. Use the `get_determiner` function from the previous lesson's prove milestone as an example to help you write the `get_preposition` function. The `get_preposition` function must have the following header and fulfill the requirements of the following documentation string.

```
def get_preposition():
    """Return a randomly chosen preposition
    from this list of prepositions:
        "about", "above", "across", "after", "along",
        "around", "at", "before", "behind", "below",
```

```
"beyond", "by", "despite", "except", "for",
"from", "in", "into", "near", "of",
"off", "on", "onto", "out", "over",
"past", "to", "under", "with", "without"
```

```
Return: a randomly chosen preposition.  
"""
```

2. Write the `get_prepositional_phrase` function to have the following header and fulfill the requirements of the following documentation string.

```
def get_prepositional_phrase(quantity):
    """Build and return a prepositional phrase composed of three
    words: a preposition, a determiner, and a noun by calling the
    get_preposition, get_determiner, and get_noun functions.

    Parameter
    quantity: an integer that determines if the determiner
               and noun in the prepositional phrase returned from
               this function are single or plural.

    Return: a prepositional phrase.
    """
```

3. Add code to the `main` function and write any other functions that you think are necessary for your program to generate and print six sentences, each with a determiner, a noun, a verb, and a prepositional phrase. The six sentences must have the following characteristics:

Quantity	Verb Tense
a. single	past
b. single	present
c. single	future
d. plural	past
e. plural	present
f. plural	future

- | | |
|-----------|---------|
| a. single | past |
| b. single | present |
| c. single | future |
| d. plural | past |
| e. plural | present |
| f. plural | future |

4. In the `test_sentences.py` file that you wrote during the previous lesson's prove milestone, write two functions named `test_get_preposition` and `test_get_prepositional_phrase` that test the `get_preposition` and `get_prepositional_phrase` functions.

Perhaps you are wondering what code you should write in the `test_get_prepositional_phrase` function. To answer that question, ask yourself, "What do we know about the `get_prepositional_phrase` function?" From its description, we know the `get_prepositional_phrase` function returns a string made of three words: a preposition, a determiner, and a noun. So you could write code in the `test_get_prepositional_phrase` function that calls the `get_prepositional_phrase` function and then calls the Python string [split method](#) to separate the returned phrase into a list of words and verifies that the length of the list is three. In addition for each word in the list, you could write an `assert` statement that verifies that each word is the correct English part of speech.

Testing Procedure

Verify that your test program works correctly by following each step in this procedure:

1. Run your `test_sentences.py` program and verify that all five of the test functions pass. If one or more of the tests don't pass, find and fix the mistakes in your program functions or test functions until the tests pass as shown in this output:

```
> python test_sentences.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson06
collected 5 items

test_sentences.py::test_get_determiner PASSED [ 20%]
test_sentences.py::test_get_noun PASSED [ 40%]
test_sentences.py::test_get_verb PASSED [ 60%]
test_sentences.py::test_get_preposition PASSED [ 80%]
test_sentences.py::test_get_prepositional_phrase PASSED [100%]

===== 5 passed in 0.10s =====
```

2. Run your `sentences.py` program and ensure that your program's output is similar to the sample run output shown here. Because your program randomly chooses the determiners, nouns, verbs, and prepositions, your program will generate different sentences than the ones shown here.

```
> python sentences.py
One girl talked for the car.
Some dogs drank above many rabbits.
One bird drinks off one child.
Some children laugh at many dogs.
The child will run on the car.
Some rabbits will talk about some cats.
```

Exceeding the Requirements

If your program fulfills the requirements for this assignment as described in the previous prove milestone and the Assignment section above, your program will earn 93% of the possible points. In order to earn the remaining 7% of points, you will need to add one or more features to your program so that it exceeds the requirements. Here are a few suggestions for additional features that you could add to your program if you wish.

- Within your `main` function add another call to `get_prepositional_phrase` so that each sentence includes two prepositional phrases like this:

One girl across one cat talked for the car.
A bird near the rabbit drinks off one child.
The child under the cat will run on the car.

Some dogs without a cat drank above many rabbits.
Some children from a bird laugh at many dogs.
Some rabbits behind one man will talk about some cats.

- Write a function named `get_adjective` and call it in your `main` function to add an adjective to the sentences produced by your program. Does it make sense to call `get_adjective` in your `get_prepositional_phrase` function?
- In `test_sentences.py` write a function named `test_get_adjective` that tests the `get_adjective` function.
- Write a function named `get_adverb` and call it in your `main` function to add an adverb to the sentences produced by your program.
- In `test_sentences.py` write a function named `test_get_adverb` that tests the `get_adverb` function.

Ponder

How hard would it be to modify your program to pass the Turing test?

Submission

To submit your program, return to I-Learn and do these two things:

1. Upload your `sentences.py` and `test_sentences.py` files for feedback.
2. Add a submission comment that specifies the grading category that best describes your program along with a one or two sentence justification for your choice. The grading criteria are:
 - a. Some attempt made
 - b. Developing but significantly deficient
 - c. Slightly deficient
 - d. Meets requirements
 - e. Exceeds requirements

07 Prepare: Lists and Repetition

During this lesson, you will learn how to store many elements in a Python list. You will learn how to write a loop that processes each element in a list. Also, you will learn that lists are passed into a function differently than numbers are passed.

Videos

Watch these videos from Microsoft about lists and repetition in Python.

[Lists](#) (12 minutes)

[Loops](#) (6 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

Lists

A Python program can store many values in a **list**. Lists are mutable, meaning they can be changed after they are created. Each value in a list is called an **element** and is stored at a unique index. An **index** is always an integer and determines where an element is stored in a list. The first index of a Python list is always zero (0). The following diagram shows a list that contains five strings. The diagram shows both the elements and the indexes of the list. Notice that each index is a unique integer, and that the first index is zero.

elements	"yellow"	"red"	"green"	"yellow"	"blue"
indexes	[0]	[1]	[2]	[3]	[4]
← list length is 5 →					

In a Python program, we can create a list by using square brackets ([and]). We can determine the number of items in a list by calling the built-in `len` function. We can retrieve an item from a list and replace an item in a list using square brackets ([and]) and an index. Example 1 contains a program that creates a list, prints the length of the list, retrieves and prints one item from the list, changes one item in the list, and then prints the entire list.

```

# Example 1

def main():
    # Create a list that contains five strings.
    colors = ["yellow", "red", "green", "yellow", "blue"]

    # Call the built-in len function
    # and print the length of the list.
    length = len(colors)
    print(f"Number of elements: {length}")

    # Print the element that is stored
    # at index 2 in the colors list.
    print(colors[2])

    # Change the element that is stored at
    # index 3 from "yellow" to "purple".
    colors[3] = "purple"

    # Print the entire colors list.
    print(colors)

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_1.py
Number of elements: 5
green
['yellow', 'red', 'green', 'purple', 'blue']

```

We can add an item to a list by using the `insert` and `append` methods. We can determine if an element is in a list by using the Python membership operator, which is the keyword `in`. We can find the index of an item within a list by using the `index` method. We can remove an item from a list by using the `pop` and `remove` methods. Example 2 shows how to create a list and add, find, and remove items from a list.

```

# Example 2

def main():
    # Create an empty list that will hold fabric names.
    fabrics = []

    # Add three elements at the end of the fabrics list.
    fabrics.append("velvet")
    fabrics.append("denim")
    fabrics.append("gingham")

    # Insert an element at the beginning of the fabrics list.
    fabrics.insert(0, "chiffon")
    print(fabrics)

    # Determine if gingham is in the fabrics list.
    if "gingham" in fabrics:
        print("gingham is in the list.")
    else:
        print("gingham is NOT in the list.")

```

```

# Get the index where velvet is stored in the fabrics list.
i = fabrics.index("velvet")

# Replace velvet with taffeta.
fabrics[i] = "taffeta"

# Remove the last element from the fabrics list.
fabrics.pop()

# Remove denim from the fabrics list.
fabrics.remove("denim")

# Get the length of the fabrics list and print it.
n = len(fabrics)
print(f"The fabrics list contains {n} elements.")
print(fabrics)

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_2.py
['chiffon', 'velvet', 'denim', 'gingham']
gingham is in the list.
The fabrics list contains 2 elements.
['chiffon', 'taffeta']

```

The lists in examples 1 and 2 store strings. Of course, it is possible to store numbers in a list, too. In fact, Python allows a program to store any data type in a list, including other lists.

Repetition

We can cause a computer to repeat a group of statements by writing `for` and `while` loops.

`for`

A `for` loop iterates over a range of numbers, such as `range(3, 10)` or a sequence, such as a list. Example 3 shows three `for` loops that iterate over a range of numbers. Notice that just like `if` statements in Python, the body of a loop starts and ends with indentation.

```

# Example 3

def main():
    # Count from zero to nine by one.
    for i in range(10):
        print(i)
    print()

    # Count from zero to eight by two.
    for i in range(0, 10, 2):
        print(i)

```

```

print()

# Count from 100 down to 70 by three.
for i in range(100, 69, -3):
    print(i)

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_3.py
0
1
2
:
8
9

0
2
4
6
8

100
97
94
:
73
70

```

In the next example at lines 8–9 and lines 15–17, there are two for loops. Both loops print each element a list named *colors*. The first loop iterates over the elements in the *colors* list. The second loop iterates over the indexes of the *colors* list. Which style of for loop do you prefer to read and write? Most programmers prefer to write a loop like the one at lines 8–9 because it is simpler than the one at lines 15–17.

```

1  # Example 4
2
3 def main():
4     # Create a list of color names.
5     colors = ["red", "orange", "yellow", "green", "blue"]
6
7     # Use a for loop to print each element in the list.
8     for color in colors:
9         print(color)
10
11     print()
12
13     # Use a different for loop to
14     # print each element in the list.
15     for i in range(len(colors)):
16         color = colors[i]
17         print(color)
18
19     # Of course, the code in the body of a loop can do
20     # much more with each element than simply print it.

```

```
21
22
23 # Call main to start this program.
24 if __name__ == "__main__":
25     main()
```

```
> python example_4.py
red
orange
yellow
green
blue

red
orange
yellow
green
blue
```

In the previous example, the code in the body of both `for` loops is very short and simply prints one element from the list each time through the loop. However, you can write as many lines of code as you need in the body of a loop to repeatedly perform all sorts of computations for each element in a list.

break

A `break` statement causes a loop to end early. In example 5 at [lines 8–12](#), there is a `for` loop that asks the user to input ten numbers one at a time. However, the loop will terminate early if the user enters a zero (0) because of the `if` statement and `break` statement at [lines 10 and 11](#).

```
1 # Example 5
2
3 def main():
4     sum = 0
5
6     # Get ten or fewer numbers from the user and
7     # add them together.
8     for i in range(10):
9         number = float(input("Please enter a number: "))
10        if number == 0:
11            break
12        sum += number
13
14    # Print the sum of the numbers for the user to see.
15    print(f"sum: {sum}")
16
17
18 # Call main to start this program.
19 if __name__ == "__main__":
20     main()
```

```
> python example_5.py
Please enter a number: 6
Please enter a number: 4
Please enter a number: -2
Please enter a number: 0
sum: 8.0
```

while

A while loop is more flexible than a for loop and repeats while some condition is true. Imagine that we need a function to compare the contents of two lists? Can we use a loop to compare the contents of two lists? Example 6 contains a while loop at [lines 35–46](#) with an if statement at [line 42](#) that finds the first index where two lists differ.

```
1 # Example 6
2
3 def main():
4     list1 = ["red", "orange", "yellow", "green", "blue"]
5     list2 = ["red", "orange", "green", "green", "blue"]
6
7     index = compare_lists(list1, list2)
8     if index == -1:
9         print("The contents of list1 and list2 are equal")
10    else:
11        print(f"list1 and list2 differ at index {index}")
12
13
14 def compare_lists(list1, list2):
15     """Compare the contents of two lists. If the contents
16     of the two lists are not equal, return the index of
17     the first difference. If the contents of the two lists
18     are equal, return -1.
19
20     Parameters
21         list1: a list
22         list2: another list
23     Return: an index or -1
24     """
25
26     # Get the length of the shortest list.
27     length1 = len(list1)
28     length2 = len(list2)
29     limit = min(length1, length2)
30
31     # Begin at the first index (0) and repeat until the
32     # computer finds two elements that are not equal or
33     # until the computer reaches the end of the shortest
34     # list, whichever comes first.
35     i = 0
36     while i < limit:
37         # Retrieve one element from each list.
38         element1 = list1[i]
39         element2 = list2[i]
40
41         # If the two elements are not
42         # equal, quit the while loop.
43         if element1 != element2:
44             break
```

```

45         # Add one to the index variable.
46         i += 1
47
48     # If the length of both lists are equal and the
49     # computer verified that all elements are equal,
50     # set i to -1 to indicate that the contents of
51     # the two lists are equal.
52     if length1 == length2 == i:
53         i = -1
54
55     return i
56
57
58 # Call main to start this program.
59 if __name__ == "__main__":
60     main()

```

```

> python example_6.py
list1 and list2 differ at index 2

```

Compound Lists

A **compound list** is a list that contains other lists. Compound lists are used to store lots of related data. Example 7 shows how to create a compound list, retrieve one inner list from the compound list, and retrieve an individual number from the inner list.

```

# Example 7

def main():
    # These are the indexes of each
    # element in the inner lists.
    YEAR_PLANTED_INDEX = 0
    HEIGHT_INDEX = 1
    GIRTH_INDEX = 2
    FRUIT_AMOUNT_INDEX = 3

    # Create a compound list that stores inner lists.
    apple_tree_data = [
        # [year_planted, height, girth, fruit_amount]
        [2012, 2.7, 3.6, 70.5],
        [2012, 2.4, 3.7, 81.3],
        [2015, 2.3, 3.6, 62.7],
        [2016, 2.1, 2.7, 42.1]
    ]

    # Retrieve one inner list from the compound list.
    one_tree = apple_tree_data[2]

    # Retrieve one value from the inner list.
    height = one_tree[HEIGHT_INDEX]

    # Print the tree's height.
    print(f"height: {height}")

# Call main to start this program.

```

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

```
> python example_7.py
height: 2.3
```

Example 8 shows how to process all elements in a compound list. The for loop at line 24 causes the computer to repeat lines 24–34 once for each inner list that is inside the compound list named `apple_tree_data`. Line 28 retrieves the fruit amount from one inner list and then line 34 adds one fruit amount to the total fruit amount.

```
1 # Example 8
2
3 def main():
4     # These are the indexes of each
5     # element in the inner lists.
6     YEAR_PLANTED_INDEX = 0
7     HEIGHT_INDEX = 1
8     GIRTH_INDEX = 2
9     FRUIT_AMOUNT_INDEX = 3
10
11    # Create a compound list that stores inner lists.
12    apple_tree_data = [
13        # [year_planted, height, girth, fruit_amount]
14        [2012, 2.7, 3.6, 70.5],
15        [2012, 2.4, 3.7, 81.3],
16        [2015, 2.3, 3.6, 62.7],
17        [2016, 2.1, 2.7, 42.1]
18    ]
19
20    total_fruit_amount = 0
21
22    # This loop will repeat once for each inner list
23    # in the apple_tree_data compound list.
24    for inner_list in apple_tree_data:
25
26        # Retrieve the fruit amount from
27        # the current inner list.
28        fruit_amount = inner_list[FRUIT_AMOUNT_INDEX]
29
30        # Print the fruit amount for the current tree.
31        print(fruit_amount)
32
33        # Add the current fruit amount to the total.
34        total_fruit_amount += fruit_amount
35
36    # Print the total fruit amount.
37    print(f"Total fruit amount: {total_fruit_amount:.1f}")
38
39
40 # Call main to start this program.
41 if __name__ == "__main__":
42     main()
```

```
> python example_8.py
70.5
81.3
62.7
42.1
Total fruit amount: 256.6
```

Values and References

In a Python program, the computer assigns values to variables differently based on their data type. Consider the small program in example 9 and the output of that program. The program in example 9 contains two integer variables named *x* and *y*. The program in example 9 does the following:

- The statement at [line 4](#) stores the value 17 into the variable *x*.
- [Line 5](#) copies the value that is in the variable *x* into the variable *y*.
- [Line 6](#) prints the values of *x* and *y* which are both 17.
- [Line 7](#) adds one to the value of *x*, making its value 18 instead of 17.
- [Line 8](#) prints the values of *x* and *y* again. The value of *x* was changed to 18. The value of *y* remained unchanged.

Why does [line 7](#) (*x* += 1) change the value of *x* but not change the value of *y*? Because [line 5](#) copies *the value* that was in *x* into *y*. In other words, *x* and *y* are two separate variables, each with its own value.

```
1 # Example 9
2
3 def main():
4     x = 17
5     y = x
6     print(f"Before changing x: {x} {y}")
7     x += 1
8     print(f"After changing x: {x} {y}")
9
10 # Call main to start this program.
11 if __name__ == "__main__":
12     main()
```

```
> python example_9.py
Before changing x: 17 17
After changing x: 18 17
```

Example 10 shows a small Python program that contains two variables named *lx* and *ly* that each refer to a list. This program is similar to the previous program, but it has two lists instead of two integers. From the output of example 10, we see there is a big difference between the way a Python program assigns integers and the way it assigns lists. The program in example 10 does the following:

- The statement at [line 4](#) creates a list and stores a reference to that list in the variable *lx*.

- Line 5 copies the reference in the variable `lx` into the variable `ly`. Line 5 does not create a copy of the list but instead causes both the variables `lx` and `ly` to refer to the same list.
- Line 6 prints the values of `lx` and `ly`. Notice that their values are the same as we expect them to be because of line 5.
- Line 7 appends the number 5 onto the list `lx`.
- Line 8 prints the values of `lx` and `ly` again. Notice in the output that when `lx` and `ly` are printed the second time, it appears that the number 5 was appended to both lists.

Why does it appear that appending the number 5 onto `lx` also appends the number 5 onto `ly`? Because `lx` and `ly` refer to the same list. There is really only one list with two references to that list. Because `lx` and `ly` refer to the same list, a change to the list through variable `lx` can be seen through variable `ly`.

```

1 # Example 10
2
3 def main():
4     lx = [7, -2]
5     ly = lx
6     print(f"Before changing lx: lx {lx} ly {ly}")
7     lx.append(5)
8     print(f"After changing lx: lx {lx} ly {ly}")
9
10 # Call main to start this program.
11 if __name__ == "__main__":
12     main()

```

```

> python example_10.py
Before changing lx: lx [7, -2] ly [7, -2]
After changing lx: lx [7, -2, 5] ly [7, -2, 5]

```

From examples 9 and 10, we learn that when a computer executes a Python statement to assign the value of a boolean, integer, or float variable to another variable (`y = x`), the computer copies *the value* of one variable into the other. However, when a computer executes a Python statement to assign the value of a list variable to another variable (`ly = lx`), the computer does not copy *the value* but instead copies *the reference* so that both variables refer to the same list in memory.

Pass by Value and Pass by Reference

The fact that the computer copies the value of some data types (boolean, integer, float) and copies the reference for other data types (list and other large data types) has important implications for passing arguments into functions. Consider the Python program in example 11 with two functions named `main` and `modify_args`. The program in example 11 does the following:

- The statement at line 5 assigns the value 5 to a variable named `x`.

- Line 6 assigns a list to a variable named *lx*.
- Line 7 prints the values of *x* and *lx* before they are passed to the *modify_args* function.
- Line 11 calls the *modify_args* function and passes *x* and *lx* to that function.
- Within the *modify_args* function, line 28 changes the value of the parameter *n* by adding one to it, and line 29 changes the value of the parameter *alist* by appending the number 4 onto it.
- Line 13 prints the values of *x* and *lx* after they were passed to the *modify_args* function. Notice in the output below that the value of *x* was not changed by the *modify_args* function. However, the value of *lx* was changed by the *modify_args* function.

```

1 # Example 11
2
3 def main():
4     print("main()")
5     x = 5
6     lx = [7, -2]
7     print(f"Before calling modify_args(): x {x} lx {lx}")
8
9     # Pass one integer and one list
10    # to the modify_args function.
11    modify_args(x, lx)
12
13    print(f"After calling modify_args(): x {x} lx {lx}")
14
15
16 def modify_args(n, alist):
17     """Demonstrate that the computer passes a value
18     for integers and passes a reference for lists.
19     Parameters
20         n: A number
21         alist: A list
22     Return: nothing
23     """
24     print("    modify_args(n, alist)")
25     print(f"    Before changing n and alist: n {n} alist {alist}")
26
27     # Change the values of both parameters.
28     n += 1
29     alist.append(4)
30
31     print(f"    After changing n and alist: n {n} alist {alist}")
32
33
34 # Call main to start this program.
35 if __name__ == "__main__":
36     main()

```

```

> python example_11.py
main()
Before calling modify_args(): x 5 lx [7, -2]
    modify_args(n, alist)
    Before changing n and alist: n 5 alist [7, -2]
    After changing n and alist: n 6 alist [7, -2, 4]
After calling modify_args(): x 5 lx [7, -2, 4]

```

From the output of example 11, we see that modifying an integer parameter changes the integer within the called function only. However, modifying a list parameter changes the list within the called function and within the calling function. Why? Because when a computer passes a boolean, integer, or float variable to a function, the computer copies *the value* of that variable into the parameter of the called function. Copying the value of an argument into a parameter is known as **pass by value**. However, when a computer passes a list variable to a function, the computer copies *the reference* so that the original variable and the parameter both refer to the same list in memory. Copying the reference of an argument into a parameter is known as **pass by reference**.

Rationale for Pass by Reference

Why are booleans and numbers passed to a function by value and lists are passed to a function by reference? To understand the answer to this question, consider the work a computer would have to do if lists were passed by value.

When a computer passes a number (or boolean) variable to a function, the number is passed by value which means the computer copies the value of the number variable into the parameter of the called function. This works well for numbers because each number variable occupies a small amount of the computer's memory. Making a copy of a number is fast, and the copy uses a small amount of memory.

However, a list may contain millions of elements and therefore occupy a large amount of the computer's memory. If lists were passed by value to a function, the computer would have to make a copy of a list each time it is passed to a function. If a list is large, copying the list takes a relatively long time and uses a lot of the computer's memory for the copy. Therefore, to make programs fast and use less memory, lists (and other large data types) are passed to a function by reference.

Tutorials

If the concepts in this prepare content seem confusing to you, reading these tutorials may help you better understand the concepts.

[Lists in Python](#)

[More on Lists](#)

[Python "for" Loops](#)

[Python "while" Loops](#)

Summary

During this lesson, you are learning how to store many values in a list. Each element in a list is stored at a unique index. Each index is an integer, and the first index in a Python list is always zero (0). The built-in `len` function will return the number of elements stored in a list. The index of the last element in a Python list is always one less than the length of the list. To retrieve or store one element in a list, you can use the square brackets ([and]) and an index. To process all the elements in a list, you can write a `for` loop. A compound list is a list that stores smaller lists.

During this lesson, you are also learning the difference between passing arguments into a function by value and by reference. Numbers are passed into a function by value, meaning that the computer copies the number from an argument into a parameter. Lists are passed into a function by reference, meaning the computer does not make a copy of a list argument but instead passes a reference to the list into a called function. Because lists are passed by reference, if a called function changes a list that is a parameter, that function is not changing a copy of the list but instead is changing the original list from the calling function.

07 Checkpoint: Lists

Purpose

Reinforce your understanding that numbers are passed to a function by value and lists are passed to a function by reference.

Problem Statement

Within a Python program, when a number is passed as an argument to a function, the computer copies the number from the argument into the parameter. In other words, the parameter gets a copy of the value that is in the argument. However, when a list is passed as an argument to a function, the computer does not copy the list. Instead, the computer copies a reference to the list into the parameter. This means the argument and parameter refer to the same list. If a called function changes a list that was passed to the function, this will change the list in both the calling and called functions because both the argument and parameter refer to the same list.

Assignment

During this checkpoint, you will examine and run a Python program that passes both a number and a list to function. Do the following:

1. Download and save the [pass_args.py](#) Python file and then open it in VS Code.
2. Run the `pass_args.py` program on your computer. Examine the code and the output and answer the following three questions:
 - a. In `main`, there are two local variables named `x` and `lx`. At line 14, both of these local variables are passed to the `modify_args` function. After the call to `modify_args` finished, which of `main`'s two local variables did the `modify_args` function change?
 - i. `x`
 - ii. `lx`
 - iii. both of them
 - iv. neither of them
 - b. Why is the `modify_args` function NOT able to change `main`'s local variable `x`?
 - i. Because `x` is a number and therefore passed to a function by value
 - ii. Because `x` is a list and therefore passed to a function by reference

- c. Why is the `modify_args` function able to change `main`'s local variable `lx`?
- i. Because `lx` is a number and therefore passed to a function by value
 - ii. Because `lx` is a list and therefore passed to a function by reference

Sample Run

```
> python pass_args.py
main()
    Before calling modify_args(): x 5  lx [7, -2]
    modify_args(n, alist)
        Before changing n and alist: n 5  alist [7, -2]
        After changing n and alist:  n 6  alist [7, -2, 4]
    After calling modify_args():  x 5  lx [7, -2, 4]
```

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson07/pass_args.py

```
1 """
2 Demonstrate that numbers are passed to a function by value
3 and lists are passed to a function by reference.
4 """
5
6 def main():
7     print("main()")
8     x = 5
9     lx = [7, -2]
10    print(f"    Before calling modify_args(): x {x} lx {lx}")
11
12    # Pass one integer and one list
13    # to the modify_args function.
14    modify_args(x, lx)
15
16    print(f"    After calling modify_args(): x {x} lx {lx}")
17
18
19 def modify_args(n, alist):
20     """Demonstrate that the computer passes a value
21     for integers and passes a reference for lists.
22     Parameters
23         n: A number
24         alist: A list
25     Return: nothing
26     """
27     print("    modify_args(n, alist)")
28     print(f"        Before changing n and alist: n {n} alist {alist}")
29
30     # Change the values of both parameters.
31     n += 1
32     alist.append(4)
33
34     print(f"        After changing n and alist: n {n} alist {alist}")
35
36
37 # If this file was executed like this:
38 # > python teach_solution.py
39 # then call the main function. However, if this file
40 # was simply imported, then skip the call to main.
41 if __name__ == "__main__":
42     main()
```

07 Team Activity: Lists

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Purpose

There are a few details about writing and calling functions in Python that, if you understand, will help you be a more effective programmer. These details include default parameter values and pass by reference. As a team during this activity, you will write and call a function that demonstrates both default parameter values and pass by reference.

Helpful Documentation

- The Python programming language allows a programmer to specify a default value for a function parameter. When a parameter has a default value, the corresponding argument is optional. The prepare content for lesson 4 includes a section about [default parameter values](#).
- Within a Python program, when a number is passed as an argument to a function, the computer copies the number from the argument into the parameter. In other words, the parameter gets a copy of the value that is in the argument. Copying the value of an argument into a parameter is known as **pass by value**.

In Python, when a list is passed as an argument to a function, the computer does *not* copy the list. Instead, the computer copies a reference to the list into the parameter. This means the argument and parameter refer to the same list. Copying a reference from an argument into a parameter is known as **pass by reference**. With pass by reference, if a called function changes a list that was passed to the function, this will of course change the list in both the calling and called functions because both the argument and parameter refer to the same list.

The prepare content for this lesson includes a section about [passing arguments by value and by reference](#).

- The Python `random` module contains functions to generate pseudo random numbers. The [random.uniform function](#) generates random floating-point numbers.
- The Python `random` module contains a [function named choice](#) that randomly chooses one element from a list.

- The built-in Python [round function](#) rounds a number to a specified number of digits after the decimal place.

Assignment

As a team, write a Python program named `random_numbers.py` that creates a list of numbers, appends more numbers onto the list, and prints the list. The program must have two functions named `main` and `append_random_numbers` as follows:

1. `main`
 - a. Has no parameters
 - b. Creates a list named *numbers* like this:

numbers = [16.2, 75.1, 52.3]
 - c. Prints the *numbers* list
 - d. Calls the `append_random_numbers` function with only one argument to add one number to *numbers*
 - e. Prints the *numbers* list
 - f. Calls the `append_random_numbers` function again with two arguments to add three numbers to *numbers*
 - g. Prints the *numbers* list
2. `append_random_numbers`
 - a. Has two parameters: a list named *numbers_list* and an integer named *quantity*. The parameter *quantity* has a default value of 1
 - b. Computes *quantity*pseudo random numbers by calling the `random.uniform` function.
 - c. Rounds the *quantity*pseudo random numbers to one digit after the decimal.
 - d. Appends the *quantity*pseudo random numbers onto the end of the *numbers_list*.

At the bottom of your program, write an if statement that calls the `main` function. Then run your program and verify that your program works correctly.

Core Requirements

1. Your program includes two functions named `main` and `append_random_numbers`. The `append_random_numbers` function has two parameters named *numbers_list* and *quantity*, and *quantity* has a default value of 1.
2. The `main` function calls `append_random_numbers` twice, first with one argument and second with two arguments.

3. The `append_random_numbers` function includes a loop that appends *quantity* random numbers at the end of *numbers_list*.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add a function named `append_random_words` that meets the following criteria:
 - a. Has two parameters: a list named *words_list* and an integer named *quantity*. The parameter *quantity* has a default value of 1
 - b. Randomly selects *quantity* words from a list of words and appends the selected words at the end of *words_list*.
2. Add statements in the `main` function that create a list of words, call the `append_random_words` function, and then print the list of words.
3. Add something or change something in your program that you think would make your program better, easier for the user, more elegant, or more fun. Be creative.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download the [`test_random_numbers.py`](#) Python file and save it in the same folder where you saved your `random_numbers.py` program. Run the `test_random_numbers.py` file and ensure that the `test_random_numbers` function passes. If it doesn't pass, there is a mistake in your `random_numbers` function. Read the output from pytest, fix the mistake, and run `test_random_numbers.py` again until the test function passes.

```
> python test_random_numbers.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson07
collected 1 item

test_random_numbers.py::test_random_numbers PASSED [100%]

===== 1 passed in 0.11s =====
```

2. Run your program and ensure that your program's output is similar* to the output below.

```
> python random_numbers.py
numbers [16.2, 75.1, 52.3]
numbers [16.2, 75.1, 52.3, 84.2]
numbers [16.2, 75.1, 52.3, 84.2, 99.5, 20.4, 25.3]
words ['join', 'love', 'smile', 'love', 'cloud', 'head']
```

- * Because your program is appending random numbers to the *numbers* list, your program's output will be slightly different than the output shown above. Specifically, the last four random numbers and the random words will be different.

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#). Please **do not look at the sample solution** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Ponder

Look again at the `append_random_numbers` function (and the `append_random_words` function if you wrote it for the stretch challenge). The first parameter, *numbers_list*, is a list. Recall that in Python, lists are passed by reference which means that any changes made to the list in the `append_random_numbers` (and `append_random_words`) function will change the list in `main`. You can verify this is true by looking at the output of your program. The numbers that were appended onto the list by the `append_random_numbers` function are stored in the same list in `main`.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson07/test_random_numbers.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from random_numbers import append_random_numbers
4 # from random_numbers import append_random_words
5 import pytest
6
7
8 def test_append_random_numbers():
9     """Verify that the append_random_numbers function works correctly.
10    Parameters: none
11    Return: nothing
12    """
13    # Create an empty list named numbers_list.
14    numbers_list = []
15
16    # Verify that the length of the empty list is zero.
17    assert len(numbers_list) == 0
18
19    # Call the append_random_numbers function to append one number.
20    append_random_numbers(numbers_list)
21
22    # Verify that the numbers list now has one element.
23    assert len(numbers_list) == 1
24
25    # Verify that all the elements in the numbers list
26    # are floating point numbers.
27    for x in numbers_list:
28        assert isinstance(x, float)
29
30    # Call the append_random_numbers function to append three numbers.
31    append_random_numbers(numbers_list, 3)
32
33    # Verify that the numbers list now has four elements.
34    assert len(numbers_list) == 4
35
36    # Verify that all the elements in the numbers list
37    # are floating point numbers.
38    for x in numbers_list:
39        assert isinstance(x, float)
40
41
42 #def test_append_random_words():
43 #     """Verify that the append_random_words function works correctly.
44 #    Parameters: none
45 #    Return: nothing
46 #    """
47 #     words_list = []
48 #     assert len(words_list) == 0
49 #
50 #     append_random_words(words_list)
51 #     assert len(words_list) == 1
52 #     for word in words_list:
53 #         assert isinstance(word, str)
54 #         assert len(word) >= 1
55 #
56 #     append_random_words(words_list, 3)
57 #     assert len(words_list) == 4
```

```
58 #     for word in words_list:  
59 #         assert isinstance(word, str)  
60 #         assert len(word) >= 1  
61  
62  
63 # Call the main function that is part of pytest so that the  
64 # computer will execute the test functions in this file.  
65 pytest.main(["-v", "--tb=line", "-rN", __file__])
```

lesson07/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 Write and call functions that demonstrate both
4 default parameter values and pass by reference.
5 """
6 import random
7
8 def main():
9     numbers = [16.2, 75.1, 52.3]
10    print(f"numbers {numbers}")
11
12    # Call the append_random_numbers function to
13    # add one random number to the numbers list.
14    append_random_numbers(numbers)
15    print(f"numbers {numbers}")
16
17    # Call the append_random_numbers function to add
18    # three random numbers to the numbers list.
19    append_random_numbers(numbers, 3)
20    print(f"numbers {numbers}")
21
22    # Create a list to store random words.
23    words = []
24
25    # Call the append_random_words function
26    # to add one random word to the list.
27    append_random_words(words)
28    print(f"words {words}")
29
30    # Call the append_random_words function
31    # to add five random words to the list.
32    append_random_words(words, 5)
33    print(f"words {words}")
34
35
36
37 def append_random_numbers(numbers_list, quantity=1):
38     """Append quantity random numbers onto the numbers list.
39     The random numbers are between 0 and 100, inclusive.
40     Parameters
41         numbers_list: A list of numbers where this function will
42             append random numbers.
43         quantity: The number of random numbers that this function
44             will append onto numbers_list.
45     Return: nothing. It's unnecessary for this function to return
46             anything because this function changes the numbers_list.
47     """
48     for _ in range(quantity):
49         random_number = random.uniform(0, 100)
50         rounded = round(random_number, 1)
51         numbers_list.append(rounded)
52
53
54 def append_random_words(words_list, quantity=1):
55     """Append quantity randomly chosen words onto the words list.
56     Parameters
57         words_list: A list of words where this function will
```

```
58         append random words.
59     quantity: The number of random words that this function
60             will append onto words_list.
61     Return: nothing. It's unnecessary for this function to return
62             anything because this function changes the words_list.
63     """
64
65     # A list of words to randomly choose from.
66     candidates = [
67         "arm", "car", "cloud", "head", "heal", "hydrogen", "jog",
68         "join", "laugh", "love", "sleep", "smile", "speak",
69         "sunshine", "toothbrush", "tree", "truth", "walk", "water"
70     ]
71
72     # Randomly choose quantity words and append them onto words_list.
73     for _ in range(quantity):
74         word = random.choice(candidates)
75         words_list.append(word)
76
77
78     # If this file was executed like this:
79     # > python teach_solution.py
80     # then call the main function. However, if this file
81     # was simply imported, then skip the call to main.
82     if __name__ == "__main__":
83         main()
```

07 Prove Milestone: Lists

Purpose

Prove that you can write a Python program that creates and uses a compound list.

Problem Statement

In chemistry, the **molar mass** of a substance is the mass in grams of one mole of the substance (grams / mole). A **mole** is simply a fixed very large quantity, specifically $602,214,076,000,000,000,000,000$ (usually written as $6.02214076 \times 10^{23}$). A molar mass calculator is a program that computes the molar mass of a substance and the number of moles of a sample of that substance. To use a molar mass calculator, a chemist enters two inputs:

- The formula for a molecule, such as H₂O (water) or C₆H₁₂O₆ (glucose)
- The mass in grams of a sample of the substance, such as 3.71

The calculator computes the molar mass of the molecule by doing the following for each element in the formula:

1. Sum the number of atoms of each element in the formula
2. Find the atomic mass of each element
3. Multiply the number of atoms by their atomic mass
4. Add the product into the molar mass of the molecule

Then the calculator uses this formula to compute the number of moles in the sample:

$$\text{number_of_moles} = \frac{\text{sample_mass}}{\text{molar_mass}}$$

Finally, the calculator prints two results for the chemist to see:

- the molar mass
- the number of moles

Example

As an example, consider a sample of glucose (C₆H₁₂O₆) with a mass of 12.37 grams. To use a molar mass calculator, a chemist would enter

- C₆H₁₂O₆
- 12.37 grams

The calculator would compute the molar mass of glucose by doing the following:

1. Sum the number of atoms of each element in the formula for glucose:

6 carbon atoms
12 hydrogen atoms
6 oxygen atoms

2. Find the atomic mass of each element:

Symbol	Name	Atomic Mass
C	Carbon	12.0107
H	Hydrogen	1.00794
O	Oxygen	15.9994

3. Multiply the number of atoms by their atomic mass:

$$\begin{aligned} 6 \times 12.0107 &= 72.0642 \\ 12 \times 1.00794 &= 12.09528 \\ 6 \times 15.9994 &= 95.9964 \end{aligned}$$

4. Add the results of the multiplications to get the molar mass of glucose:

$$72.0642 + 12.09528 + 95.9964 = 180.15588 \text{ grams/mole}$$

Then the calculator would divide the mass of the sample of glucose by the molar mass of glucose which results in the number of moles in the sample:

$$\frac{12.37 \text{ grams}}{180.15588 \text{ grams/mole}} = 0.06866 \text{ moles}$$

The calculator would then print two results for the chemist to see:

- the molar mass of glucose: 180.15588 grams/mole
- the number of moles in the sample: 0.06866 moles

Table of Elements

There are 94 elements known to occur naturally on earth. The symbol, name, and atomic mass of all 94 elements are shown in the following table.

Symbol	Name	Atomic Mass
"Ac",	"Actinium",	227
"Ag",	"Silver",	107.8682
"Al",	"Aluminum",	26.9815386
"Ar",	"Argon",	39.948

Symbol	Name	Atomic Mass
"As",	"Arsenic",	74.9216
"At",	"Astatine",	210
"Au",	"Gold",	196.966569
"B",	"Boron",	10.811
"Ba",	"Barium",	137.327
"Be",	"Beryllium",	9.012182
"Bi",	"Bismuth",	208.9804
"Br",	"Bromine",	79.904
"C",	"Carbon",	12.0107
"Ca",	"Calcium",	40.078
"Cd",	"Cadmium",	112.411
"Ce",	"Cerium",	140.116
"Cl",	"Chlorine",	35.453
"Co",	"Cobalt",	58.933195
"Cr",	"Chromium",	51.9961
"Cs",	"Cesium",	132.9054519
"Cu",	"Copper",	63.546
"Dy",	"Dysprosium",	162.5
"Er",	"Erbium",	167.259
"Eu",	"Europium",	151.964
"F",	"Fluorine",	18.9984032
"Fe",	"Iron",	55.845
"Fr",	"Francium",	223
"Ga",	"Gallium",	69.723
"Gd",	"Gadolinium",	157.25
"Ge",	"Germanium",	72.64
"H",	"Hydrogen",	1.00794
"He",	"Helium",	4.002602
"Hf",	"Hafnium",	178.49
"Hg",	"Mercury",	200.59
"Ho",	"Holmium",	164.93032
"I",	"Iodine",	126.90447
"In",	"Indium",	114.818
"Ir",	"Iridium",	192.217
"K",	"Potassium",	39.0983
"Kr",	"Krypton",	83.798
"La",	"Lanthanum",	138.90547
"Li",	"Lithium",	6.941
"Lu",	"Lutetium",	174.9668
"Mg",	"Magnesium",	24.305
"Mn",	"Manganese",	54.938045

Symbol	Name	Atomic Mass
"Mo",	"Molybdenum",	95.96
"N",	"Nitrogen",	14.0067
"Na",	"Sodium",	22.98976928
"Nb",	"Niobium",	92.90638
"Nd",	"Neodymium",	144.242
"Ne",	"Neon",	20.1797
"Ni",	"Nickel",	58.6934
"Np",	"Neptunium",	237
"O",	"Oxygen",	15.9994
"Os",	"Osmium",	190.23
"P",	"Phosphorus",	30.973762
"Pa",	"Protactinium",	231.03588
"Pb",	"Lead",	207.2
"Pd",	"Palladium",	106.42
"Pm",	"Promethium",	145
"Po",	"Polonium",	209
"Pr",	"Praseodymium",	140.90765
"Pt",	"Platinum",	195.084
"Pu",	"Plutonium",	244
"Ra",	"Radium",	226
"Rb",	"Rubidium",	85.4678
"Re",	"Rhenium",	186.207
"Rh",	"Rhodium",	102.9055
"Rn",	"Radon",	222
"Ru",	"Ruthenium",	101.07
"S",	"Sulfur",	32.065
"Sb",	"Antimony",	121.76
"Sc",	"Scandium",	44.955912
"Se",	"Selenium",	78.96
"Si",	"Silicon",	28.0855
"Sm",	"Samarium",	150.36
"Sn",	"Tin",	118.71
"Sr",	"Strontium",	87.62
"Ta",	"Tantalum",	180.94788
"Tb",	"Terbium",	158.92535
"Tc",	"Technetium",	98
"Te",	"Tellurium",	127.6
"Th",	"Thorium",	232.03806
"Ti",	"Titanium",	47.867
"Tl",	"Thallium",	204.3833
"Tm",	"Thulium",	168.93421

Symbol	Name	Atomic Mass
"U",	"Uranium",	238.02891
"V",	"Vanadium",	50.9415
"W",	"Tungsten",	183.84
"Xe",	"Xenon",	131.293
"Y",	"Yttrium",	88.90585
"Yb",	"Ytterbium",	173.054
"Zn",	"Zinc",	65.38
"Zr",	"Zirconium",	91.224

Assignment

During this prove milestone and the next prove assignment, you will write and test a molar mass calculator named `chemistry.py`. During this milestone, you will complete part of the calculator by writing a function named `make_periodic_table` and the `main` function. The `make_periodic_table` function must create and return a compound list that contains data for all 94 naturally occurring elements.

Helpful Documentation

If you are interested in the chemistry concepts involved in a molar mass calculator, you can watch this Khan Academy video titled [Calculating molar mass and number of moles](#) (6 minutes).

The prepare content for this lesson explains how to [create and use a compound list](#).

The [prepare content for lesson 5](#) explains how to use `pytest`, `assert`, and `approx` to automatically verify that functions are correct. It also contains an [example test function](#) and links to additional documentation about `pytest`.

Steps

Do the following:

1. Using VS Code, create a new file and save it as `chemistry.py`
2. In the `chemistry.py` file, write a function named `make_periodic_table` that takes no parameters and creates and returns a compound list. The compound list must contain all the data in the table of elements shown in the Problem Statement section above. The data within the compound list must be organized like this:
[]

```
periodic_table_list = [
    # [symbol, name, atomic_mass]
    ["Ac", "Actinium", 227],
    ["Ag", "Silver", 107.8682],
    ["Al", "Aluminum", 26.9815386],
    :
]
```

We **strongly recommend** that you **do not type the data** in the table of elements but instead that you use **copy and paste** to copy the data from this assignment into your program. If you don't know how to use copy and paste to help you quickly write the `make_periodic_table` function, ask a fellow student, a tutor, a teaching assistant, or your teacher for help.

3. In the `chemistry.py` file, write the `main` function that takes no parameters and returns nothing. The `main` function should do the following:
 - a. Call the `make_periodic_table` function and store the returned list in a variable.
 - b. Print the name and atomic mass for each chemical element on a separate line.
Do not print the chemical element symbols.
4. At the bottom of your `chemistry.py` file, add a call to the `main` function. Be certain to protect the call to `main` with an `if` statement as taught in the [prepare content](#) for lesson 5.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download the [test_chemistry_1.py](#) Python file and save it in the same folder where you saved your `chemistry.py` program. Run the `test_chemistry_1.py` file and ensure that the `test_make_periodic_table` function passes. If it doesn't pass, there is a mistake in your `make_periodic_table` function. Read the output from `pytest`, fix the mistake, and run the `test_chemistry_1.py` file again until the test function passes.

```
> python test_chemistry_1.py
=====
 test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson07
collected 1 item

test_chemistry_1.py::test_make_periodic_table PASSED [100%]

===== 1 passed in 0.14s =====
```

2. Run your `chemistry.py` program and ensure that your program's output matches the following output. The three vertical dots (:) in the output below are called a

vertical ellipsis and mean the output continues. Your program shouldn't print the vertical ellipsis. Instead, your program should print the rest of the output.

```
> python chemistry.py
Actinium 227
Silver 107.8682
Aluminum 26.9815386
Argon 39.948
Arsenic 74.9216
Astatine 210
Gold 196.966569
:
```

Submission

On or before the due date, return to I-Learn and report your progress on this milestone.

lesson07/test_chemistry_1.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from chemistry import make_periodic_table
4 from pytest import approx
5 import pytest
6
7
8 # These are the indexes of the
9 # elements in the periodic table.
10 SYMBOL_INDEX = 0
11 NAME_INDEX = 1
12 ATOMIC_MASS_INDEX = 2
13
14
15 def test_make_periodic_table():
16     """Verify that the make_periodic_table function works correctly.
17     Parameters: none
18     Return: nothing
19     """
20     # Call the make_periodic_table and store the returned
21     # periodic table in a variable named periodic_table_list.
22     periodic_table_list = make_periodic_table()
23     assert isinstance(periodic_table_list, list), \
24         "make_periodic_table function must return a list:" \
25         f" expected a list but found a {type(periodic_table_list)}"
26
27     # Create a key function that will be used by the sorted method.
28     by_name = lambda element: element[NAME_INDEX]
29
30     # Sort the periodic table by the element names.
31     periodic_table_list = sorted(periodic_table_list, key=by_name)
32
33     # Check each element in the sorted periodic table.
34     i = 0
35     check_element(periodic_table_list[i], ['Ac', 'Actinium', 227])
36     i += 1
37     check_element(periodic_table_list[i], ['Al', 'Aluminum', 26.9815386])
38     i += 1
39     check_element(periodic_table_list[i], ['Sb', 'Antimony', 121.76])
40     i += 1
41     check_element(periodic_table_list[i], ['Ar', 'Argon', 39.948])
42     i += 1
43     check_element(periodic_table_list[i], ['As', 'Arsenic', 74.9216])
44     i += 1
45     check_element(periodic_table_list[i], ['At', 'Astatine', 210])
46     i += 1
47     check_element(periodic_table_list[i], ['Ba', 'Barium', 137.327])
48     i += 1
49     check_element(periodic_table_list[i], ['Be', 'Beryllium', 9.012182])
50     i += 1
51     check_element(periodic_table_list[i], ['Bi', 'Bismuth', 208.9804])
52     i += 1
53     check_element(periodic_table_list[i], ['B', 'Boron', 10.811])
54     i += 1
55     check_element(periodic_table_list[i], ['Br', 'Bromine', 79.904])
56     i += 1
57     check_element(periodic_table_list[i], ['Cd', 'Cadmium', 112.411])
```

```

58     i += 1
59     check_element(periodic_table_list[i], ['Ca', 'Calcium', 40.078])
60     i += 1
61     check_element(periodic_table_list[i], ['C', 'Carbon', 12.0107])
62     i += 1
63     check_element(periodic_table_list[i], ['Ce', 'Cerium', 140.116])
64     i += 1
65     check_element(periodic_table_list[i], ['Cs', 'Cesium', 132.9054519])
66     i += 1
67     check_element(periodic_table_list[i], ['Cl', 'Chlorine', 35.453])
68     i += 1
69     check_element(periodic_table_list[i], ['Cr', 'Chromium', 51.9961])
70     i += 1
71     check_element(periodic_table_list[i], ['Co', 'Cobalt', 58.933195])
72     i += 1
73     check_element(periodic_table_list[i], ['Cu', 'Copper', 63.546])
74     i += 1
75     check_element(periodic_table_list[i], ['Dy', 'Dysprosium', 162.5])
76     i += 1
77     check_element(periodic_table_list[i], ['Er', 'Erbium', 167.259])
78     i += 1
79     check_element(periodic_table_list[i], ['Eu', 'Europium', 151.964])
80     i += 1
81     check_element(periodic_table_list[i], ['F', 'Fluorine', 18.9984032])
82     i += 1
83     check_element(periodic_table_list[i], ['Fr', 'Francium', 223])
84     i += 1
85     check_element(periodic_table_list[i], ['Gd', 'Gadolinium', 157.25])
86     i += 1
87     check_element(periodic_table_list[i], ['Ga', 'Gallium', 69.723])
88     i += 1
89     check_element(periodic_table_list[i], ['Ge', 'Germanium', 72.64])
90     i += 1
91     check_element(periodic_table_list[i], ['Au', 'Gold', 196.966569])
92     i += 1
93     check_element(periodic_table_list[i], ['Hf', 'Hafnium', 178.49])
94     i += 1
95     check_element(periodic_table_list[i], ['He', 'Helium', 4.002602])
96     i += 1
97     check_element(periodic_table_list[i], ['Ho', 'Holmium', 164.93032])
98     i += 1
99     check_element(periodic_table_list[i], ['H', 'Hydrogen', 1.00794])
100    i += 1
101    check_element(periodic_table_list[i], ['In', 'Indium', 114.818])
102    i += 1
103    check_element(periodic_table_list[i], ['I', 'Iodine', 126.90447])
104    i += 1
105    check_element(periodic_table_list[i], ['Ir', 'Iridium', 192.217])
106    i += 1
107    check_element(periodic_table_list[i], ['Fe', 'Iron', 55.845])
108    i += 1
109    check_element(periodic_table_list[i], ['Kr', 'Krypton', 83.798])
110    i += 1
111    check_element(periodic_table_list[i], ['La', 'Lanthanum', 138.90547])
112    i += 1
113    check_element(periodic_table_list[i], ['Pb', 'Lead', 207.2])
114    i += 1
115    check_element(periodic_table_list[i], ['Li', 'Lithium', 6.941])
116    i += 1
117    check_element(periodic_table_list[i], ['Lu', 'Lutetium', 174.9668])
118    i += 1

```

```

119 check_element(periodic_table_list[i], ['Mg', 'Magnesium', 24.305])
120 i += 1
121 check_element(periodic_table_list[i], ['Mn', 'Manganese', 54.938045]
122 i += 1
123 check_element(periodic_table_list[i], ['Hg', 'Mercury', 200.59])
124 i += 1
125 check_element(periodic_table_list[i], ['Mo', 'Molybdenum', 95.96])
126 i += 1
127 check_element(periodic_table_list[i], ['Nd', 'Neodymium', 144.242])
128 i += 1
129 check_element(periodic_table_list[i], ['Ne', 'Neon', 20.1797])
130 i += 1
131 check_element(periodic_table_list[i], ['Np', 'Neptunium', 237])
132 i += 1
133 check_element(periodic_table_list[i], ['Ni', 'Nickel', 58.6934])
134 i += 1
135 check_element(periodic_table_list[i], ['Nb', 'Niobium', 92.90638])
136 i += 1
137 check_element(periodic_table_list[i], ['N', 'Nitrogen', 14.0067])
138 i += 1
139 check_element(periodic_table_list[i], ['Os', 'Osmium', 190.23])
140 i += 1
141 check_element(periodic_table_list[i], ['O', 'Oxygen', 15.9994])
142 i += 1
143 check_element(periodic_table_list[i], ['Pd', 'Palladium', 106.42])
144 i += 1
145 check_element(periodic_table_list[i], ['P', 'Phosphorus', 30.973762]
146 i += 1
147 check_element(periodic_table_list[i], ['Pt', 'Platinum', 195.084])
148 i += 1
149 check_element(periodic_table_list[i], ['Pu', 'Plutonium', 244])
150 i += 1
151 check_element(periodic_table_list[i], ['Po', 'Polonium', 209])
152 i += 1
153 check_element(periodic_table_list[i], ['K', 'Potassium', 39.0983])
154 i += 1
155 check_element(periodic_table_list[i], ['Pr', 'Praseodymium', 140.90]
156 i += 1
157 check_element(periodic_table_list[i], ['Pm', 'Promethium', 145])
158 i += 1
159 check_element(periodic_table_list[i], ['Pa', 'Protactinium', 231.03]
160 i += 1
161 check_element(periodic_table_list[i], ['Ra', 'Radium', 226])
162 i += 1
163 check_element(periodic_table_list[i], ['Rn', 'Radon', 222])
164 i += 1
165 check_element(periodic_table_list[i], ['Re', 'Rhenium', 186.207])
166 i += 1
167 check_element(periodic_table_list[i], ['Rh', 'Rhodium', 102.9055])
168 i += 1
169 check_element(periodic_table_list[i], ['Rb', 'Rubidium', 85.4678])
170 i += 1
171 check_element(periodic_table_list[i], ['Ru', 'Ruthenium', 101.07])
172 i += 1
173 check_element(periodic_table_list[i], ['Sm', 'Samarium', 150.36])
174 i += 1
175 check_element(periodic_table_list[i], ['Sc', 'Scandium', 44.955912]
176 i += 1
177 check_element(periodic_table_list[i], ['Se', 'Selenium', 78.96])
178 i += 1
179 check_element(periodic_table_list[i], ['Si', 'Silicon', 28.0855])

```

```

180     i += 1
181     check_element(periodic_table_list[i], ['Ag', 'Silver', 107.8682])
182     i += 1
183     check_element(periodic_table_list[i], ['Na', 'Sodium', 22.98976928])
184     i += 1
185     check_element(periodic_table_list[i], ['Sr', 'Strontium', 87.62])
186     i += 1
187     check_element(periodic_table_list[i], ['S', 'Sulfur', 32.065])
188     i += 1
189     check_element(periodic_table_list[i], ['Ta', 'Tantalum', 180.94788])
190     i += 1
191     check_element(periodic_table_list[i], ['Tc', 'Technetium', 98])
192     i += 1
193     check_element(periodic_table_list[i], ['Te', 'Tellurium', 127.6])
194     i += 1
195     check_element(periodic_table_list[i], ['Tb', 'Terbium', 158.92535])
196     i += 1
197     check_element(periodic_table_list[i], ['Tl', 'Thallium', 204.3833])
198     i += 1
199     check_element(periodic_table_list[i], ['Th', 'Thorium', 232.03806])
200     i += 1
201     check_element(periodic_table_list[i], ['Tm', 'Thulium', 168.93421])
202     i += 1
203     check_element(periodic_table_list[i], ['Sn', 'Tin', 118.71])
204     i += 1
205     check_element(periodic_table_list[i], ['Ti', 'Titanium', 47.867])
206     i += 1
207     check_element(periodic_table_list[i], ['W', 'Tungsten', 183.84])
208     i += 1
209     check_element(periodic_table_list[i], ['U', 'Uranium', 238.02891])
210     i += 1
211     check_element(periodic_table_list[i], ['V', 'Vanadium', 50.9415])
212     i += 1
213     check_element(periodic_table_list[i], ['Xe', 'Xenon', 131.293])
214     i += 1
215     check_element(periodic_table_list[i], ['Yb', 'Ytterbium', 173.054])
216     i += 1
217     check_element(periodic_table_list[i], ['Y', 'Yttrium', 88.90585])
218     i += 1
219     check_element(periodic_table_list[i], ['Zn', 'Zinc', 65.38])
220     i += 1
221     check_element(periodic_table_list[i], ['Zr', 'Zirconium', 91.224])
222
223
224 def check_element(actual, expected):
225     """Verify that the actual element that came from the
226     periodic_table_list contains the same values as the
227     expected element.
228
229     Parameters
230         actual: a list that came from the periodic_table_list.
231             expected: a list that contains the expected values.
232     Return: nothing
233     """
234     name = expected[NAME_INDEX]
235     assert actual[NAME_INDEX] == name, \
236         f"{name} is missing from the periodic table."
237
238     # Verify that the element's symbol is correct.
239     act_symbol = actual[SYMBOL_INDEX]
240     exp_symbol = expected[SYMBOL_INDEX]

```

```
241     assert act_symbol == exp_symbol, \
242         f"wrong symbol for {name}: " \
243         f"expected {exp_symbol} but found {act_symbol}."
244
245     # Verify that the element's atomic mass is correct.
246     act_mass = actual[ATOMIC_MASS_INDEX]
247     exp_mass = expected[ATOMIC_MASS_INDEX]
248     assert act_mass == approx(exp_mass), \
249         f"wrong atomic mass for {name}: " \
250         f"expected {exp_mass} but found {act_mass}"
251
252
253     # Call the main function that is part of pytest so that the
254     # computer will execute the test functions in this file.
255     pytest.main(["-v", "--tb=line", "-rN", __file__])
```

08 Prepare: Dictionaries

In this lesson, you will learn how to store data in and retrieve data from Python dictionaries. You will also learn how to write a loop that processes all the items in a dictionary.

Video

Watch this video about dictionaries in Python.

[Dictionaries](#) (6 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson:

Dictionaries

A Python program can store many items in a **dictionary**. Each **item** in a dictionary is a key value pair. Each **key** within a dictionary must be unique. In other words, no key can appear more than once in a dictionary. Values within a dictionary do not have to be unique. Dictionaries are mutable, meaning they can be changed after they are created. Dictionaries were invented to enable computers to find items quickly.

The following table represents a dictionary that contains five items (five key value pairs). Notice that each of the keys is unique.

items		
keys	values	
42-039-4736	Clint Huish	↑
61-315-0160	Amelia Davis	5
10-450-1203	Ana Soares	items
75-421-2310	Abdul Ali	
07-103-5621	Amelia Davis	↓

We can create a dictionary by using curly braces ({} and {}). We can add an item to a dictionary and find an item in a dictionary by using square brackets ([and]) and a key. The following code example shows how to create a dictionary, add an item, remove an item, and find an item in a dictionary.

```

1 # Example 1
2
3 def main():
4     # Create a dictionary with student IDs as
5     # the keys and student names as the values.
6     students = {
7         "42-039-4736": "Clint Huish",
8         "61-315-0160": "Amelia Davis",
9         "10-450-1203": "Ana Soares",
10        "75-421-2310": "Abdul Ali",
11        "07-103-5621": "Amelia Davis"
12    }
13
14    # Add an item to the dictionary.
15    students["81-298-9238"] = "Sama Patel"
16
17    # Remove an item from the dictionary.
18    students.pop("61-315-0160")
19
20    # Get the number of items in the dictionary and print it.
21    length = len(students)
22    print(f"length: {length}")
23
24    # Print the entire dictionary.
25    print(students)
26    print()
27
28    # Get a student ID from the user.
29    id = input("Enter a student ID: ")
30
31    # Check if the student ID is in the dictionary.
32    if id in students:
33
34        # Find the student ID in the dictionary and
35        # retrieve the corresponding student name.
36        name = students[id]
37
38        # Print the student's name.
39        print(name)
40
41    else:
42        print("No such student")
43
44
45    # Call main to start this program.
46    if __name__ == "__main__":
47        main()

```

```

> python example_1.py
length: 5
{'42-039-4736': 'Clint Huish', '10-450-1203': 'Ana Soares',
'75-421-2310': 'Abdul Ali', '07-103-5621': 'Amelia Davis',
'81-298-9238': 'Sama Patel'}

Enter a student ID: 10-450-1203
Ana Soares

```

Line 15 in the previous code example, adds an item to the dictionary. To add an item to an existing dictionary, write code that follows this template:

```
dictionary_name[key] = value
```

Notice that line 15 follows this template.

Line 32 in the previous code example, uses the Python membership operator, which is the keyword `in`, to check if a key is stored in a dictionary. To check if a key is stored in a dictionary, write code that follows this template:

```
if key in dictionary_name:
```

Notice that line 32 follows this template.

Line 36 in the previous code example, finds a key and retrieves its corresponding value from a dictionary. To find a key and retrieve its corresponding value, write code that follows this template:

```
value = dictionary_name[key]
```

Notice that line 36 follows this template.

Compound Values

A **simple value** is a value that doesn't contain parts, such as an integer. A **compound value** is a value that has parts, such as a list. In example 1 above, the `students` dictionary has simple keys and values. Each key is a single string, and each value is a single string. It is possible to store compound values in a dictionary. Example 2 shows a `students` dictionary where each value is a Python list. Because each list contains multiple parts, we say that the dictionary stores compound values.

```
# Example 2

def main():
    # Create a dictionary with student IDs as the keys
    # and student data stored in a list as the values.
    students = {
        # student_ID: [given_name, surname, email_address, credits]
        "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
        "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
        "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0]
    }
```

Finding One Item

The reason Python dictionaries were developed is to make finding items easy and fast. As explained in example 1, to find an item in a dictionary, a programmer needs to write just one line of code that follows this template:

```
value = dictionary_name[key]
```

That one line of code will cause the computer to search the dictionary until it finds the *key*. Then the computer will return the *value* that corresponds to the *key*. Some students forget how easy it is to find items in a dictionary, and when asked to write code to find an item, they write complex code like [lines 24–28](#) in example 3.

```
1 # Example 3
2
3 def main():
4     # Create a dictionary with student IDs as the keys
5     # and student data stored in a list as the values.
6     students = {
7         # student_ID: [given_name, surname, email_address, credits]
8         "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
9         "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
10        "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
11        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
12        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0]
13    }
14
15    # Get a student ID from the user.
16    id = input("Enter a student ID: ")
17
18    # This is a difficult and slow way to find an item in a
19    # dictionary. Don't write code like this to find an item
20    # in a dictionary!
21
22    # For each item in the dictionary, check if
23    # its key is the same as the variable id.
24    student = None
25    for key, value in students.items(): # Bad example!
26        if key == id:                # Don't use a loop like
27            student = value          # this to find an item
28            break                     # in a dictionary.
```

Compare the `for` loop at [lines 24–28](#) in the previous example to this one line of code.

```
value = students[id]
```

Clearly, writing one line of code is easier for a programmer than writing the `for` loop. Not only is the one line of code easier to write, but the computer will execute it much, much faster than the `for` loop. Therefore, when you need to write code to find an item in a dictionary, don't write a loop. Instead, write one line of code that uses the square brackets ([and]) and a key to find an item. Example 4 shows the correct way to find an item in a dictionary.

```
1 # Example 4
```

```

2
3 def main():
4     # Create a dictionary with student IDs as the keys
5     # and student data stored in a list as the values.
6     students = {
7         # student_ID: [given_name, surname, email_address, credits]
8         "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
9         "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
10        "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
11        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
12        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0]
13    }
14
15    # These are the indexes of the elements in the value lists.
16    GIVEN_NAME_INDEX = 0
17    SURNAME_INDEX = 1
18    EMAIL_INDEX = 2
19    CREDITS_INDEX = 3
20
21    # Get a student ID from the user.
22    id = input("Enter a student ID: ")
23
24    # Check if the student ID is in the dictionary.
25    if id in students:
26
27        # Find the student ID in the dictionary and
28        # retrieve the corresponding value, which is a list.
29        value = students[id]
30
31        # Retrieve the student's given name (first name) and
32        # surname (last name or family name) from the list.
33        given_name = value[GIVEN_NAME_INDEX]
34        surname = value[SURNAME_INDEX]
35
36        # Print the student's name.
37        print(f"{given_name} {surname}")
38
39    else:
40        print("No such student")
41
42
43    # Call main to start this program.
44    if __name__ == "__main__":
45        main()

```

```

> python example_4.py
Enter a student ID: 61-315-0160
Amelia Davis

> python example_4.py
Enter a student ID: 25-143-1202
No such student

```

Processing All Items

Occasionally, you may need to write a program that processes all the items in a dictionary. Processing all the items in a dictionary is different than finding one item in a

dictionary. Processing all the items is done using a `for` loop and the `dict.items()` method as shown in example 5 on line 25.

```
1 # Example 5
2
3 def main():
4     # Create a dictionary with student IDs as the keys
5     # and student data stored in a list as the values.
6     students = {
7         "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
8         "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
9         "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
10        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
11        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0],
12        "81-298-9238": ["Sama", "Patel", "pat21004@byui.edu", 8]
13    }
14
15    # These are the indexes of the elements in the value lists.
16    GIVEN_NAME_INDEX = 0
17    SURNAME_INDEX = 1
18    EMAIL_INDEX = 2
19    CREDITS_INDEX = 3
20
21    total = 0
22
23    # For each item in the list add the number
24    # of credits that the student has earned.
25    for item in students.items():
26        key = item[0]
27        value = item[1]
28
29        # Retrieve the number of credits from the value list.
30        credits = value[CREDITS_INDEX]
31
32        # Add the number of credits to the total.
33        total += credits
34
35    print(f"Total credits earned by all students: {total}")
36
37
38    # Call main to start this program.
39    if __name__ == "__main__":
40        main()
```

```
> python example_5.py
Total credits earned by all students: 47
```

As with all the example code in CSE 111, example 5 contains working Python code. Even though the code works, we can combine lines 25–27 into a single line of code by using a Python shortcut called unpacking. Instead of writing lines 25–27, like this:

```
for item in students.items():
    key = item[0]
    value = item[1]
```

We can write one line of code that combines the three lines of code and unpacks the item in the `for` statement like this:

```
for key, value in students.items():
```

Example 6 contains the same code as example 5 except example 6 uses the Python unpacking shortcut at [line 25](#).

```
1 # Example 6
2
3 def main():
4     # Create a dictionary with student IDs as the keys
5     # and student data stored in a list as the values.
6     students = {
7         "42-039-4736": ["Clint", "Huish", "hui20001@byui.edu", 16],
8         "61-315-0160": ["Amelia", "Davis", "dav21012@byui.edu", 3],
9         "10-450-1203": ["Ana", "Soares", "soa22005@byui.edu", 15],
10        "75-421-2310": ["Abdul", "Ali", "ali20003@byui.edu", 5],
11        "07-103-5621": ["Amelia", "Davis", "dav19008@byui.edu", 0],
12        "81-298-9238": ["Sama", "Patel", "pat21004@byui.edu", 8]
13    }
14
15    # These are the indexes of the elements in the value lists.
16    GIVEN_NAME_INDEX = 0
17    SURNAME_INDEX = 1
18    EMAIL_INDEX = 2
19    CREDITS_INDEX = 3
20
21    total = 0
22
23    # For each item in the list add the number
24    # of credits that the student has earned.
25    for key, value in students.items():
26
27        # Retrieve the number of credits from the value list.
28        credits = value[CREDITS_INDEX]
29
30        # Add the number of credits to the total.
31        total += credits
32
33    print(f"Total credits earned by all students: {total}")
34
35
36    # Call main to start this program.
37    if __name__ == "__main__":
38        main()
```

```
> python example_6.py
Total credits earned by all students: 47
```

Dictionaries Are Similar to Lists

Dictionaries are similar to lists in a few ways. The following table shows the similarities and differences of lists and dictionaries.

	Lists	Dictionaries
Similar	A list can store many elements.	A dictionary can store many items.
Same	Lists are mutable, meaning a program can add and remove elements after a list is created.	Dictionaries are mutable, meaning a program can add and remove items after a dictionary is created.
Different	Each element in a list does not have to be unique.	Each item in a dictionary is a key value pair. Each key must be unique within a dictionary. Each value does not have to be unique.
Different	Lists were designed for efficiently storing elements. Lists use less memory than dictionaries. However, finding an element in a list is relatively slow.	Dictionaries were designed for quickly finding items. Finding an item in a dictionary is fast. However, dictionaries use more memory than lists.
Same	Lists are passed by reference into a function.	Dictionaries are passed by reference into a function.
Different	A programmer uses square brackets ([and]) to create a list.	A programmer uses curly braces ({ and }) to create a dictionary.
Different	A programmer calls the <code>insert</code> and <code>append</code> methods to add an element to a list.	A programmer uses square brackets ([and]) to add an item to a dictionary.
Similar	A programmer uses square brackets ([and]) and an index to retrieve an element from a list.	A programmer uses square brackets ([and]) and a key to retrieve a value from a dictionary.
Similar	A programmer uses square brackets ([and]) and an index to replace an element in a list.	A programmer uses square brackets ([and]) and a key to replace a value in a dictionary.

	Lists	Dictionaries
Same	A programmer uses the <code>pop</code> method to remove an element from a list.	A programmer uses the <code>pop</code> method to remove an item from a dictionary.

Converting between Lists and Dictionaries

It is possible to convert two lists into a dictionary by using the built-in `zip` and `dict` functions. The contents of the first list will become the keys in the dictionary, and the contents of the second list will become the values. This implies that the two lists must have the same length, and the elements in the first list must be unique because keys in a dictionary must be unique.

It is also possible to convert a dictionary into two lists by using the `keys` and `values` methods and the built-in `list` function. The following code example starts with two lists, converts them into a dictionary, and then converts the dictionary into two lists.

```

1 # Example 7
2
3 def main():
4     # Create a list that contains five student numbers.
5     numbers = ["42-039-4736", "61-315-0160",
6                 "10-450-1203", "75-421-2310", "07-103-5621"]
7
8     # Create a list that contains five student names.
9     names = ["Clint Huish", "Amelia Davis",
10               "Ana Soares", "Abdul Ali", "Amelia Davis"]
11
12     # Convert the numbers list and names list into a dictionary.
13     student_dict = dict(zip(numbers, names))
14
15     # Print the entire student dictionary.
16     print("Dictionary:", student_dict)
17     print()
18
19     # Convert the student dictionary into
20     # two lists named keys and values.
21     keys = list(student_dict.keys())
22     values = list(student_dict.values())
23
24     # Print both lists.
25     print("Keys:", keys)
26     print()
27     print("Values:", values)
28
29
30     # Call main to start this program.
31     if __name__ == "__main__":
32         main()

```

```
> python example_7.py
Dictionary: {'42-039-4736': 'Clint Huish',
'61-315-0160': 'Amelia Davis', '10-450-1203': 'Ana Soares',
'75-421-2310': 'Abdul Ali', '07-103-5621': 'Amelia Davis'}
Keys: ['42-039-4736', '61-315-0160', '10-450-1203',
'75-421-2310', '07-103-5621']
Values: ['Clint Huish', 'Amelia Davis', 'Ana Soares',
'Abdul Ali', 'Amelia Davis']
```

Tutorials

The following tutorials contain more information about dictionaries in Python.

Official Python [tutorial about dictionaries](#)

RealPython tutorial titled [Dictionaries in Python](#)

Summary

A dictionary in a Python program can store many pieces of data called items. An item is a key value pair. Each key that is stored in a dictionary must be unique. Values do not have to be unique. To create a dictionary, we use curly braces ({ and }). To add an item and find an item in a dictionary, we use the square brackets ([and]) and a key. To process all items in a dictionary, we write a for each loop. Dictionaries were invented to enable a computer to find items very quickly. Do not write a for each loop to find an item in a dictionary. To find an item in a dictionary, use square brackets ([and]) and a key.

08 Checkpoint: Dictionaries

Purpose

Improve your ability to use dictionaries in a Python program.

Helpful Documentation

The [prepare content for the previous lesson](#) explains how to retrieve an element from a list.

The [prepare content for this lesson](#) explains how to find items in a dictionary.

Assignment

Write a program that stores information about vehicles in a Python dictionary. Your program must ask a user to enter a vehicle identification number (VIN), then find that VIN in the dictionary, and print the manufacturer, model, and color of the vehicle. Do the following:

1. Download and save the [vehicles.py](#) Python file and then open it in VS Code.
2. Using VS Code, read the comments in the program. Then replace all three occurrences of the keyword pass with code that does what the comments describe.
3. Run your program and verify that it works correctly.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python vehicles.py
Please enter a VIN: QT71603
QT71603 is not in the dictionary.

> python vehicles.py
Please enter a VIN: 5TDZA23CXTU102983
Toyota Sienna gold
```

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson08/vehicles.py

```
1 def main():
2     # Create a dictionary that contains data about six vehicles.
3     # The key for each vehicle in the dictionary is the vehicle's
4     # identification number (VIN). The value for each vehicle is
5     # a list that contains the year, manufacturer, model, color,
6     # engine design, and engine displacement.
7     vehicles_dict = {
8         # VIN: [year, manufacturer, model, color, eng_design, eng_displ
9         "1J4GL48K4UF993861": [2002, "Jeep", "Liberty", "blue", "V6", 3.
10        "1YVGF22C8AN381568": [2002, "Mazda", "626", "white", "I4", 2.0]
11        "WPOAA0926HG410293": [1987, "Porsche", "924S", "red", "I4", 2.5
12        "5TDZA23CXTU102983": [2006, "Toyota", "Sienna", "gold", "V6", 3
13        "1GKKVRED5ZL382610": [2011, "GMC", "Acadia", "charcoal", "V6",
14        "2T3BF4DV9QR146782": [2012, "Toyota", "RAV 4", "green", "I4", 2
15    }
16
17 MANUFACTURER_INDEX = 1
18 MODEL_INDEX = 2
19 COLOR_INDEX = 3
20
21 # Ask the user for a vehicle identification number (VIN).
22 vin = input("Please enter a VIN: ")
23
24 # Check if the vin is a key that is in the vehicles dictionary.
25 if pass:
26
27     # Find the data for the vehicle that the user wants.
28     pass
29
30     # Print the manufacturer, model, and color of the vehicle.
31     # Don't print the year, engine design, or displacement.
32     pass
33
34 else:
35     # Print a message stating that the VIN entered
36     # by the user is not in the dictionary.
37     print(f"{vin} is not in the dictionary.")
38
39
40 # If this file was executed like this:
41 # > python teach_solution.py
42 # then call the main function. However, if this file
43 # was simply imported, then skip the call to main.
44 if __name__ == "__main__":
45     main()
```

lesson08/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def main():
4     # Create a dictionary that contains data about six vehicles.
5     # The key for each vehicle in the dictionary is the vehicle's
6     # identification number (VIN). The value for each vehicle is
7     # a list that contains the year, manufacturer, model, color,
8     # engine design, and engine displacement.
9     vehicles_dict = {
10         # VIN: [year, manufacturer, model, color, eng_design, eng_displ
11         "1J4GL48K4UF993861": [2002, "Jeep", "Liberty", "blue", "V6", 3.
12         "1YVGF22C8AN381568": [2002, "Mazda", "626", "white", "I4", 2.0]
13         "WP0AA0926HG410293": [1987, "Porsche", "924S", "red", "I4", 2.5
14         "5TDZA23CXTU102983": [2006, "Toyota", "Sienna", "gold", "V6", 3
15         "1GKKVRED5ZL382610": [2011, "GMC", "Acadia", "charcoal", "V6",
16         "2T3BF4DV9QR146782": [2012, "Toyota", "RAV 4", "green", "I4", 2
17     }
18
19 MANUFACTURER_INDEX = 1
20 MODEL_INDEX = 2
21 COLOR_INDEX = 3
22
23 # Ask the user for a vehicle identification number (VIN).
24 vin = input("Please enter a VIN: ")
25
26 # Check if the VIN is a key that is in the vehicles dictionary.
27 if vin in vehicles_dict:
28
29     # Find the data for the vehicle that the user wants.
30     value_list = vehicles_dict[vin]
31
32     # Print the manufacturer, model, and color of the vehicle.
33     # Don't print the year, engine design, or displacement.
34     manufacturer = value_list[MANUFACTURER_INDEX]
35     model = value_list[MODEL_INDEX]
36     color = value_list[COLOR_INDEX]
37     print(manufacturer, model, color)
38
39 else:
40     # Print a message stating that the VIN entered
41     # by the user is not in the dictionary.
42     print(f"{vin} is not in the dictionary.")
43
44
45 # If this file was executed like this:
46 # > python teach_solution.py
47 # then call the main function. However, if this file
48 # was simply imported, then skip the call to main.
49 if __name__ == "__main__":
50     main()
```

08 Team Activity: Dictionaries

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Problem Statement

The Church of Jesus Christ of Latter-day Saints uses lots of computer technology to collect and store family history data, including data about individuals and marriages. Interestingly, the data about individuals must be stored separately from the data about marriages because some individuals get married multiple times. However, in order to make the data understandable to people, when a person views marriage data, a program must combine the marriage data and the individual data.

Helpful Documentation

The [prepare content for the previous lesson](#) explains how to retrieve an element from a list.

The prepare content for this lesson explains how to [find an item](#) in a dictionary and how to [process all the items](#) in a dictionary.

Assignment

As a team, write a Python program that stores data about individuals in one dictionary and stores data about marriages in a different dictionary. Your program must combine the data in the two dictionaries and print the combined data so that it is understandable to a user. Start your program by downloading and saving the [family_history.py](#) Python file and then open it in VS Code and complete the Core Requirements.

Core Requirements

1. Within your program, the `print_death_age` function must print the name and age at death for each person in the `people` dictionary.
2. The `count_genders` function must count and print the number of males and the number of females in the `people` dictionary.

3. The `print_marriages` function must print the following for each marriage in the `marriages` dictionary:
 - a. The name and age in the wedding year of the husband
 - b. The year of the wedding
 - c. The name and age in the wedding year of the wife

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add code to the `print_death_age` function that prints the birth year and death year for each person.
2. Add to your program a function named `count_marriages` that counts and prints the number of marriages that each person had in his or her lifetime. According to the data, who married the most times?

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your `family_history.py` program and verify that it prints the same results as shown below.

```
> python family_history.py
Ages at Death
Lola Park 43
Savanna Foster 49
Tiffany Hughes 58
Ignacio Torres 65
Yasmin Li 15
Trent Ross 52
Samyukta Nguyen 57
Joel Johnson 76
Whitney Nelson 66
Khalid Ali 55
Davina Patel 85
Enzo Ruiz 0
Lauren Smith 2
Lucas Ross 53
Jamal Gray 21
Fatima Soares 86
Ephraim Foster 54
Peter Price 46
Rosalina Jimenez 81
Rachel Johnson 64
Vanessa Bennet 80
Jose Castillo 47
Liam Myers 48
Isabella Lopez 52
Megan Anderson 36

Genders
Number of males: 12
Number of females: 13

Marriages
Ignacio Torres 18 > 1711 < Tiffany Hughes 22
Trent Ross 17 > 1722 < Savanna Foster 48
Ignacio Torres 31 > 1724 < Tiffany Hughes 35
Samyukta Nguyen 57 > 1774 < Whitney Nelson 17
Joel Johnson 51 > 1775 < Whitney Nelson 18
Joel Johnson 68 > 1792 < Davina Patel 17
Khalid Ali 45 > 1804 < Whitney Nelson 47
Khalid Ali 49 > 1808 < Davina Patel 33
Jamal Gray 20 > 1830 < Fatima Soares 18
Lucas Ross 53 > 1853 < Fatima Soares 41
Peter Price 27 > 1859 < Davina Patel 84
Ephraim Foster 44 > 1875 < Fatima Soares 63
Jose Castillo 21 > 1905 < Vanessa Bennet 25
Jose Castillo 33 > 1917 < Rachel Johnson 41
Liam Myers 23 > 1925 < Isabella Lopez 18
Jose Castillo 41 > 1925 < Rosalina Jimenez 50
Jose Castillo 44 > 1928 < Megan Anderson 19
```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to this [sample solution](#). Please **do not look at the sample solutions** until you have either finished the program or diligently worked for at

least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson08/family_history.py

```
1 # Each value in the people dictionary is a list. These
2 # are the indexes of the elements in those lists.
3 NAME_INDEX = 0
4 GENDER_INDEX = 1
5 BIRTH_YEAR_INDEX = 2
6 DEATH_YEAR_INDEX = 3
7
8 # Each value in the marriages dictionary is a list.
9 # These are the indexes of the elements in those lists.
10 HUSBAND_KEY_INDEX = 0
11 WIFE_KEY_INDEX = 1
12 WEDDING_YEAR_INDEX = 2
13
14
15 def main():
16     people_dict = {
17         # Each item in the people dictionary is a key value pair.
18         # Each key is a unique identifier that begins with the
19         # letter "P". Each value is a list of data about a person.
20         # Each item in the dictionary is in this format:
21         # person_key: [name, gender, birth_year, death_year]
22         "P143": ["Lola Park", "F", 1663, 1706],
23         "P338": ["Savanna Foster", "F", 1674, 1723],
24         "P201": ["Tiffany Hughes", "F", 1689, 1747],
25         "P203": ["Ignacio Torres", "M", 1693, 1758],
26         "P128": ["Yasmin Li", "F", 1701, 1716],
27         "P342": ["Trent Ross", "M", 1705, 1757],
28         "P202": ["Samyukta Nguyen", "M", 1717, 1774],
29         "P132": ["Joel Johnson", "M", 1724, 1800],
30         "P445": ["Whitney Nelson", "F", 1757, 1823],
31         "P318": ["Khalid Ali", "M", 1759, 1814],
32         "P317": ["Davina Patel", "F", 1775, 1860],
33         "P313": ["Enzo Ruiz", "M", 1782, 1782],
34         "P475": ["Lauren Smith", "F", 1800, 1802],
35         "P455": ["Lucas Ross", "M", 1800, 1853],
36         "P435": ["Jamal Gray", "M", 1810, 1831],
37         "P204": ["Fatima Soares", "F", 1812, 1898],
38         "P206": ["Ephraim Foster", "M", 1831, 1885],
39         "P500": ["Peter Price", "M", 1832, 1878],
40         "P207": ["Rosalina Jimenez", "F", 1875, 1956],
41         "P425": ["Rachel Johnson", "F", 1876, 1940],
42         "P121": ["Vanessa Bennet", "F", 1880, 1960],
43         "P152": ["Jose Castillo", "M", 1884, 1931],
44         "P205": ["Liam Myers", "M", 1902, 1950],
45         "P465": ["Isabella Lopez", "F", 1907, 1959],
46         "P168": ["Megan Anderson", "F", 1909, 1945]
47     }
48
49     marriages_dict = {
50         # Each item in the marriages dictionary is a key value pair.
51         # Each key is a unique identifier that begins with the
52         # letter "M". Each value is a list of data about a marriage.
53         # Each item in the dictionary is in this format:
54         # marriage_key: [husband_key, wife_key, wedding_year]
55         "M48": ["P203", "P201", 1711],
56         "M45": ["P342", "P338", 1722],
57         "M36": ["P203", "P201", 1724],
```

```

58         "M47": ["P202", "P445", 1774],
59         "M21": ["P132", "P445", 1775],
60         "M59": ["P132", "P317", 1792],
61         "M63": ["P318", "P445", 1804],
62         "M12": ["P318", "P317", 1808],
63         "M54": ["P435", "P204", 1830],
64         "M34": ["P455", "P204", 1853],
65         "M55": ["P500", "P317", 1859],
66         "M52": ["P206", "P204", 1875],
67         "M78": ["P152", "P121", 1905],
68         "M50": ["P152", "P425", 1917],
69         "M64": ["P205", "P465", 1925],
70         "M62": ["P152", "P207", 1925],
71         "M70": ["P152", "P168", 1928]
72     }
73
74     # Call the print_death_age function to print
75     # each person's name and age at death.
76     print_death_age(people_dict)
77
78     # Print a blank line.
79     print()
80
81     # Call the count_genders function to count
82     # and print the number of males and females.
83     count_genders(people_dict)
84
85     # Print a blank line.
86     print()
87
88     # Call the print_marriages function to print
89     # human readable data about the marriages.
90     print_marriages(marriages_dict, people_dict)
91
92
93 def print_death_age(people_dict):
94     """For each person in the people dictionary,
95     print the person's name and age at death.
96
97     Parameter
98         people_dict: a dictionary that contains data about people
99             Each item in the dictionary is in this format:
100                 person_key: [name, gender, birth_year, death_year]
101
102     Return: nothing
103     """
104     pass
105
106 def count_genders(people_dict):
107     """Count and print the number of males
108     and females in the people dictionary.
109
110     Parameter
111         people_dict: a dictionary that contains data about people
112             Each item in the dictionary is in this format:
113                 person_key: [name, gender, birth_year, death_year]
114
115     Return: nothing
116     """
117     pass
118

```

```
119 def print_marriages(marriages_dict, people_dict):
120     """For each marriage in the marriages dictionary, print
121     the husband's name, his age at wedding, the wedding year,
122     the wife's name, and her age at wedding.
123
124     Parameters
125         marriages_dict: a dictionary that contains data about
126             marriages. Each item in the dictionary is in this format:
127             marriage_key: [husband_key, wife_key, wedding_year]
128         people_dict: a dictionary that contains data about people
129             Each item in the dictionary is in this format:
130             person_key: [name, gender, birth_year, death_year]
131     Return: nothing
132     """
133     pass
134
135
136     # If this file was executed like this:
137     # > python teach_solution.py
138     # then call the main function. However, if this file
139     # was simply imported, then skip the call to main.
140     if __name__ == "__main__":
141         main()
```

lesson08/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 # Each value in the people dictionary is a list. These
4 # are the indexes of the elements in those lists.
5 NAME_INDEX = 0
6 GENDER_INDEX = 1
7 BIRTH_YEAR_INDEX = 2
8 DEATH_YEAR_INDEX = 3
9
10 # Each value in the marriages dictionary is a list.
11 # These are the indexes of the elements in those lists.
12 HUSBAND_KEY_INDEX = 0
13 WIFE_KEY_INDEX = 1
14 WEDDING_YEAR_INDEX = 2
15
16
17 def main():
18     people_dict = {
19         # Each item in the people dictionary is a key value pair.
20         # Each key is a unique identifier that begins with the
21         # letter "P". Each value is a list of data about a person.
22         # Each item in the dictionary is in this format:
23         # person_key: [name, gender, birth_year, death_year]
24         "P143": ["Lola Park", "F", 1663, 1706],
25         "P338": ["Savanna Foster", "F", 1674, 1723],
26         "P201": ["Tiffany Hughes", "F", 1689, 1747],
27         "P203": ["Ignacio Torres", "M", 1693, 1758],
28         "P128": ["Yasmin Li", "F", 1701, 1716],
29         "P342": ["Trent Ross", "M", 1705, 1757],
30         "P202": ["Samyukta Nguyen", "M", 1717, 1774],
31         "P132": ["Joel Johnson", "M", 1724, 1800],
32         "P445": ["Whitney Nelson", "F", 1757, 1823],
33         "P318": ["Khalid Ali", "M", 1759, 1814],
34         "P317": ["Davina Patel", "F", 1775, 1860],
35         "P313": ["Enzo Ruiz", "M", 1782, 1782],
36         "P475": ["Lauren Smith", "F", 1800, 1802],
37         "P455": ["Lucas Ross", "M", 1800, 1853],
38         "P435": ["Jamal Gray", "M", 1810, 1831],
39         "P204": ["Fatima Soares", "F", 1812, 1898],
40         "P206": ["Ephraim Foster", "M", 1831, 1885],
41         "P500": ["Peter Price", "M", 1832, 1878],
42         "P207": ["Rosalina Jimenez", "F", 1875, 1956],
43         "P425": ["Rachel Johnson", "F", 1876, 1940],
44         "P121": ["Vanessa Bennet", "F", 1880, 1960],
45         "P152": ["Jose Castillo", "M", 1884, 1931],
46         "P205": ["Liam Myers", "M", 1902, 1950],
47         "P465": ["Isabella Lopez", "F", 1907, 1959],
48         "P168": ["Megan Anderson", "F", 1909, 1945]
49     }
50
51     marriages_dict = {
52         # Each item in the marriages dictionary is a key value pair.
53         # Each key is a unique identifier that begins with the
54         # letter "M". Each value is a list of data about a marriage.
55         # Each item in the dictionary is in this format:
56         # marriage_key: [husband_key, wife_key, wedding_year]
57         "M48": ["P203", "P201", 1711],
```

```

58     "M45": ["P342", "P338", 1722],
59     "M36": ["P203", "P201", 1724],
60     "M47": ["P202", "P445", 1774],
61     "M21": ["P132", "P445", 1775],
62     "M59": ["P132", "P317", 1792],
63     "M63": ["P318", "P445", 1804],
64     "M12": ["P318", "P317", 1808],
65     "M54": ["P435", "P204", 1830],
66     "M34": ["P455", "P204", 1853],
67     "M55": ["P500", "P317", 1859],
68     "M52": ["P206", "P204", 1875],
69     "M78": ["P152", "P121", 1905],
70     "M50": ["P152", "P425", 1917],
71     "M64": ["P205", "P465", 1925],
72     "M62": ["P152", "P207", 1925],
73     "M70": ["P152", "P168", 1928]
74 }
75
76 # Call the print_death_age function to print
77 # each person's name and age at death.
78 print_death_age(people_dict)
79
80 # Print a blank line.
81 print()
82
83 # Call the count_genders function to count
84 # and print the number of males and females.
85 count_genders(people_dict)
86
87 # Print a blank line.
88 print()
89
90 # Call the print_marriages function to print
91 # human readable data about the marriages.
92 print_marriages(marriages_dict, people_dict)
93
94 # Print a blank line.
95 print()
96
97 count_marriages(marriages_dict, people_dict)
98
99
100 def print_death_age(people_dict):
101     """For each person in the people dictionary,
102     print the person's name and age at death.
103
104     Parameter
105         people_dict: a dictionary that contains data about people
106             Each item in the dictionary is in this format:
107                 person_key: [name, gender, birth_year, death_year]
108
109     Return: nothing
110     """
111     print("Ages at Death")
112
113     # For each person in the people dictionary, do the following:
114     for person_key, person_list in people_dict.items():
115         # Get the person's name, birth year, and death year.
116         name = person_list[NAME_INDEX]
117         birth_year = person_list[BIRTH_YEAR_INDEX]
118         death_year = person_list[DEATH_YEAR_INDEX]

```

```

119     # Compute the person's age at death.
120     death_age = death_year - birth_year
121
122     # Print the data about the person for the user to see.
123     print(name, death_age)
124
125
126 def count_genders(people_dict):
127     """Count and print the number of males
128     and females in the people dictionary.
129
130     Parameter
131         people_dict: a dictionary that contains data about people
132             Each item in the dictionary is in this format:
133                 person_key: [name, gender, birth_year, death_year]
134     Return: nothing
135     """
136     num_males = 0
137     for person_key, person_list in people_dict.items():
138         gender = person_list[GENDER_INDEX]
139         if gender == "M":
140             num_males += 1
141
142     num_people = len(people_dict)
143     num_females = num_people - num_males
144
145     print("Genders")
146     print(f"Number of males: {num_males}")
147     print(f"Number of females: {num_females}")
148
149
150 def print_marriages(marriages_dict, people_dict):
151     """For each marriage in the marriages dictionary, print
152     the husband's name, his age at wedding, the wedding year,
153     the wife's name, and her age at wedding.
154
155     Parameters
156         marriages_dict: a dictionary that contains data about
157             marriages. Each item in the dictionary is in this format:
158                 marriage_key: [husband_key, wife_key, wedding_year]
159         people_dict: a dictionary that contains data about people
160             Each item in the dictionary is in this format:
161                 person_key: [name, gender, birth_year, death_year]
162     Return: nothing
163     """
164     print("Marriages")
165
166     # For each marriage in the marriage dictionary, do the following:
167     for marriage_key, marriage_list in marriages_dict.items():
168
169         # Get the husband person key, wife person
170         # key, and year they were married.
171         husband_key = marriage_list[HUSBAND_KEY_INDEX]
172         wife_key = marriage_list[WIFE_KEY_INDEX]
173         wedding_year = marriage_list[WEDDING_YEAR_INDEX]
174
175         # Use the husband person key to get the husband's
176         # data from the people dictionary and then get the
177         # husband's name and birth year from his data.
178         husband_list = people_dict[husband_key]
179         husband_name = husband_list[NAME_INDEX]
```

```

180     husband_birth = husband_list[BIRTH_YEAR_INDEX]
181
182     # Calculate the husband's age when he married.
183     husband_age = wedding_year - husband_birth
184
185     # Use the wife person key to get the wife's data
186     # from the people dictionary and then get the
187     # wife's name and birth year from her data.
188     wife_list = people_dict[wife_key]
189     wife_name = wife_list[NAME_INDEX]
190     wife_birth = wife_list[BIRTH_YEAR_INDEX]
191
192     # Calculate the wife's age when she married.
193     wife_age = wedding_year - wife_birth
194
195     # Print the data about the marriage for the user to see.
196     print(f"{husband_name} {husband_age}" \
197           f" > {wedding_year} < {wife_name} {wife_age}")
198
199
200 def count_marriages(marriages_dict, people_dict):
201     """Count and print the number of times that each person married.
202
203     Parameters
204         marriages_dict: a dictionary that contains data about
205             marriages. Each item in the dictionary is in this format:
206                 marriage_key: [husband_key, wife_key, wedding_year]
207         people_dict: a dictionary that contains data about people
208             Each item in the dictionary is in this format:
209                 person_key: [name, gender, birth_year, death_year]
210     Return: nothing
211     """
212     print("Number of Marriages")
213
214     NUM_MARRIAGES_INDEX = 4
215
216     # Add an extra variable to each person_list. The variable will
217     # be used to count the number of marriages that each person had.
218     for person_key, person_list in people_dict.items():
219         person_list.append(0)
220
221     # For each marriage in the marriage dictionary,
222     # add one to the husband's number of marriages
223     # and add one to the wife's number of marriages.
224     for marriage_key, marriage_list in marriages_dict.items():
225
226         # Get the husband person key and the wife person key.
227         husband_key = marriage_list[HUSBAND_KEY_INDEX]
228         wife_key = marriage_list[WIFE_KEY_INDEX]
229
230         # Add one to the number of times
231         # the husband has been married.
232         husband_list = people_dict[husband_key]
233         husband_list[NUM_MARRIAGES_INDEX] += 1
234
235         # Add one to the number of times
236         # the wife has been married.
237         wife_list = people_dict[wife_key]
238         wife_list[NUM_MARRIAGES_INDEX] += 1
239
240     # For each person in the people dictionary, print the

```

```
241     # person's name and number of times that person married.
242     for person_key, person_list in people_dict.items():
243         name = person_list[NAME_INDEX]
244         num_marriages = person_list[NUM_MARRIAGES_INDEX]
245         print(f"{name} {num_marriages}")
246
247
248     # If this file was executed like this:
249     # > python teach_solution.py
250     # then call the main function. However, if this file
251     # was simply imported, then skip the call to main.
252     if __name__ == "__main__":
253         main()
```

08 Prove Assignment: Dictionaries

Purpose

Prove that you can write a Python program that uses a dictionary and lists.

Assignment

During this assignment, you will write and test the remaining parts of the molar mass calculator that you started writing in the previous lesson's prove milestone. When you are finished with this prove assignment, your `chemistry.py` program must contain at least four functions named as follows:

1. `main`
2. `create_periodic_table`
3. `parse_formula`
4. `compute_molar_mass`

Helpful Documentation

The [prove milestone of the previous lesson](#) explains how a molar mass calculator should work.

The [prepare content for this lesson](#) explains how to create and use a dictionary in a Python program.

The [prepare content for lesson 5](#) explains how to use `pytest`, `assert`, and `approx` to automatically verify that functions are correct. It also contains an [example test function](#) and links to additional documentation about `pytest`.

Steps

Do the following:

1. Change the compound list that is in your `make_periodic_table` function to a compound dictionary. Each item in the dictionary should have a chemical symbol as the key and the chemical name and atomic mass in a list as the value, like this:

```
periodic_table_dict = {  
    # symbol: [name, atomic_mass]  
    "Ac": ["Actinium", 227],
```

```

        "Ag": ["Silver", 107.8682],
        "Al": ["Aluminum", 26.9815386],
        ...
    }

```

2. Copy and paste the following Python code into your chemistry.py program. Be certain not to paste the code inside an existing function.

```

class FormulaError(ValueError):
    """FormulaError is the type of error that
    parse_formula will raise if a formula is invalid.
    """

    def parse_formula(formula, periodic_table_dict):
        """Convert a chemical formula for a molecule into a compound
        list that stores the quantity of atoms of each element
        in the molecule. For example, this function will convert
        "H2O" to [[["H", 2], ["O", 1]]] and
        "P04H2(CH2)12CH3" to [[["P", 1], ["O", 4], ["H", 29], ["C", 13]]]

        Parameters
        formula: a string that contains a chemical formula
        periodic_table_dict: the compound dictionary returned
            from make_periodic_table
        Return: a compound list that contains chemical symbols and
            quantities like this [[["Fe", 2], ["O", 3]]]
        """
        assert isinstance(formula, str), \
            "wrong data type for parameter formula; " \
            f"formula is a {type(formula)} but must be a string"
        assert isinstance(periodic_table_dict, dict), \
            "wrong data type for parameter periodic_table_dict; " \
            f"periodic_table_dict is a {type(periodic_table_dict)} " \
            "but must be a dictionary"

        def parse_quant(formula, index):
            quant = 1
            if index < len(formula) and formula[index].isdecimal():
                start = index
                index += 1
                while index < len(formula) and formula[index].isdecimal():
                    index += 1
                quant = int(formula[start:index])
            return quant, index

        def get_quant(elem_dict, symbol):
            return 0 if symbol not in elem_dict else elem_dict[symbol]

        def parse_r(formula, index, level):
            start_index = index
            start_level = level
            elem_dict = {}
            while index < len(formula):
                ch = formula[index]
                if ch == "(":
                    group_dict, index = parse_r(formula, index+1, level+1)
                    quant, index = parse_quant(formula, index)
                    for symbol in group_dict:
                        prev = get_quant(elem_dict, symbol)
                        curr = prev + group_dict[symbol] * quant
                        elem_dict[symbol] = curr
                index += 1
            return elem_dict, index

        elem_dict = parse_r(formula, 0, 0)
        return elem_dict

```

```

        elem_dict[symbol] = curr
    elif ch.isalpha():
        symbol = formula[index:index+2]
        if symbol in periodic_table_dict:
            index += 2
        else:
            symbol = formula[index:index+1]
            if symbol in periodic_table_dict:
                index += 1
            else:
                raise FormulaError("invalid formula, "
                                    f"unknown element symbol: {symbol}",
                                    formula, index)
        quant, index = parse_quant(formula, index)
        prev = get_quant(elem_dict, symbol)
        elem_dict[symbol] = prev + quant
    elif ch == ")":
        if level == 0:
            raise FormulaError("invalid formula, "
                                "unmatched close parenthesis",
                                formula, index)
        level -= 1
        index += 1
        break
    else:
        if ch.isdecimal():
            # Decimal digit not preceded by an
            # element symbol or close parenthesis
            message = "invalid formula"
        else:
            # Illegal character: [^()0-9a-zA-Z]
            message = "invalid formula, illegal character"
        raise FormulaError(message, formula, index)
    if level > 0 and level >= start_level:
        raise FormulaError("invalid formula, "
                            "unmatched open parenthesis",
                            formula, start_index - 1)
return elem_dict, index

# Return the compound list of element symbols and
# quantities. Each element in the compound list
# will be a list in this form: ["symbol", quantity]
elem_dict, _ = parse_r(formula, 0, 0)
return list(elem_dict.items())


# Indexes for inner lists in the periodic table
NAME_INDEX = 0
ATOMIC_MASS_INDEX = 1

# Indexes for inner lists in a symbol_quantity_list
SYMBOL_INDEX = 0
QUANTITY_INDEX = 1


def compute_molar_mass(symbol_quantity_list, periodic_table_dict):
    """Compute and return the total molar mass of all the
    elements listed in symbol_quantity_list.

    Parameters
    symbol_quantity_list is a compound list. Each small

```

```

list in symbol_quantity_list has this form:
["symbol", quantity].
periodic_table_dict is the compound dictionary returned
from make_periodic_table.
Return: the total molar mass of all the elements in
symbol_quantity_list.

For example, if symbol_quantity_list is [[{"H": 2}, {"O": 1}],
this function will calculate and return
atomic_mass("H") * 2 + atomic_mass("O") * 1
1.00794 * 2 + 15.9994 * 1
18.01528
"""
# For each list in the compound symbol_quantity_list:
# Separate the list into symbol and quantity.
# Get the atomic mass for the symbol from the dictionary.
# Multiply the atomic mass by the quantity.
# Add the product into the total molar mass.

# Return the total molar mass.
return

```

The code that you pasted includes a `FormulaError` class and a function named `parse_formula`. Both of them are complete and work correctly, and you should not change them. The `parse_formula` function converts a chemical formula for a molecule, such as "C13H16N2O2" (melatonin), into a compound list, such as `[{"C": 13}, {"H": 16}, {"N": 2}, {"O": 2}]`. In the code that you pasted, this compound list is known as a *symbol_quantity_list* because it contains the symbols of chemical elements and the quantity of each element that appears in a chemical formula.

3. The code that you pasted also includes the header and documentation string for a function named `compute_molar_mass`. Read the docstring and comments in the `compute_molar_mass` function and write the code for that function. Note that you can complete the `compute_molar_mass` function by writing ten or fewer lines of code.
4. Modify the `main` function in your `chemistry.py` program so that it does the following:

```

def main():
    # Get a chemical formula for a molecule from the user.

    # Get the mass of a chemical sample in grams from the user.

    # Call the make_periodic_table function and
    # store the periodic table in a variable.

    # Call the parse_formula function to convert the
    # chemical formula given by the user to a compound
    # list that stores element symbols and the quantity
    # of atoms of each element in the molecule.

    # Call the compute_molar_mass function to compute the
    # molar mass of the molecule from the compound list.

    # Compute the number of moles in the sample.

```

```
# Print the molar mass.  
# Print the number of moles.
```

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download the [test_chemistry_2.py](#) Python file and save it in the same folder where you saved your chemistry.py program. Run the test_chemistry_2.py file and ensure that all three of the test functions pass. If any of the test functions don't pass, there is a mistake in your chemistry.py program. Read the output from pytest, fix the mistake, and run the test_chemistry_2.py file again until the test functions pass.

```
> python test_chemistry_2.py  
===== test session starts =====  
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy  
rootdir: C:\Users\cse111\lesson07  
collected 3 items  
  
test_chemistry_2.py::test_make_periodic_table PASSED [ 33%]  
test_chemistry_2.py::test_parse_formula PASSED [ 66%]  
test_chemistry_2.py::test_compute_molar_mass PASSED [100%]  
  
===== 3 passed in 0.17s =====
```

2. Run your finished chemistry.py program. Enter the input shown below and ensure that your program prints the output shown below.

```
> python chemistry.py  
Enter the molecular formula of the sample: C13H18O2  
Enter the mass in grams of the sample: 5.04  
206.28082 grams/mole  
0.02443 moles
```

Exceeding the Requirements

If your program fulfills the requirements for this assignment as described in the previous prove milestone and the Assignment section above, your program will earn 93% of the possible points. In order to earn the remaining 7% of points, you will need to add one or more features to your program so that it exceeds the requirements. Here are a few suggestions for additional features that you could add to your program if you wish.

- Add a dictionary that contains known chemical formulas and their names. For example:

```
{}
```

```
known_molecules_dict = {  
    "Al2O3": "aluminum oxide",  
    "CH3OH": "methanol",  
    "C2H6O": "ethanol",  
    "C2H5OH": "ethanol",  
    "C3H8O": "isopropyl alcohol",  
    "C3H8": "propane",  
    "C4H10": "butane",  
    "C6H6": "benzene",  
    "C6H14": "hexane",  
    "C8H18": "octane",  
    "CH3(CH2)6CH3": "octane",  
    "C13H18O2": "ibuprofen",  
    "C13H16N2O2": "melatonin",  
    "Fe2O3": "iron oxide",  
    "FeS2": "iron pyrite",  
    "H2O": "water"  
}
```

Then write a function named `get_formula_name` with the following header and documentation string.

```
def get_formula_name(formula, known_molecules_dict):  
    """Try to find formula in the known_molecules_dict.  
    If formula is in the known_molecules_dict, return  
    the name of the chemical formula; otherwise return  
    "unknown compound".  
    """
```

Call the `get_formula_name` function from your `main` function and print the compound name for the user to see with the other output.

- Add the atomic number for each element to the compound dictionary of elements. The atomic number of an element is the number of protons in the nucleus of that element. Write a function named `sum_protons` with the following header and documentation string.

```
def sum_protons(symbol_quantity_list, periodic_table_dict):  
    """Compute and return the total number of protons in  
    all the elements listed in symbol_quantity_list.  
    """
```

Call the `sum_protons` function from your `main` function and print the number of protons for the user to see with the other output.

Submission

To submit your program, return to I-Learn and do these two things:

1. Upload your `chemistry.py` file for feedback.
2. Add a submission comment that specifies the grading category that best describes your program along with a one or two sentence justification for your choice. The grading criteria are:

- a. Some attempt made
- b. Developing but significantly deficient
- c. Slightly deficient
- d. Meets requirements
- e. Exceeds requirements

lesson08/test_chemistry_2.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from chemistry import make_periodic_table, \
4     parse_formula, compute_molar_mass, FormulaError
5 from pytest import approx
6 import pytest
7
8
9 # These are the indexes of the
10 # elements in the periodic table.
11 NAME_INDEX = 0
12 ATOMIC_MASS_INDEX = 1
13
14
15 def test_make_periodic_table():
16     """Verify that the make_periodic_table function works correctly.
17     Parameters: none
18     Return: nothing
19     """
20     # Call the make_periodic_table function and store the returned
21     # dictionary in a variable named periodic_table_dict.
22     periodic_table_dict = make_periodic_table()
23
24     # Verify that the make_periodic_table function returns a dictionary
25     assert isinstance(periodic_table_dict, dict), \
26         "make_periodic_table function must return a dictionary: " \
27         f" expected a dictionary but found a {type(periodic_table_dict)}"
28
29     # Check each item in the periodic table dictionary.
30     check_element(periodic_table_dict, "Ac", ["Actinium", 227])
31     check_element(periodic_table_dict, "Ag", ["Silver", 107.8682])
32     check_element(periodic_table_dict, "Al", ["Aluminum", 26.9815386])
33     check_element(periodic_table_dict, "Ar", ["Argon", 39.948])
34     check_element(periodic_table_dict, "As", ["Arsenic", 74.9216])
35     check_element(periodic_table_dict, "At", ["Astatine", 210])
36     check_element(periodic_table_dict, "Au", ["Gold", 196.966569])
37     check_element(periodic_table_dict, "B", ["Boron", 10.811])
38     check_element(periodic_table_dict, "Ba", ["Barium", 137.327])
39     check_element(periodic_table_dict, "Be", ["Beryllium", 9.012182])
40     check_element(periodic_table_dict, "Bi", ["Bismuth", 208.9804])
41     check_element(periodic_table_dict, "Br", ["Bromine", 79.904])
42     check_element(periodic_table_dict, "C", ["Carbon", 12.0107])
43     check_element(periodic_table_dict, "Ca", ["Calcium", 40.078])
44     check_element(periodic_table_dict, "Cd", ["Cadmium", 112.411])
45     check_element(periodic_table_dict, "Ce", ["Cerium", 140.116])
46     check_element(periodic_table_dict, "Cl", ["Chlorine", 35.453])
47     check_element(periodic_table_dict, "Co", ["Cobalt", 58.933195])
48     check_element(periodic_table_dict, "Cr", ["Chromium", 51.9961])
49     check_element(periodic_table_dict, "Cs", ["Cesium", 132.9054519])
50     check_element(periodic_table_dict, "Cu", ["Copper", 63.546])
51     check_element(periodic_table_dict, "Dy", ["Dysprosium", 162.5])
52     check_element(periodic_table_dict, "Er", ["Erbium", 167.259])
53     check_element(periodic_table_dict, "Eu", ["Europium", 151.964])
54     check_element(periodic_table_dict, "F", ["Fluorine", 18.9984032])
55     check_element(periodic_table_dict, "Fe", ["Iron", 55.845])
56     check_element(periodic_table_dict, "Fr", ["Francium", 223])
57     check_element(periodic_table_dict, "Ga", ["Gallium", 69.723])
```

```

58 check_element(periodic_table_dict, "Gd", ["Gadolinium", 157.25])
59 check_element(periodic_table_dict, "Ge", ["Germanium", 72.64])
60 check_element(periodic_table_dict, "H", ["Hydrogen", 1.00794])
61 check_element(periodic_table_dict, "He", ["Helium", 4.002602])
62 check_element(periodic_table_dict, "Hf", ["Hafnium", 178.49])
63 check_element(periodic_table_dict, "Hg", ["Mercury", 200.59])
64 check_element(periodic_table_dict, "Ho", ["Holmium", 164.93032])
65 check_element(periodic_table_dict, "I", ["Iodine", 126.90447])
66 check_element(periodic_table_dict, "In", ["Indium", 114.818])
67 check_element(periodic_table_dict, "Ir", ["Iridium", 192.217])
68 check_element(periodic_table_dict, "K", ["Potassium", 39.0983])
69 check_element(periodic_table_dict, "Kr", ["Krypton", 83.798])
70 check_element(periodic_table_dict, "La", ["Lanthanum", 138.90547])
71 check_element(periodic_table_dict, "Li", ["Lithium", 6.941])
72 check_element(periodic_table_dict, "Lu", ["Lutetium", 174.9668])
73 check_element(periodic_table_dict, "Mg", ["Magnesium", 24.305])
74 check_element(periodic_table_dict, "Mn", ["Manganese", 54.938045])
75 check_element(periodic_table_dict, "Mo", ["Molybdenum", 95.96])
76 check_element(periodic_table_dict, "N", ["Nitrogen", 14.0067])
77 check_element(periodic_table_dict, "Na", ["Sodium", 22.98976928])
78 check_element(periodic_table_dict, "Nb", ["Niobium", 92.90638])
79 check_element(periodic_table_dict, "Nd", ["Neodymium", 144.242])
80 check_element(periodic_table_dict, "Ne", ["Neon", 20.1797])
81 check_element(periodic_table_dict, "Ni", ["Nickel", 58.6934])
82 check_element(periodic_table_dict, "Np", ["Neptunium", 237])
83 check_element(periodic_table_dict, "O", ["Oxygen", 15.9994])
84 check_element(periodic_table_dict, "Os", ["Osmium", 190.23])
85 check_element(periodic_table_dict, "P", ["Phosphorus", 30.973762])
86 check_element(periodic_table_dict, "Pa", ["Protactinium", 231.03588])
87 check_element(periodic_table_dict, "Pb", ["Lead", 207.2])
88 check_element(periodic_table_dict, "Pd", ["Palladium", 106.42])
89 check_element(periodic_table_dict, "Pm", ["Promethium", 145])
90 check_element(periodic_table_dict, "Po", ["Polonium", 209])
91 check_element(periodic_table_dict, "Pr", ["Praseodymium", 140.90765])
92 check_element(periodic_table_dict, "Pt", ["Platinum", 195.084])
93 check_element(periodic_table_dict, "Pu", ["Plutonium", 244])
94 check_element(periodic_table_dict, "Ra", ["Radium", 226])
95 check_element(periodic_table_dict, "Rb", ["Rubidium", 85.4678])
96 check_element(periodic_table_dict, "Re", ["Rhenium", 186.207])
97 check_element(periodic_table_dict, "Rh", ["Rhodium", 102.9055])
98 check_element(periodic_table_dict, "Rn", ["Radon", 222])
99 check_element(periodic_table_dict, "Ru", ["Ruthenium", 101.07])
100 check_element(periodic_table_dict, "S", ["Sulfur", 32.065])
101 check_element(periodic_table_dict, "Sb", ["Antimony", 121.76])
102 check_element(periodic_table_dict, "Sc", ["Scandium", 44.955912])
103 check_element(periodic_table_dict, "Se", ["Selenium", 78.96])
104 check_element(periodic_table_dict, "Si", ["Silicon", 28.0855])
105 check_element(periodic_table_dict, "Sm", ["Samarium", 150.36])
106 check_element(periodic_table_dict, "Sn", ["Tin", 118.71])
107 check_element(periodic_table_dict, "Sr", ["Strontium", 87.62])
108 check_element(periodic_table_dict, "Ta", ["Tantalum", 180.94788])
109 check_element(periodic_table_dict, "Tb", ["Terbium", 158.92535])
110 check_element(periodic_table_dict, "Tc", ["Technetium", 98])
111 check_element(periodic_table_dict, "Te", ["Tellurium", 127.6])
112 check_element(periodic_table_dict, "Th", ["Thorium", 232.03806])
113 check_element(periodic_table_dict, "Ti", ["Titanium", 47.867])
114 check_element(periodic_table_dict, "Tl", ["Thallium", 204.3833])
115 check_element(periodic_table_dict, "Tm", ["Thulium", 168.93421])
116 check_element(periodic_table_dict, "U", ["Uranium", 238.02891])
117 check_element(periodic_table_dict, "V", ["Vanadium", 50.9415])
118 check_element(periodic_table_dict, "W", ["Tungsten", 183.84])

```

```

119     check_element(periodic_table_dict, "Xe", ["Xenon", 131.293])
120     check_element(periodic_table_dict, "Y", ["Yttrium", 88.90585])
121     check_element(periodic_table_dict, "Yb", ["Ytterbium", 173.054])
122     check_element(periodic_table_dict, "Zn", ["Zinc", 65.38])
123     check_element(periodic_table_dict, "Zr", ["Zirconium", 91.224])
124
125
126 def check_element(periodic_table_dict, symbol, expected):
127     """Verify that the actual element that came from the
128     periodic_table_dict contains the same values as the
129     expected element.
130
131     Parameters
132         symbol: a symbol for a chemical element
133         expected: a list that contains the expected values for symbol
134     Return: nothing
135     """
136     # Verify that symbol is in the periodic table dictionary.
137     assert symbol in periodic_table_dict, \
138         f'{symbol}' is missing from the periodic table dictionary.'
139     actual = periodic_table_dict[symbol]
140
141     # Verify that the element's name is correct.
142     act_name = actual[NAME_INDEX]
143     exp_name = expected[NAME_INDEX]
144     assert act_name == exp_name, \
145         f'wrong name for {symbol}: ' \
146         f'expected {exp_name} but found {act_name}'
147
148     # Verify that the element's atomic mass is correct.
149     act_mass = actual[ATOMIC_MASS_INDEX]
150     exp_mass = expected[ATOMIC_MASS_INDEX]
151     assert act_mass == approx(exp_mass), \
152         f"wrong atomic mass for {exp_name}: " \
153         f"expected {exp_mass} but found {act_mass}"
154
155
156 def test_parse_formula():
157     """Verify that the parse_formula function works correctly.
158
159     Parameters: none
160     Return: nothing
161     """
162     periodic_table_dict = make_periodic_table()
163     assert isinstance(periodic_table_dict, dict), \
164         "make_periodic_table function must return a dictionary: " \
165         f" expected a dictionary but found a {type(periodic_table_dict)}"
166
167     sym_quant_list = parse_formula("H2O", periodic_table_dict)
168     assert isinstance(sym_quant_list, list), \
169         "parse_formula function must return a list: " \
170         f" expected a list but found a {type(sym_quant_list)}"
171
172     assert parse_formula("H2O", periodic_table_dict) \
173         == [("H",2), ("O",1)]
174     assert parse_formula("C6H6", periodic_table_dict) \
175         == [("C",6), ("H",6)]
176     assert parse_formula("(C2(NaCl)4H2)2C4Na", periodic_table_dict) \
177         == [("C",8), ("Na",9), ("Cl",8), ("H",4)]
178     with pytest.raises(FormulaError):
179         parse_formula("L", periodic_table_dict)

```

```

180     with pytest.raises(FormulaError):
181         parse_formula("4H", periodic_table_dict)
182     with pytest.raises(FormulaError):
183         parse_formula("H2L4", periodic_table_dict)
184     with pytest.raises(FormulaError):
185         parse_formula("-H", periodic_table_dict)
186     with pytest.raises(FormulaError):
187         parse_formula("(H2O", periodic_table_dict)
188     with pytest.raises(FormulaError):
189         parse_formula("H2)O3", periodic_table_dict)
190
191
192 def test_compute_molar_mass():
193     """Verify that the compute_molar_mass function works correctly.
194
195     Parameters: none
196     Return: nothing
197     """
198     periodic_table_dict = make_periodic_table()
199     assert isinstance(periodic_table_dict, dict), \
200         "make_periodic_table function must return a dictionary: " \
201         f" expected a dictionary but found a {type(periodic_table_dict)}"
202
203     molar_mass = compute_molar_mass([["O", 2]], periodic_table_dict)
204     assert isinstance(molar_mass, int) or isinstance(molar_mass, float),
205         "compute_molar_mass function must return a number: " \
206         f" expected a number but found a {type(molar_mass)}"
207
208     assert compute_molar_mass([], periodic_table_dict) == 0
209     assert compute_molar_mass([["O", 2]], periodic_table_dict) \
210         == approx(31.9988)
211     assert compute_molar_mass([["C", 6], ["H", 6]], periodic_table_dict) \
212         == approx(78.11184)
213     assert compute_molar_mass([["C", 13], ["H", 16], ["N", 2], ["O", 2]],
214         periodic_table_dict) == approx(232.27834)
215
216
217     # Call the main function that is part of pytest so that the
218     # computer will execute the test functions in this file.
219     pytest.main(["-v", "--tb=line", "-rN", __file__])

```

09 Prepare: Text Files

Most computers permanently store lots of data on devices such as hard drives, solid state drives, and thumb drives. The data that is stored on these devices is organized into files. Just as a human can write words on a paper, a computer can store words and other data in a file. During this lesson, you will learn how to write Python code that reads text from text files.

Concepts

Broadly speaking, there are two types of files: text files and binary files. A **text file** stores words and numbers as human readable text. A **binary file** stores pictures, diagrams, sounds, music, movies, and other media as numbers in a format that is not directly readable by humans.

Text Files

In order to read data from a text file, the file must exist on one of the computer's drives, and your program must do these three things:

1. Open the file for reading text
2. Read from the file, usually one line of text at a time
3. Close the file

The built-in [open function](#) opens a file for reading or writing. Here is an excerpt from the official documentation for the open function:

```
| open(filename, mode="rt")
```

Open a file and return a corresponding file object.

filename is the name of the file to be opened.

mode is an optional string that specifies the mode in which the file will be opened. It defaults to "rt" which means open for reading in text mode. Other common values are "wt" for writing a text file (truncating the file if it already exists), and "at" for appending to the end of a text file.

Example 1 contains a program that opens a text file named [plants.txt](#) for reading at line 26. At line 30 there is a for loop that reads the text in the file one line at a time and repeats the body of the for loop once for each line of text in the file. In the body of the for loop at lines 32–38, the code removes surrounding white space, if there is any, from each line of text and then stores each line of text in a list.

```

1 # Example 1
2
3 def main():
4     # Read the contents of a text file
5     # named plants.txt into a list.
6     text_list = read_list("plants.txt")
7
8     # Print the entire list.
9     print(text_list)
10
11
12 def read_list(filename):
13     """Read the contents of a text file into a list and
14     return the list. Each element in the list will contain
15     one line of text from the text file.
16
17     Parameter filename: the name of the text file to read
18     Return: a list of strings
19     """
20
21     # Create an empty list that will store
22     # the lines of text from the text file.
23     text_list = []
24
25     # Open the text file for reading and store a reference
26     # to the opened file in a variable named text_file.
27     with open(filename, "rt") as text_file:
28
29         # Read the contents of the text
30         # file one line at a time.
31         for line in text_file:
32
33             # Remove white space, if there is any,
34             # from the beginning and end of the line.
35             clean_line = line.strip()
36
37             # Append the clean line of text
38             # onto the end of the list.
39             text_list.append(clean_line)
40
41     # Return the list that contains the lines of text.
42     return text_list
43
44 # Call main to start this program.
45 if __name__ == "__main__":
46     main()

```

```

> python example_1.py
['baobab', 'kangaroo paw', 'eucalyptus', 'heliconia', 'tulip',
'chupasangre cactus', 'prickly pear cactus', 'ginkgo biloba']

```

After the body of a `for` loop that reads from a file, we can write a call to the `file.close` method. However, when calling the `open` function, most programmers use a `with` block as shown in example 1 at line 26 and nest the `for` loop inside the `with` block as shown at lines 30–38. When the `with` block ends, the computer will automatically close the file, so that the programmer doesn't have to write a call to the `file.close` method.

CSV Files

Many computer systems import and export data in CSV files. CSV is an acronym for comma separated values. A **CSV file** is a text file that contains tabular data with each row on a separate line of the file and each cell (column) separated by a comma. The following example shows the contents of a CSV file named [hymns.csv](#) that stores data about religious songs. Notice that the first row of the file contains column headings, the next four rows contain data about four hymns, and each row contains three columns separated by commas.

```
Title,Author,Composer
O Holy Night,John Dwight,Adolphe Adam
Away in a Manger,Anonymous,William Kirkpatrick
Joy to the World,Isaac Watts,George Handel
With Wondering Awe,Anonymous,Anonymous
```

Python has a standard [module named csv](#) that includes functionality to read from and write to CSV files. The program in example 2 shows how to open a CSV file and use the csv module to read the data and print it to a terminal window. In example 2 at line 8, there is a call to the Python built-in open function, which opens the hymns.csv file for reading. At line 12, the program creates a csv.reader object that will read from the hymns.csv file. Within the for loop at lines 16 and 17 the csv.reader reads and prints each row from the CSV file.

```
1 # Example 2
2
3 import csv
4
5 def main():
6     # Open the CSV file for reading and store a reference
7     # to the opened file in a variable named csv_file.
8     with open("hymns.csv", "rt") as csv_file:
9
10         # Use the csv module to create a reader object
11         # that will read from the opened CSV file.
12         reader = csv.reader(csv_file)
13
14         # Read the rows in the CSV file one row at a time.
15         # The reader object returns each row as a list.
16         for row_list in reader:
17             print(row_list)
18
19
20     # Call main to start this program.
21 if __name__ == "__main__":
22     main()
```

```
> python example_2.py
['Title', 'Author', 'Composer']
['O Holy Night', 'John Dwight', 'Adolphe Adam']
['Away in a Manger', 'Anonymous', 'William Kirkpatrick']
['Joy to the World', 'Isaac Watts', 'George Handel']
['With Wondering Awe', 'Anonymous', 'Anonymous']
```

When a `csv.reader` reads a row from a CSV file, the reader returns the row as a list of strings. The output from example 2 shows that a `csv.reader` returns a list of strings. In the output, notice the five lists of strings, (strings surrounded by square brackets [...]) that were printed by the print statement at [line 17](#). Notice also that the reader reads all the rows from a CSV file, including the first row, which contains column headings.

You might recall that in CSE 110, you wrote a program that reads from a CSV file without using a `csv.reader`. That program split each row of text from the CSV file using the string `split` method. Unfortunately, using the `split` method will not work for all CSV files. Consider the following `hymns.csv` file that contains rows for the hymns "Far, Far Way on Judea's Plains" and "Oh, Come, All Ye Faithful". Both of these hymns have commas in their titles. If we use the string `split` method to separate the columns in this CSV file, the hymn titles will be split. A `csv.reader` will correctly split rows in all valid CSV files.

```
Title,Author,Composer
"Far, Far Way on Judea's Plains",John Mcfarlane,John Mcfarlane
"Oh, Come, All Ye Faithful",John Wade,John Wade
"Christ the Lord is Risen Today",Charles Wesley,Anonymous
```

Processing Each Row in a CSV File

After reading each row from a CSV file, the `for` loop in the previous example simply prints the row list to a terminal window. Of course, a `for` loop can do much more than simply print each row. Consider the following CSV file named `dentists.csv` that stores data about dental offices. Notice that the first row of the file contains column headings, the next four rows contain data about four dental offices, and each row contains five columns separated by commas.

```
Company Name,Address,Phone Number,Employees,Patients
Eagle Rock Dental Care,556 Trejo Suite C,208-359-2224,7,1205
Apple Tree Dental,33 Winn Drive Suite 2,208-359-1500,10,1520
Rockhouse Dentistry,106 E 1st N,208-356-5600,12,1982
Cornerstone Family Dental,44 S Center Street,208-356-4240,8,1453
```

The program in example 3 processes each row in the `dentists.csv` file to determine which dental office has the most patients per employee. Notice that the first row of the `dentists.csv` file contains column headings. The headings contain no numbers and aren't needed for the calculations, so the program skips the first row by calling the built-in `next` function at [line 25](#).

```
1 # Example 3
2
3 import csv
4
5 # Indexes of some of the columns
6 # in the dentists.csv file.
7 COMPANY_NAME_INDEX = 0
8 NUM_EMPS_INDEX = 3
9 NUM_PATIENTS_INDEX = 4
```

```

10
11
12 def main():
13     # Open a file named dentists.csv and store a reference
14     # to the opened file in a variable named dentists_file.
15     with open("dentists.csv", "rt") as dentists_file:
16
17         # Use the csv module to create a reader
18         # object that will read from the opened file.
19         reader = csv.reader(dentists_file)
20
21         # The first row of the CSV file contains column
22         # headings and not data about a dental office,
23         # so this statement skips the first row of the
24         # CSV file.
25         next(reader)
26
27         running_max = 0
28         most_office = None
29
30         # Read each row in the CSV file one at a time.
31         # The reader object returns each row as a list.
32         for row_list in reader:
33
34             # For the current row, retrieve the
35             # values in columns 0, 3, and 4.
36             company = row_list[COMPANY_NAME_INDEX]
37             num_employees = int(row_list[NUM_EMPS_INDEX])
38             num_patients = int(row_list[NUM_PATIENTS_INDEX])
39
40             # Compute the number of patients per
41             # employee for the current dental office.
42             patients_per_emp = num_patients / num_employees
43
44             # If the current dental office has more
45             # patients per employee than the running
46             # maximum, assign running_max and most_office
47             # to be the current dental office.
48             if patients_per_emp > running_max:
49                 running_max = patients_per_emp
50                 most_office = company
51
52             # Print the results for the user to see.
53             print(f"{most_office} has {running_max:.1f}"
54                  " patients per employee")
55
56
57     # Call main to start this program.
58     if __name__ == "__main__":
59         main()

```

```

> python example_3.py
Cornerstone Family Dental has 181.6 patients per employee

```

Reading a CSV File into a Compound List

The program in example 3 reads and processes each row in a CSV file. That program needs to access the data in each row once only. If a program needs to access the contents of a CSV file multiple times, the program can read the contents of the file into a compound list and then access the data from the list. The program in example 4 contains a function named `read_compound_list` that reads the contents of a CSV file into a compound list.

```
1 # Example 4
2
3 import csv
4
5 def main():
6     # Read the contents of the dentists.csv file
7     # into a compound list.
8     dentists_list = read_compound_list("dentists.csv")
9
10    # Print the entire list.
11    print(dentists_list)
12
13
14 def read_compound_list(filename):
15     """Read the contents of a CSV file into a compound
16     list and return the list. Each element in the
17     compound list will be a small list that contains
18     the values from one row of the CSV file.
19
20     Parameter filename: the name of the CSV file to read
21     Return: a list of lists that contain strings
22     """
23
24     # Create an empty list that will
25     # store the data from the CSV file.
26     compound_list = []
27
28     # Open the CSV file for reading and store a reference
29     # to the opened file in a variable named csv_file.
30     with open(filename, "rt") as csv_file:
31
32         # Use the csv module to create a reader object
33         # that will read from the opened CSV file.
34         reader = csv.reader(csv_file)
35
36         # Read the rows in the CSV file one row at a time.
37         # The reader object returns each row as a list.
38         for row_list in reader:
39
40             # If the current row is not blank,
41             # append it to the compound_list.
42             if len(row_list) != 0:
43
44                 # Append one row from the CSV
45                 # file to the compound list.
46                 compound_list.append(row_list)
47
48     # Return the compound list.
49     return compound_list
50
51 # Call main to start this program.
```

```
52 if __name__ == "__main__":
53     main()
```

```
> python example_4.py
[['Company Name', 'Address', 'Phone Number', 'Employees',
'Patients'], ['Eagle Rock Dental Care', '556 Trejo Suite C',
'208-359-2224', '7', '1205'], ['Apple Tree Dental',
'33 Winn Drive Suite 2', '208-359-1500', '10', '1520'],
['Rockhouse Dentistry', '106 E 1st N', '208-356-5600', '12',
'1982'], ['Cornerstone Family Dental', '44 S Center Street',
'208-356-4240', '8', '1453']]
```

Reading a CSV File into a Compound Dictionary

If the values in one of the columns of a CSV file are unique, then a program can read the contents of a CSV file into a compound dictionary and then use the dictionary to quickly find data. Recall that each item in a dictionary is a key value pair. The values from the unique column in a CSV file will be the keys in the dictionary. The program in example 5 shows how to read the data from a CSV file into a compound dictionary. Notice in example 5, because of lines 9, 14, 58, and 62, that the program uses the dental office phone numbers as the keys in the dictionary.

```
1 # Example 5
2
3 import csv
4
5
6 def main():
7     # Index of the phone number column
8     # in the dentists.csv file.
9     PHONE_INDEX = 2
10
11    # Read the contents of the dentists.csv into a
12    # compound dictionary named dentists_dict. Use
13    # the phone numbers as the keys in the dictionary.
14    dentists_dict = read_dict("dentists.csv", PHONE_INDEX)
15
16    # Print the dentists compound dictionary.
17    print(dentists_dict)
18
19
20 def read_dict(filename, key_column_index):
21     """Read the contents of a CSV file into a compound
22     dictionary and return the dictionary.
23
24     Parameters
25         filename: the name of the CSV file to read.
26             key_column_index: the index of the column
27                 to use as the keys in the dictionary.
28     Return: a compound dictionary that contains
29             the contents of the CSV file.
30     """
31
32     # Create an empty dictionary that will
33     # store the data from the CSV file.
34     dictionary = {}
```

```

34
35     # Open the CSV file for reading and store a reference
36     # to the opened file in a variable named csv_file.
37     with open(filename, "rt") as csv_file:
38
39         # Use the csv module to create a reader object
40         # that will read from the opened CSV file.
41         reader = csv.reader(csv_file)
42
43         # The first row of the CSV file contains column
44         # headings and not data, so this statement skips
45         # the first row of the CSV file.
46         next(reader)
47
48         # Read the rows in the CSV file one row at a time.
49         # The reader object returns each row as a list.
50         for row_list in reader:
51
52             # If the current row is not blank, add the
53             # data from the current to the dictionary.
54             if len(row_list) != 0:
55
56                 # From the current row, retrieve the data
57                 # from the column that contains the key.
58                 key = row_list[key_column_index]
59
60                 # Store the data from the current
61                 # row into the dictionary.
62                 dictionary[key] = row_list
63
64             # Return the dictionary.
65             return dictionary
66
67
68     # Call main to start this program.
69     if __name__ == "__main__":
70         main()

```

```

> python example_5.py
{'208-359-2224': ['Eagle Rock Dental Care', '556 Trejo Suite...'],
 '208-359-1500': ['Apple Tree Dental', '33 Winn Drive Suite 2...'],
 '208-356-5600': ['Rockhouse Dentistry', '106 E 1st N', '208-...'],
 '208-356-4240': ['Cornerstone Family Dental', '44 S Center S...']}

```

Tutorials

If any of the concepts or topics in the previous section seem unfamiliar to you, reading these tutorials may help you understand the concepts better.

[Python "for" Loops](#)

[Reading and Writing CSV Files in Python](#)

Summary

A text file stores words and numbers as human readable text. During this lesson, you are learning how to write Python code to read from text files. To read from a text file, your program must first open the file by calling the built-in `open` function. You should write the code to open a file in a Python `with` block because the computer will automatically close the file when the `with` block ends, and you won't need to remember to write code to close the file.

A CSV file is a text file that contains rows and columns of data. CSV is an acronym that stands for comma separated values. Within each row in a CSV file, the data values are separated by commas. Python includes a standard module named `csv` that helps us easily write code to read from CSV files. Sometimes a program simply needs to use the values in a CSV file in calculations, so we write Python code to perform calculations for each row. Other times, we write Python code to read the contents of a CSV file into a compound list or compound dictionary.

09 Checkpoint: Text Files

Purpose

Check your understanding of text files and lists by writing a program that reads the contents of a text file into a list and then changes some of the values in the list.

Helpful Documentation

The [Text Files section](#) of the prepare content for this lesson contains example code that shows how to read a text file into a list.

The [Lists section](#) of the prepare content for lesson 7 contains example code that shows how to replace and remove elements in a list.

The [list.count method](#) counts and returns the number of times a value appears in a list.

Assignment

You must do the following to setup VS Code so that your program can read from a text file:

1. Download the [provinces.txt](#) file and save it in the same folder where you will save your Python program.
2. Using VS Code, open the folder where you saved the provinces.txt file by doing the following:
 - o On a computer running the **Mac OSX** operating system:
 - a. Click the "File" menu, then "Open..."
 - b. Find and select the folder where you saved the provinces.txt file.
 - c. Click the "Open" button.
 - o On a computer running the **Windows** operating system:
 - a. Click the "File" menu, then "Open Folder..."
 - b. Find and select the folder where you saved the provinces.txt file.
 - c. Click the "Select Folder" button.

Now that you have correctly setup VS Code so that your program can read the provinces.txt file, open that file in VS Code and examine it. Notice that the file contains

a list of names of Canadian Provinces. Notice also that the file contains "AB" several times which is an abbreviation for Alberta.

Write a Python program named `provinces.py` that reads the contents of `provinces.txt` into a list and then modifies the list. Your program must do the following:

1. Open the `provinces.txt` file for reading.
2. Read the contents of the file into a list where each line of text in the file is stored in a separate element in the list.
3. Print the entire list.
4. Remove the first element from the list.
5. Remove the last element from the list.
6. Replace all occurrences of "AB" in the list with "Alberta".
7. Count the number of elements that are "Alberta" and print that number.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and ensure that your program's output matches the output below. (Your program may wrap the province names differently.)

```
> python provinces.py
['Alberta', 'Ontario', 'Prince Edward Island', 'Ontario',
 'Quebec', 'Saskatchewan', 'AB', 'Nova Scotia', 'Alberta',
 'Northwest Territories', 'Saskatchewan', 'Nunavut',
 'Nova Scotia', 'Prince Edward Island', 'Alberta',
 'Nova Scotia', 'Nova Scotia', 'Prince Edward Island',
 'British Columbia', 'Ontario', 'Ontario',
 'Newfoundland and Labrador', 'Ontario', 'Ontario',
 'Saskatchewan', 'Nova Scotia', 'Prince Edward Island',
 'Saskatchewan', 'Ontario', 'Newfoundland and Labrador',
 'Ontario', 'British Columbia', 'Manitoba', 'Ontario',
 'Alberta', 'Saskatchewan', 'Ontario', 'Yukon', 'Ontario',
 'New Brunswick', 'British Columbia', 'Manitoba', 'Yukon',
 'British Columbia', 'Manitoba', 'Yukon',
 'Newfoundland and Labrador', 'Ontario', 'Yukon', 'Ontario',
 'AB', 'Nova Scotia', 'Newfoundland and Labrador', 'Yukon',
 'Nunavut', 'Northwest Territories', 'Nunavut', 'Yukon',
 'British Columbia', 'Ontario', 'AB', 'Saskatchewan',
 'Prince Edward Island', 'Saskatchewan',
 'Prince Edward Island', 'Alberta', 'Ontario', 'Alberta',
 'Manitoba', 'AB', 'British Columbia', 'Alberta']
```

Alberta occurs 9 times in the modified list.

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson09/provinces.txt

```
1 Alberta
2 Ontario
3 Prince Edward Island
4 Ontario
5 Quebec
6 Saskatchewan
7 AB
8 Nova Scotia
9 Alberta
10 Northwest Territories
11 Saskatchewan
12 Nunavut
13 Nova Scotia
14 Prince Edward Island
15 Alberta
16 Nova Scotia
17 Nova Scotia
18 Prince Edward Island
19 British Columbia
20 Ontario
21 Ontario
22 Newfoundland and Labrador
23 Ontario
24 Ontario
25 Saskatchewan
26 Nova Scotia
27 Prince Edward Island
28 Saskatchewan
29 Ontario
30 Newfoundland and Labrador
31 Ontario
32 British Columbia
33 Manitoba
34 Ontario
35 Alberta
36 Saskatchewan
37 Ontario
38 Yukon
39 Ontario
40 New Brunswick
41 British Columbia
42 Manitoba
43 Yukon
44 British Columbia
45 Manitoba
46 Yukon
47 Newfoundland and Labrador
48 Ontario
49 Yukon
50 Ontario
51 AB
52 Nova Scotia
53 Newfoundland and Labrador
54 Yukon
55 Nunavut
56 Northwest Territories
57 Nunavut
```

58	Yukon
59	British Columbia
60	Ontario
61	AB
62	Saskatchewan
63	Prince Edward Island
64	Saskatchewan
65	Prince Edward Island
66	Alberta
67	Ontario
68	Alberta
69	Manitoba
70	AB
71	British Columbia
72	Alberta

lesson09/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 Write a Python program named provinces.py that reads the contents
4 of provinces.txt into a list and then modifies the list.
5 """
6
7 def main():
8     # Read the contents of a text file named
9     # provinces.txt into a list named provinces_list.
10    provinces_list = read_list("provinces.txt")
11
12    # As a debugging aid, print the entire list.
13    print(provinces_list)
14
15    # Remove the first element from the list.
16    provinces_list.pop(0)
17    #print(provinces_list)
18
19    # Remove the last element from the list.
20    provinces_list.pop()
21    #print(provinces_list)
22
23    # Replace all occurrences of "AB" with "Alberta".
24    for i in range(len(provinces_list)):
25        if provinces_list[i] == "AB":
26            provinces_list[i] = "Alberta"
27    #print(provinces_list)
28
29    # Call the list.count method which will count the
30    # number of times that Alberta appears in the list.
31    count = provinces_list.count("Alberta")
32
33    print()
34    print(f"Alberta occurs {count} times in the modified list.")
35
36
37 def read_list(filename):
38     """Read the contents of a text file into a list
39     and return the list that contains the lines of text.
40
41     Parameter filename: the name of the text file to read
42     Return: a list of strings
43     """
44
45     # Create an empty list that will store
46     # the lines of text from the text file.
47     text_list = []
48
49     # Open the text file for reading and store a reference
50     # to the opened file in a variable named text_file.
51     with open(filename, "rt") as text_file:
52
53         # Read the contents of the text
54         # file one line at a time.
55         for line in text_file:
56
57             # Remove white space, if there is any,
```

```
58     # from the beginning and end of the line.
59     clean_line = line.strip()
60
61     # Append the clean line of text
62     # onto the end of the list.
63     text_list.append(clean_line)
64
65 # Return the list that contains the lines of text.
66 return text_list
67
68
69 # If this file was executed like this:
70 # > python teach_solution.py
71 # then call the main function. However, if this file
72 # was simply imported, then skip the call to main.
73 if __name__ == "__main__":
74     main()
```

Reading Files

You must do the following to setup VS Code so that your program can read from a text file:

1. Download the text file (.txt or .csv) and save it in the same folder where you will save your Python program.
2. Using VS Code, open the folder where you saved the text file by doing the following:
 - o On a computer running the **Mac OSX** operating system:
 - a. Click the "File" menu, then "Open..."
 - b. Find and select the folder where you saved the text file.
 - c. Click the "Open" button.
 - o On a computer running the **Windows** operating system:
 - a. Click the "File" menu, then "Open Folder..."
 - b. Find and select the folder where you saved the text file.
 - c. Click the "Select Folder" button.

09 Team Activity: CSV Files

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Problem Statement

A common task for many knowledge workers is to use a number, key, or ID to find information about a person. For example, a knowledge worker may use a phone number or e-mail address as a key to find (or look up) additional information about a customer. During this activity, your team will write a Python program that uses a student's I-Number to look up the student's name.

Helpful Documentation

- The Reading Files article explains how to setup VS Code so that your Python program can [read from a text file](#).
- The prepare content for this lesson includes a section about [reading a CSV file into a compound dictionary](#).
- The prepare content for lesson 8 explains how to [use dictionaries](#).
- Your program can call the string [replace method](#) to replace all occurrences of a character with another character or to remove all occurrences of a character by replacing them with the empty string ("").

Assignment

Download the [students.csv](#) file and save it in the same folder where you will save your Python program. Open the file in VS Code and examine it. Notice that the I-Numbers and names in the file are separated by a comma. Notice also that the I-Numbers are stored in the file without any dashes between the digits.

As a team, write a Python program named `students.py` that has at least two functions named `main` and `read_dict`. You must write the `read_dict` function with one of the following two headers and documentation strings. Choose the header that makes the most sense to you.

```
def read_dict(filename):
    """Read the contents of a CSV file into a
    dictionary and return the dictionary.

    Parameters
        filename: the name of the CSV file to read.
    Return: a dictionary that contains
            the contents of the CSV file.
    """
```

```
def read_dict(filename, key_column_index):
    """Read the contents of a CSV file into a compound
    dictionary and return the dictionary.

    Parameters
        filename: the name of the CSV file to read.
        key_column_index: the index of the column
            to use as the keys in the dictionary.
    Return: a compound dictionary that contains
            the contents of the CSV file.
    """
```

Core Requirements

Your program must do the following:

1. Open the `students.csv` file for reading, skip the first line of text in the file because it contains only headings, and read the other lines of the file into a dictionary. The program must store each student I-Number as a key and each name as a value in the dictionary.
2. Get an I-Number from the user, use the I-Number to find the corresponding student name in the dictionary, and print the name.
3. If a user enters an I-Number that doesn't exist in the dictionary, your program must print the message, "No such student" (without the quotes).

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add code to remove dashes from the I-Number that the user enters. This will allow the user to enter I-Numbers with dashes or without dashes and still allow the computer to search in the dictionary.
2. When a user enters an I-Number, your program should ensure it is a valid I-Number.

- a. If there are too few digits in the I-Number, your program should print, "Invalid I-Number: too few digits" (without the quotes).
 - b. If there are too many digits in the I-Number, your program should print, "Invalid I-Number: too many digits" (without the quotes).
 - c. If the given I-Number contains any characters besides digits and dashes, your program should output "Invalid I-Number" (without the quotes).
3. Add something or change something in your program that you think would make your program better, easier for the user, more elegant, or more fun. Be creative.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download the [test_students.py](#) Python file and save it in the same folder where you saved your students.py program. Run the test_students.py file and ensure that the test_read_dict function passes. If it doesn't pass, there is a mistake in your read_dict function. Read the output from pytest, fix the mistake, and run the test_students.py file again until the test function passes.

```
> python test_students.py
===== test session starts =====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson07
collected 1 item

test_students.py::test_students PASSED [100%]

===== 1 passed in 0.12s =====
```

2. Run your program and enter the inputs shown below. Ensure that your program's output matches the output below.

```
> python students.py
Please enter an I-Number (xxxxxxxxxx): 551234151
No such student

> python students.py
Please enter an I-Number (xxxxxxxxxx): 751766201
James Smith
```

3. Run your program and enter this I-Number as input: 00-115-2306 (including the dashes). Many users will want to enter I-Numbers with dashes. How should your program handle the dashes?
4. Run your program and enter an I-Number with too few digits or too many digits. How should your program handle these invalid I-Numbers?

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#). Please ***do not look at the sample solution*** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson09/students.csv

1	I-Number,Name
2	751766201,James Smith
3	751762102,Esther Einboden
4	052058203,Cassidy Benavidez
5	323021604,Joel Hatch
6	251041405,Brianna Ririe
7	001152306,Stefano Hisler
8	182706207,Hyeonbeom Park
9	124712708,Maren Thomas
10	212505409,Tyler Clark

lesson09/test_students.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from students import read_dict
4 from inspect import signature
5 from os import path
6 from tempfile import mktemp
7 import pytest
8
9
10 def test_read_dict():
11     """Verify that the read_dict function works correctly.
12     Parameters: none
13     Return: nothing
14     """
15     I_NUMBER_INDEX = 0
16
17     # Verify that the read_dict function uses its filename
18     # parameter by doing the following:
19     # 1. Get a filename for a file that doesn't exist.
20     # 2. Call the read_dict function with the filename.
21     # 3. Verify that the open function inside the read_dict
22     #     function raises a FileNotFoundError.
23     filename = mktemp(dir=".", prefix="not", suffix=".csv")
24     with pytest.raises(FileNotFoundError):
25         call_read_dict(filename, I_NUMBER_INDEX)
26         pytest.fail("read_dict function must use its filename parameter")
27
28     # Call the read_dict function which will read the students.csv
29     # file and create and return a dictionary.
30     filename = path.join(path.dirname(__file__), "students.csv")
31     students_dict = call_read_dict(filename, I_NUMBER_INDEX)
32
33     # Verify that the read_dict function returns a dictionary.
34     assert isinstance(students_dict, dict), \
35         "read_dict function must return a dictionary: " \
36         f" expected a dictionary but found a {type(students_dict)}"
37
38     # Verify that the students dictionary contains exactly nine items.
39     length = len(students_dict)
40     exp_len = 9
41     assert length == exp_len, \
42         "students dictionary has too" \
43         f" {'few' if length < exp_len else 'many'} items: " \
44         f" expected {exp_len} but found {length}"
45
46     # Verify the correctness of the nine items in the dictionary.
47     check_student(students_dict, "751766201", "James Smith")
48     check_student(students_dict, "751762102", "Esther Einboden")
49     check_student(students_dict, "052058203", "Cassidy Benavidez")
50     check_student(students_dict, "323021604", "Joel Hatch")
51     check_student(students_dict, "251041405", "Brianna Ririe")
52     check_student(students_dict, "001152306", "Stefano Hisler")
53     check_student(students_dict, "182706207", "Hyeonbeom Park")
54     check_student(students_dict, "124712708", "Maren Thomas")
55     check_student(students_dict, "212505409", "Tyler Clark")
```

```

58 def call_read_dict(filename, key_column_index):
59     """Call the read_dict function with the correct number of
60     parameters.
61     """
62     sig = signature(read_dict)
63     length = len(sig.parameters)
64     min_len = 1
65     max_len = 2
66     assert length == min_len or length == max_len, \
67         "The read_dict function contains too " \
68         f"{'few' if length < min_len else 'many'} parameters; " \
69         f"expected {min_len} or {max_len} parameters but found {length}"
70     if length == min_len:
71         dictionary = read_dict(filename)
72     else:
73         dictionary = read_dict(filename, key_column_index)
74     return dictionary
75
76
77 def check_student(students_dict, inumber, exp_name):
78     """Verify that the data for one student stored in the
79     students dictionary is correct.
80
81     Parameters
82         students_dict: a dictionary that contains student data
83         inumber: a student's I-Number that should be in the dictionary
84         exp_name: the student's expected name
85     Return: nothing
86     """
87
88     # Verify that inumber is in the students dictionary.
89     assert inumber in students_dict, \
90         f"'{inumber}' is missing from the students dictionary."
91
92     actual = students_dict[inumber]
93     assert isinstance(actual, str) or isinstance(actual, list), \
94         "Each value in the students dictionary must be either a string \
95         f'or a list. The value for {inumber} is of type {type(actual)} \
96         'which is not a string or a list."
97
98     if isinstance(actual, str):
99         # Verify that the student's name is correct.
100        assert actual == exp_name, \
101            f'Wrong name for "{inumber}"; ' \
102            f'expected {exp_name} but found {actual}'
103    else:
104        length = len(actual)
105        min_len = 1
106        max_len = 2
107        assert length == min_len or length == max_len, \
108            f"The value list for student {inumber} contains too " \
109            f"{'few' if length < min_len else 'many'} elements; " \
110            f"expected {min_len} or {max_len} elements but found {length}"
111
112        if length == min_len:
113            # Verify that the student's name is correct.
114            NAME_INDEX = 0
115            act_name = actual[NAME_INDEX]
116            assert act_name == exp_name, \
117                f'Wrong name for "{inumber}"; ' \
118                f'expected {exp_name} but found {act_name}'

```

```
119     # Verify that the student's I-Number is correct.
120     I_NUMBER_INDEX = 0
121     act_inum = actual[I_NUMBER_INDEX]
122     assert act_inum == inumber, \
123             'Inconsistent I-Numbers in the key and value. ' \
124             f'The key is {inumber} but {act_inum} is in ' \
125             'the corresponding value.'
126
127     # Verify that the student's name is correct.
128     NAME_INDEX = 1
129     act_name = actual[NAME_INDEX]
130     assert act_name == exp_name, \
131             f'Wrong name for "{inumber}"; ' \
132             f'expected {exp_name} but found {act_name}'
133
134
135     # Call the main function that is part of pytest so that the
136     # computer will execute the test functions in this file.
137     pytest.main(["-v", "--tb=line", "-rN", __file__])
```

lesson09/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2 """
3 A common task for many knowledge workers is to use a number, key,
4 or ID to look up information about a person. For example, a
5 knowledge worker may use a phone number or e-mail address as a key
6 to find (or look up) additional information about a customer.
7 During this activity, your team will write a Python program that
8 uses a student's I-Number to look up the student's name.
9 """
10 import csv
11
12
13
14 def main():
15     # The column headings and indexes.
16     I_NUMBER_INDEX = 0
17     NAME_INDEX = 1
18
19     # Read the contents of a CSV file named students.csv
20     # into a dictionary named students_dict. Use the I-Number
21     # as the key in the dictionary.
22     students_dict = read_dict("students.csv", I_NUMBER_INDEX)
23
24     # Get an I-Number from the user.
25     inumber = input("Please enter an I-Number (xx-xxx-xxxx): ")
26
27     # The I-Numbers are stored in the CSV file as digits only (without
28     # any dashes), so we remove all dashes from the user's input.
29     inumber = inumber.replace("-", "")
30
31     # Determine if the user input is formatted correctly.
32     if not inumber.isdigit():
33         print("Invalid character in I-Number")
34     else:
35         if len(inumber) < 9:
36             print("Invalid I-Number: too few digits")
37         elif len(inumber) > 9:
38             print("Invalid I-Number: too many digits")
39         else:
40             # The user input is a valid I-Number. Find
41             # the I-Number in the list of I-Numbers.
42             if inumber not in students_dict:
43                 print("No such student")
44             else:
45                 # Retrieve the student name that corresponds
46                 # to the I-Number that the user entered.
47                 value = students_dict[inumber]
48                 name = value[NAME_INDEX]
49
50                 # Print the student name.
51                 print(name)
52
53
54 def read_dict(filename, key_column_index):
55     """Read the contents of a CSV file into a compound
56     dictionary and return the dictionary.
57
```

```

58     Parameters
59         filename: the name of the CSV file to read.
60             key_column_index: the index of the column
61                 to use as the keys in the dictionary.
62             Return: a compound dictionary that contains
63                 the contents of the CSV file.
64 """
65 # Create an empty dictionary that will
66 # store the data from the CSV file.
67 dictionary = {}
68
69 # Open the CSV file for reading and store a reference
70 # to the opened file in a variable named csv_file.
71 with open(filename, "rt") as csv_file:
72
73     # Use the csv module to create a reader object
74     # that will read from the opened CSV file.
75     reader = csv.reader(csv_file)
76
77     # The first row of the CSV file contains column
78     # headings and not data, so this statement skips
79     # the first row of the CSV file.
80     next(reader)
81
82     # Read the rows in the CSV file one row at a time.
83     # The reader object returns each row as a list.
84     for row_list in reader:
85
86         # From the current row, retrieve the data
87         # from the column that contains the key.
88         key = row_list[key_column_index]
89
90         # Store the data from the current row
91         # into the dictionary.
92         dictionary[key] = row_list
93
94     # Return the dictionary.
95 return dictionary
96
97
98 # If this file was executed like this:
99 # > python teach_solution.py
100 # then call the main function. However, if this file
101 # was simply imported, then skip the call to main.
102 if __name__ == "__main__":
103     main()

```

09 Prove Milestone: Text Files

Purpose

Prove that you can write a Python program that reads CSV files and creates, populates, and uses a dictionary.

Problem Statement

A local grocery store subscribes to an online service that enables its customers to order groceries online. After a customer completes an order, the online service sends a CSV file that contains the customer's requests to the grocery store. The store needs you to write a program that reads the CSV file and prints to the terminal window a receipt that lists the purchased items and shows the subtotal, the sales tax amount, and the total.

Assignment

During this milestone, you will write half of a Python program named `receipt.py` that prints to the terminal window a receipt for a customer's grocery order. Specifically, by the end of this milestone, your program must contain at least these two functions:

1. `main`
2. `read_dict`

and must read and process these two CSV files:

- The `products.csv` file is a catalog of all the products that the grocery store sells.
- The `request.csv` file contains the items ordered by a customer.

Helpful Documentation

The Reading Files article explains how to setup VS Code so that your Python program can [read from a text file](#).

The prepare content for this lesson shows how to read the contents of a [CSV file into a compound dictionary](#) and how to read and [process a CSV file](#) without storing it in a dictionary.

The prepare content for lesson 8 explains how to [find an item in a dictionary](#).

The video titled [How to Read and Use a Dictionary](#) (34 minutes) shows a BYU-Idaho faculty member solving a problem that is similar to this prove assignment.

Steps

Do the following:

1. Download both of these files: [products.csv](#) and [request.csv](#) and save them in the same folder where you will save your Python program.
2. Open the products.csv file in VS Code and examine it. Notice that each row in this file contains three values separated by commas: a product number, a product name, and a retail price. Also, notice that each product number in the products.csv file is unique. This means that your program can read the products.csv file into a dictionary and use the product numbers as keys in the dictionary.
3. In VS Code, create a new file and save it as receipt.py in the same folder where you saved the products.csv and request.csv files.
4. In receipt.py, write a function named read_dict that will open a CSV file for reading and use a csv.reader to read each row and populate a compound dictionary with the contents of the products.csv file. The read_dict function must have this header and documentation string:

```
def read_dict(filename, key_column_index):
    """Read the contents of a CSV file into a compound
    dictionary and return the dictionary.

    Parameters
        filename: the name of the CSV file to read.
        key_column_index: the index of the column
            to use as the keys in the dictionary.
    Return: a compound dictionary that contains
        the contents of the CSV file.
    """

```

Recall that each item in a dictionary has a key and a value. Each item in the *products* dictionary must have a product number as the key and a list with the product number, product name, and price as the value as shown in the following table.

Products	
Key	Value
"D150"	["D150", "1 gallon milk", 2.85]
"D083"	["D083", "1 cup yogurt", 0.75]
"P143"	["P143", "1 lb baby carrots", 1.39]

Products	
Key	Value
"W231"	["W231", "32 oz granola", 3.21]
"W112"	["W112", "wheat bread", 2.55]
"C013"	["C013", "twix candy bar", 0.85]
:	:

5. Open the `request.csv` file in VS Code and examine it. Notice that each row contains only two values, a product number and a quantity. Notice also that product number D083 appears twice in the file. It appears twice because the customer who created the order in the `request.csv` file added four yogurts to his order and then later added three more yogurts to his order. Because product numbers may appear multiple times in the `request.csv` file, your program must not read the contents of `request.csv` into a dictionary.
6. In your `receipt.py` program, write another function named `main` that does the following:
 - a. Calls the `read_dict` function and stores the compound dictionary in a variable named `products_dict`.
 - b. Prints the `products_dict`.
 - c. Opens the `request.csv` file for reading.
 - d. Contains a loop that reads and processes each row from the `request.csv` file. Within the body of the loop, your program must do the following for each row:
 - i. Use the requested product number to find the corresponding item in the `products_dict`.
 - ii. Print the product name, requested quantity, and product price.

Because product number D083 appears twice in the `request.csv` file, your program must not read the `request.csv` file into a dictionary. Recall that each key in a dictionary is unique. If your program reads the `request.csv` file into a dictionary, when your program reads line 3 of the `request.csv` file, your program will put a request for four yogurts into the dictionary. Then when your program reads line 6 of the `request.csv` file, your program will replace the request for four yogurts with a request for three yogurts. In other words, if your program reads the `request.csv` file into a dictionary, your program will think that the customer ordered only three yogurts instead of the seven (4 + 3) that he ordered. Therefore, your program must not read the `request.csv` file into a dictionary but should instead read and process each row similar to [example 3](#) in the prepare content for this lesson.

7. At the bottom of your `receipt.py` file, add a call to the `main` function. Be certain to protect the call to `main` with an `if` statement as taught in the [prepare content for lesson 5](#).

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download the [test_products.py](#) file and save it in the same folder where you saved your receipt.py program. Run the test_products.py file and ensure that the test_read_dict function passes. If it doesn't pass, there is a mistake in your read_dict function. Read the output from pytest, fix the mistake, and run the test_products.py file again until the test function passes.

```
> python test_products.py
=====
platform win32--Python 3.8.6, pytest-6.1.2, py-1.9.0, pluggy
rootdir: C:\Users\cse111\lesson09
collected 1 item

test_products.py::test_read_dict PASSED [100%]

=====
1 passed in 0.12s =====
```

2. Run your program and verify that it prints the *products* dictionary and requested items as shown in the sample output below.

```
> python receipt.py

All Products
{'D150': ['D150', '1 gallon milk', '2.85'], 'D083': ['D083',
'1 cup yogurt', '0.75'], 'D215': ['D215', '1 lb cheddar
cheese', '3.35'], 'P019': ['P019', 'iceberg lettuce',
'1.15'], 'P020': ['P020', 'green leaf lettuce', '1.79'],
'P021': ['P021', 'butterhead lettuce', '1.83'], 'P025':
['P025', '8 oz arugula', '2.19'], 'P143': ['P143', '1 lb
baby carrots', '1.39'], 'W231': ['W231', '32 oz granola',
'3.21'], 'W112': ['W112', 'wheat bread', '2.55'], 'C013':
['C013', 'twix candy bar', '0.85'], 'H001': ['H001', '8
rolls toilet tissue', '6.45'], 'H014': ['H014', 'facial
tissue', '2.49'], 'H020': ['H020', 'aluminum foil', '2.39'],
'H021': ['H021', '12 oz dish soap', '3.19'], 'H025':
['H025', 'toilet cleaner', '4.50']}
```



```
Requested Items
wheat bread: 2 @ 2.55
1 cup yogurt: 4 @ 0.75
32 oz granola: 1 @ 3.21
twix candy bar: 2 @ 0.85
1 cup yogurt: 3 @ 0.75
```

Submission

On or before the due date, return to I-Learn and report your progress on this milestone.

lesson09/products.csv

	Product #,Name,Price
1	D150,1 gallon milk,2.85
2	D083,1 cup yogurt,0.75
3	D215,1 lb cheddar cheese,3.35
4	P019,iceberg lettuce,1.15
5	P020,green leaf lettuce,1.79
6	P021,butterhead lettuce,1.83
7	P025,8 oz arugula,2.19
8	P143,1 lb baby carrots,1.39
9	W231,32 oz granola,3.21
10	W112,wheat bread,2.55
11	C013,twix candy bar,0.85
12	H001,8 rolls toilet tissue,6.45
13	H014,facial tissue,2.49
14	H020,aluminum foil,2.39
15	H021,12 oz dish soap,3.19
16	H025,toilet cleaner,4.50

lesson09/request.csv

	Product #,Quantity
1	W112,2
2	D083,4
3	W231,1
4	C013,2
5	D083,3

lesson09/test_products.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from receipt import read_dict
4 from os import path
5 from tempfile import mktemp
6 from pytest import approx
7 import pytest
8
9
10 def test_read_dict():
11     """Verify that the read_dict function works correctly.
12     Parameters: none
13     Return: nothing
14     """
15     PRODUCT_NUM_INDEX = 0
16
17     # Verify that the read_dict function uses its filename
18     # parameter by doing the following:
19     # 1. Get a filename for a file that doesn't exist.
20     # 2. Call the read_dict function with the filename.
21     # 3. Verify that the open function inside the read_dict
22     #     function raises a FileNotFoundError.
23     filename = mktemp(dir=".", prefix="not", suffix=".csv")
24     with pytest.raises(FileNotFoundError):
25         read_dict(filename, PRODUCT_NUM_INDEX)
26         pytest.fail("read_dict function must use its filename parameter")
27
28     # Call the read_dict function and store the returned
29     # dictionary in a variable named products_dict.
30     filename = path.join(path.dirname(__file__), "products.csv")
31     products_dict = read_dict(filename, PRODUCT_NUM_INDEX)
32
33     # Verify that the read_dict function returns a dictionary.
34     assert isinstance(products_dict, dict), \
35         "read_dict function must return a dictionary: \"\n36         f\" expected a dictionary but found a {type(products_dict)}\""
37
38     # Verify that the products dictionary contains exactly 16 items.
39     length = len(products_dict)
40     exp_len = 16
41     assert length == exp_len, \
42         "products dictionary has too" \
43         "f\" {\'few\' if length < exp_len else \'many\'} items: \"\n44         f\" expected {exp_len} but found {length}\""
45
46     # Check each item in the products dictionary.
47     check_product(products_dict, "D150", ["1 gallon milk", 2.85])
48     check_product(products_dict, "D083", ["1 cup yogurt", 0.75])
49     check_product(products_dict, "D215", ["1 lb cheddar cheese", 3.35])
50     check_product(products_dict, "P019", ["iceberg lettuce", 1.15])
51     check_product(products_dict, "P020", ["green leaf lettuce", 1.79])
52     check_product(products_dict, "P021", ["butterhead lettuce", 1.83])
53     check_product(products_dict, "P025", ["8 oz arugula", 2.19])
54     check_product(products_dict, "P143", ["1 lb baby carrots", 1.39])
55     check_product(products_dict, "W231", ["32 oz granola", 3.21])
56     check_product(products_dict, "W112", ["wheat bread", 2.55])
57     check_product(products_dict, "C013", ["twix candy bar", 0.85])
```

```

58     check_product(products_dict, "H001", ["8 rolls toilet tissue", 6.45]
59     check_product(products_dict, "H014", ["facial tissue", 2.49])
60     check_product(products_dict, "H020", ["aluminum foil", 2.39])
61     check_product(products_dict, "H021", ["12 oz dish soap", 3.19])
62     check_product(products_dict, "H025", ["toilet cleaner", 4.50])
63
64
65 def check_product(products_dict, product_number, expected_value):
66     """Verify that the data for one product number stored in the
67     products dictionary is correct.
68
69     Parameters
70         products_dict: a dictionary that contains product data
71         product_number: the product number of the product that this
72             function will verify
73         expected_value: the data that should be in the products
74             dictionary for the product_number
75     Return: nothing
76     """
77     assert product_number in products_dict
78     actual_value = products_dict[product_number]
79     length = len(actual_value)
80     min_len = 2
81     max_len = 3
82     assert min_len <= length and length <= max_len, \
83         f"value list for product {product_number} contains too" \
84         f" {'few' if length < min_len else 'many'} elements:" \
85         f" expected {min_len} or {max_len} elements but found {length}"
86
87     if length == min_len:
88         NAME_INDEX = 0
89         PRICE_INDEX = 1
90     else:
91         NAME_INDEX = 1
92         PRICE_INDEX = 2
93
94     # Verify that the product's name is correct.
95     act_name = actual_value[NAME_INDEX]
96     exp_name = expected_value[0]
97     assert act_name == exp_name, \
98         f"wrong name for product {product_number}: " \
99         f"expected {exp_name} but found {act_name}"
100
101    # Verify that the product's price is correct.
102    act_price = actual_value[PRICE_INDEX]
103    if isinstance(act_price, str):
104        act_price = float(act_price)
105    exp_price = expected_value[1]
106    assert act_price == approx(exp_price), \
107        f"wrong price for product {product_number}: " \
108        f"expected {exp_price} but found {act_price}"
109
110
111    # Call the main function that is part of pytest so that the
112    # computer will execute the test functions in this file.
113    pytest.main(["-v", "--tb=line", "-rN", __file__])

```

10 Prepare: Handling Exceptions

Errors and exceptional situations sometimes occur while a program is running. Such errors include a program attempting to read from a file that doesn't exist, a connection error when connecting to a server on a network, data that cannot be found on a server, and calculations that produce undefined results. A well written program doesn't crash when an error occurs but instead handles errors in a graceful manner that may include adjusting to an error, printing an error message for the user to see, and saving an error message to a log file. During this lesson, you will learn to write code that handles errors that may occur while your Python program is running.

Videos

Watch these two videos from Microsoft about error handling.

[Error Handling Concepts](#) (13 minutes)

[Error Handling Demonstration](#) (4 minutes)

Concepts

Here are the Python programming concepts and topics that you should learn during this lesson.

What Is an Exception?

An **exception** is a relatively rare event that sometimes occurs while a Python program is running. For example, an exception occurs when a Python program tries to open a file for reading, and that file doesn't exist. There are many different [built-in exceptions](#) that may occur while a Python program is running.

When an exceptional event occurs, a Python function **raises** an exception which may be handled by code at another location in the executing Python program. The Python keyword to raise an exception is `raise`. Normally, you will not need to write code to raise an exception because the built-in functions, such as `open`, `int`, and `float`, will raise an exception when necessary.

How to Handle an Exception

The Python keywords to handle exceptions are `try`, `except`, `else`, and `finally`. The following example code contains the outline of a complete try-except-else-finally block. Read the code and its comments carefully to understand the correct syntax and organization of a try-except-else-finally block.

```
1 # Example 1
2
3 try:
4     # Write normal code here. This block must include
5     # code that falls into two groups:
6     # 1. Code that may cause an exception to be raised
7     # 2. Code that depends on the results of the code
8     #     in the first group
9 except ZeroDivisionError as zero_div_err:
10    # Code that the computer executes if the code in the try
11    # block caused a function to raise a ZeroDivisionError.
12 except ValueError as val_err:
13    # Code that the computer executes if the code in the
14    # try block caused a function to raise a ValueError.
15 except (TypeError, KeyError, IndexError) as error:
16    # Code that the computer executes if the code in the
17    # try block caused a function to raise a TypeError,
18    # KeyError, or IndexError.
19 except Exception as excep:
20    # Code that the computer executes if the code in the try
21    # block caused a function to raise any exception that
22    # was not handled by one of the previous except blocks.
23 except:
24    # Code that the computer executes if the code in the
25    # try block caused a function to raise anything that
26    # was not handled by one of the previous except blocks.
27 else:
28    # Code that the computer executes after the code
29    # in the try block if the code in the try block
30    # did not cause any function to raise an exception.
31 finally:
32    # Code that the computer executes after all the other
33    # code in try, except, and else blocks regardless of
34    # whether an exception was raised or not.
```

As shown in example 1 above, when we want to write code that will handle exceptions, we first write a `try` block, and we put into that `try` block the normal code that might cause an exception. Then we write `except` blocks to handle the exceptions. Each `except` block may handle one type of exception like the code at [line 12](#):

```
except ValueError as val_err:
```

Or each `except` block may handle several types of exceptions, like the code at [line 15](#):

```
except (TypeError, KeyError, IndexError) as error:
```

Or one `except` block may handle all possible types of exceptions, like the code at [line 19](#):

```
except Exception as excep:
```

Or a bare except block may handle anything that can be raised, including SystemExit, KeyboardInterrupt and GeneratorExit, like the code at [line 23](#):

```
except:
```

The Python programming language requires us to order except blocks from most specific at the top to least specific (most general) at the bottom. However, in most programs, it is a bad idea to write except blocks that are very general, including an except block that handles all possible exception types and a bare except block.

It is usually a bad idea to write an except block that handles all types of exceptions or a bare except block because such a block will handle SyntaxError. Normally, a program should not handle SyntaxError. Instead, a program should crash for a syntax error and print the line number where the syntax error occurred so that a programmer can find and fix the syntax error. As explained in the [prepare content](#) for lesson 6, syntax errors are caused by a programmer mistyping code and not by bad user input or missing files. A programmer should find and fix all syntax errors in a program long before the program is given to users, so there is no reason to handle syntax errors in an except block.

As shown at [line 27](#) in example 1 above, following the except blocks, a programmer may write an optional else block which the computer will execute if the try block does not raise any exceptions. It is uncommon to need to write code in the else block of try and except, and professional programmers almost never do it.

As shown at [line 31](#) in example 1 above, at the end of the exception handling code, a programmer may write an optional finally block. The finally block contains code that the computer executes after all the other code in the try, except, and else blocks regardless of whether an exception was raised or not. The code in the finally block usually contains "clean up" code that frees resources that the code in the try block used. For example, if the code in the try block opens a file, the code in the finally block could close that file. In CSE 111, you won't need to write a finally block.

Common Exception Types

There are many different types of [built-in exceptions](#) that may occur while a Python program is running. This section shows how seven types of exceptions may occur.

TypeError

The computer raises a [TypeError](#) when the code that calls a function passes an argument with the wrong data type. The code in example 2 attempts to pass a string of text to the round function which causes the computer to raise a TypeError as shown in the output below the code example.

```
# Example 2

def main():
    try:
        text = input("Please enter a number: ")
        integer = round(text)
        print(integer)

    except TypeError as type_err:
        print(type_err)

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

```
> python type_error.py
Please enter a number: 25.7
type str doesn't define __round__ method
```

ValueError

The computer raises a [ValueError](#) when the code that calls a function passes an argument with the correct data type but with an invalid value. A common event that causes the computer to raise a ValueError is when the int function or float function tries to convert a string to a number but the string contains characters that are not digits. The code in example 3 and its output show a ValueError.

```
# Example 3

def main():
    try:
        number = float(input("Please enter a number: "))
        print(number)

    except ValueError as val_err:
        print(val_err)

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

```
> python value_error.py
Please enter a number: 45.7u
could not convert string to float: '45.7u'
```

ZeroDivisionError

The computer raises a [ZeroDivisionError](#) when a program attempts to divide a number by zero (0) as shown in example 4 and its output.

```
# Example 4

def main():
    try:
```

```

players = int(input("Enter the number of players: "))
teams = int(input("Enter the number of teams: "))

players_per_team = players / teams

print(f"Each team should have {players_per_team} players")

except ZeroDivisionError as zero_div_err:
    print(zero_div_err)

if __name__ == "__main__":
    main()

```

```

> python zero_div_error.py
Enter the number of players: 20
Enter the number of teams: 0
division by zero

```

IndexError

Recall from [lesson 7](#) that each element in a list is stored at a unique index and that an index is always an integer. If we write code that tries to use an index that doesn't exist in a list, when the computer executes that code, the computer will raise an [IndexError](#). The program in example 5 creates a list that contains three surnames. Then the program attempts to change the surname at index 3. Of course, the list contains only three elements, and the index of the last element is 2, so the statement fails and causes the computer to raise an `IndexError`.

```

# Example 5

def main():
    try:
        # Create a list that contains three family names.
        surnames = ["Smith", "Lopez", "Marsh"]

        # Attempt to change the surname at index 3. Because
        # there are only three names in the surnames list and
        # therefore the last index is 2, this statement will
        # fail and cause the computer to raise an IndexError.
        surnames[3] = "Olsen"

    except IndexError as index_err:
        print(index_err)

if __name__ == "__main__":
    main()

```

```

> python index_error_write.py
list assignment index out of range

```

The program in example 6 is similar to example 5, and both programs cause the computer to raise an `IndexError`. The program in example 6 creates a list that contains three surnames. Then the program attempts to print the surname at index 3. Of course,

this statement fails because the list contains only three elements, and the index of the last element is 2.

```
# Example 6

def main():
    try:
        # Create a list that contains three family names.
        surnames = ["Smith", "Lopez", "Marsh"]

        # Attempt to print the surname at index 3. Because
        # there are only three names in the surnames list and
        # therefore the last index is 2, this statement will
        # fail and cause the computer to raise an IndexError.
        print(surnames[3])

    except IndexError as index_err:
        print(index_err)

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

```
> python index_error_read.py
list index out of range
```

KeyError

As shown in example 7, if we write code that attempts to find a key in a dictionary and that key doesn't exist in the dictionary, then the computer will raise a [KeyError](#).

```
# Example 7

def main():
    try:
        # Create a dictionary with student IDs as
        # the keys and student names as the values.
        students = {
            "42-039-4736": "Clint Huish",
            "61-315-0160": "Amelia Davis",
            "10-450-1203": "Ana Soares",
            "75-421-2310": "Abdul Ali",
            "07-103-5621": "Amelia Davis"
        }

        # Attempt to find the key "50-420-1021",
        # which is not in the dictionary. This will
        # cause the computer to raise a KeyError.
        name = students["50-420-1021"]

        print(name)

    except KeyError as key_err:
        print(type(key_err).__name__, key_err)

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

```
> python key_error.py  
KeyError '50-420-1021'
```

Of course, it is very unlikely that a programmer would write a program that tries to find a hard-coded key that is not in a dictionary. However, it is common for a user to enter a key that is not in a dictionary. This is why the programs in [examples 1 and 4](#) in the prepare content for lesson 8 include an `if` statement above the line of code that searches the dictionary, like this:

```
# Get a student ID from the user.  
id = input("Enter a student ID: ")  
  
# Check if the student ID is in the dictionary.  
if id in students:  
  
    # Find the student ID in the dictionary and  
    # retrieve the corresponding student name.  
    name = students[id]  
  
    # Print the student's name.  
    print(name)  
  
else:  
    print("No such student")
```

FileNotFoundException

If we write a call to the `open` function that attempts to open a file for reading and that file doesn't exist, the computer will raise a [FileNotFoundException](#). Example 8 contains code where such an error might occur.

```
# Example 8  
  
def main():  
    try:  
        with open("products.vcs", "rt") as products_file:  
            for line in products_file:  
                print(line)  
  
    except FileNotFoundError as not_found_err:  
        print(not_found_err)  
  
if __name__ == "__main__":  
    main()
```

```
> python file_not_found.py  
[Errno 2] No such file or directory: 'products.vcs'
```

PermissionError

Nearly all computer operating systems, such as Microsoft Windows, Mac OS X, and Linux, allow multiple people to use a single computer. Because people need to store private data in files on a computer, the operating systems implement file access permission rules. These rules help to prevent unauthorized access to files.

If we write a call to the `open` function that attempts to open a file and the person executing our program doesn't have permission to access the file, the computer will raise a [PermissionError](#). Example 9 contains code where such an error might occur.

```
# Example 9

def main():
    try:
        with open("contacts.csv", "rt") as contacts_file:
            for line in contacts_file:
                print(line)

    except PermissionError as perm_err:
        print(perm_err)

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

```
> python permission_error.py
[Errno 13] Permission denied: 'contacts.csv'
```

Example: Arithmetic

Example 10 contains a complete program with `except` blocks to handle two types of exceptions: `ValueError` and `ZeroDivisionError`.

```
# Example 10
"""

Sam, who is the owner of Sam's Sandwich Shop, requested
this program, which computes the number of sandwiches per
employee that were made in his restaurant in one day.
"""

def main():
    try:
        # Get the number of sandwiches made today and the
        # number of employees who worked today from the user.
        sandwiches = int(input("Number of sandwiches made today: "))
        employees = int(input("Number of employees who worked today: "))

        # Compute the number of sandwiches per employee
        # that were made today in the restaurant.
        sands_per_emp = sandwiches / employees

        # Print the results for the user to see.
        print(f"{sands_per_emp:.1f} sandwiches per employee")

    except ValueError as val_err:
```

```

print(f"Error: {val_err}")
print("You entered text that is not an integer. Please")
print("run the program again and enter an integer.")

except ZeroDivisionError as zero_div_err:
    print(f"Error: {zero_div_err}")
    print("You entered 0 for the number of employees.")
    print("Please run the program again and enter an integer")
    print("larger than 0 for the number of employees.")

# Call main to start this program.
if __name__ == "__main__":
    main()

```

```

> python example_10.py
Number of sandwiches that were made today: 35u
Error: invalid literal for int() with base 10: '35u'
You entered text that is not an integer. Please
run the program again and enter an integer.

> python example_10.py
Number of sandwiches that were made today: 350
Number of employees who worked today: 0
Error: division by zero
You entered 0 for the number of employees.
Please run the program again and enter an integer
larger than 0 for the number of employees.

> python example_10.py
Number of sandwiches that were made today: 350
Number of employees who worked today: 8
43.8 sandwiches per employee

```

Example: Reading from a File

The program in example 11 below handles exceptions that might occur when the program opens and reads from a file. This program contains only one try block, which begins at line 12 and includes all the regular code in the main function. This one try block has three except blocks at lines 49, 53, and 57 that handle FileNotFoundError, PermissionError, and ZeroDivisionError.

```

1 # Example 11
2
3 import csv
4
5 DATE_INDEX = 0
6 START_MILES_INDEX = 1
7 END_MILES_INDEX = 2
8 GALLONS_INDEX = 3
9
10
11 def main():
12     try:
13         # Open the fuel_usage.csv file.

```

```

14     filename = "fuel_usage.csv"
15     with open(filename, "rt") as usage_file:
16
17         # Use the standard csv module to get
18         # a reader object for the CSV file.
19         reader = csv.reader(usage_file)
20
21         # The first line of the CSV file contains
22         # headings and not fuel usage data, so this
23         # statement skips the first line of the file.
24         next(reader)
25
26         # Print headers for the three columns.
27         print("Date,Start,End,Gallons,Miles/Gallon")
28
29         # Process each row in the CSV file.
30         for row in reader:
31
32             # From the current row of the CSV file, get
33             # the date, the starting and ending odometer
34             # readings, and the number of gallons used.
35             date = row[DATE_INDEX]
36             start_miles = float(row[START_MILES_INDEX])
37             end_miles = float(row[END_MILES_INDEX])
38             gallons = float(row[GALLONS_INDEX])
39
40             # Call the miles_per_gallon function.
41             mpg = miles_per_gallon(
42                 start_miles, end_miles, gallons)
43
44             # Display the results for one row.
45             mpg = round(mpg, 1)
46             print(date, start_miles, end_miles,
47                   gallons, mpg, sep=",")
48
49     except FileNotFoundError as not_found_err:
50         print(f"Error: cannot open {filename}"
51               " because it doesn't exist.")
52
53     except PermissionError as perm_err:
54         print(f"Error: cannot read from {filename}"
55               " because you don't have permission.")
56
57     except ZeroDivisionError as zero_div_err:
58         print(f"Error: {filename} contains a"
59               " zero in the gallons column.")
60
61 def miles_per_gallon(start_miles, end_miles, gallons):
62     """Compute and return the average number of miles
63     that a vehicle traveled per gallon of fuel.
64
65     Parameters
66         start_miles: starting odometer reading in miles.
67         end_miles: ending odometer reading in miles.
68         gallons: amount of fuel used in U.S. gallons.
69     Return: miles per gallon
70     """
71     mpg = abs(end_miles - start_miles) / gallons
72     return mpg
73
74

```

```
75 # Call main to start this program.
76 if __name__ == "__main__":
77     main()
```

Validating User Input

To **validate user input** means to check user input to ensure it is in the correct format before using that input. The program in example 12 validates user input by handling exceptions. Notice in the get_float function at line 14 there is a try block. The try block is part of a loop that validates user input in the get_float function. Notice at line 44 that the except block handles `ValueError` which is the type of exception that the float function raises when it tries to convert text to a number but the text contains characters that are not numeric.

```
1 # Example 12
2
3 def main():
4     gender = input("Enter your gender (M or F): ")
5
6     weight = get_float("Enter your weight in kg: ", 20, 500)
7     height = get_float("Enter your height in cm: ", 60, 250)
8     age = get_float("Enter your age in years: ", 10, 120)
9
10    bmr = basal_metabolic_rate(gender, weight, height, age)
11    print(f"Your basal metabolic rate is {bmr} calories per day.")
12
13
14 def get_float(prompt, lower_bound, upper_bound):
15     """Get a number from the user, validate that the user
16     entered a number and not some other text, validate that
17     the number is between a lower and upper bound, and then
18     return the number. If the user enters an invalid number,
19     this function will prompt the user repeatedly until the
20     user enters a valid number.
21
22     Parameters
23         prompt: A string to display to the user.
24         lower_bound: The smallest number that the user may enter.
25         upper_bound: The largest number that the user may enter.
26     Return: The number entered by the user.
27     """
28
29     while True:
30         try:
31             text = input(prompt)
32             number = float(text)
33             if number < lower_bound:
34                 print(f"{number} is too small.")
35                 print("Please enter another number.")
36             elif number > upper_bound:
37                 print(f"{number} is too large.")
38                 print("Please enter another number.")
39             else:
40                 # If the computer gets to this line of code,
41                 # the user entered a valid number between
```

```

41             # lower_bound and upper_bound, so exit the loop.
42             break
43
44     except ValueError as val_err:
45         # The user entered at least one character that is
46         # not part of a floating point number, so print a
47         # message asking the user to enter a number.
48         print(f"{text} is not a number.")
49         print("Please enter a number.")
50
51     return number
52
53
54 def basal_metabolic_rate(gender, weight, height, age):
55     """Calculate and return a person's basal metabolic rate
56     in calories per day. weight must be in kilograms, height
57     must be in centimeters, and age must be in years.
58     """
59     if gender.upper() == "F":
60         bmr = 447.593 + 9.247 * weight \
61             + 3.098 * height - 4.330 * age
62     else:
63         bmr = 88.362 + 13.397 * weight \
64             + 4.799 * height - 5.677 * age
65     return bmr
66
67
68 # Call main to start this program.
69 if __name__ == "__main__":
70     main()

```

Tutorials

If the concepts above seem vague, these tutorials may clear some confusion for you:

Python Exceptions: [An Introduction](#)

Official Python [Errors and Exceptions tutorial](#)

The Most [Diabolical Python Antipattern](#)

Understanding the [Python Traceback](#)

The official Python [built-in exceptions](#) reference contains a list of all the built-in exceptions. It also includes the [class hierarchy](#) for the built-in exceptions that is helpful for ordering except blocks from most specific to most general.

Summary

Errors and exceptional situations sometimes occur while a program is running. When an exceptional situation occurs, a computer will raise an exception. With the try and except

keywords, you can write Python code that will handle exceptions. Put normal program code inside a `try` block and write an `except` block for each type of exception that you want your program to handle.

There are many types of exceptions in Python, but there are only seven types that your code will need to handle in CSE 111: `TypeError`, `ValueError`, `ZeroDivisionError`, `IndexError`, `KeyError`, `FileNotFoundException`, and `PermissionError`. When writing code that writes to or reads from a file, a programmer usually writes `try` and `except` blocks to handle `FileNotFoundException` and `PermissionError`. Also, when writing code that gets input from a user, a programmer usually writes `try` and `except` blocks to help validate the user's input.

lesson10/fuel_usage.csv

	Date	Start	End	Usage (gal)
1	2019-Sep-2	71010	71325	10.3
2	2019-Sep-9	71325	71641	10.7
3	2019-Sep-16	71641	71942	9.8
4	2019-Sep-23	71942	72298	11.2
5	2019-Sep-30	72298	72601	10.4

10 Checkpoint: Handling Exceptions

Purpose

Improve your understanding of how to handle exceptions in a Python program.

Assignment

Do the following:

1. Download and save the [accidents.csv](#) and [get_line.py](#) files in the same folder. `get_line.py` is a simple program that asks a user to input the name of a text file and a line number. Then it prints the text that is in the file on the requested line.
2. Open `get_line.py` in VS Code and notice the `try` block at line 10 and the five `except` blocks at lines 27–74, each handling a different type of exception.
3. Run the `get_line.py` program five times and enter the input shown in the Sample Run section below. For each of the first four times that you run the program, find the lines of code in `get_line.py` that handled the exception that was raised.

Sample Run

```
> python get_line.py
Enter the name of text file: 
FileNotFoundError: [Errno 2] No such file or directory: 'notofile.csv'
The file notofile.csv doesn't exist.
Run the program again and enter the name of an existing file.

> python get_line.py
Enter the name of text file: 
Enter a line number: 
ValueError: invalid literal for int() with base 10: 'hey'
You entered an invalid integer for the line number.
Run the program again and enter an integer for the line number.

> python get_line.py
Enter the name of text file: 
Enter a line number: 
IndexError: list index out of range
-300 is is a negative integer.
Run the program again and enter a line number between 1 and 7.

> python get_line.py
Enter the name of text file: 
Enter a line number: 
IndexError: list index out of range
75 is greater than the number of lines in accidents.csv.
There are only 7 lines in accidents.csv.
Run the program again and enter a line number between 1 and 7.

> python get_line.py
Enter the name of text file: 
Enter a line number: 
2012,33782,2362000,5615000,31006,3167,440,9420,6396,1262
```

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson10/get_line.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 """
4 This program asks the user for
5 1) the name of a text file
6 2) a line number
7 and prints the text from that line of the file.
8 """
9 def main():
10     try:
11         # Get a filename from the user.
12         filename = input("Enter the name of text file: ")
13
14         # Read the text file specified by the user into a list.
15         text_lines = read_list(filename)
16
17         # Get a line number from the user.
18         linenum = int(input("Enter a line number: "))
19
20         # Get the line that the user requested from the list.
21         line = text_lines[linenum - 1]
22
23         # Print the line that the user requested.
24         print()
25         print(line)
26
27     except FileNotFoundError as not_found_err:
28         # This code will be executed if the user enters
29         # the name of a file that doesn't exist.
30         print()
31         print(type(not_found_err).__name__, not_found_err, sep=": ")
32         print(f"The file {filename} doesn't exist.")
33         print("Run the program again and enter the" \
34             " name of an existing file.")
35
36     except PermissionError as perm_err:
37         # This code will be executed if the user enters the name
38         # of a file and doesn't have permission to read that file.
39         print()
40         print(type(perm_err).__name__, perm_err, sep=": ")
41         print(f"You don't have permission to read {filename}.")
42         print("Run the program again and enter the name" \
43             " of a file that you are allowed to read.")
44
45     except ValueError as val_err:
46         # This code will be executed if the user enters
47         # an invalid integer for the line number.
48         print()
49         print(type(val_err).__name__, val_err, sep=": ")
50         print("You entered an invalid integer for the line number.")
51         print("Run the program again and enter an integer for" \
52             " the line number.")
53
54     except IndexError as index_err:
55         # This code will be executed if the user enters a valid
56         # integer for the line number, but the integer is greater
57         # than the number of lines in the file.
```

```

58     print()
59     print(type(index_err).__name__, index_err, sep=": ")
60     length = len(text_lines)
61     if linenum < 0:
62         print(f"{linenum} is a negative integer.")
63     else:
64         print(f"{linenum} is greater than the number" \
65               f" of lines in {filename}.")
66         print(f"There are only {length} lines in {filename}.")
67         print(f"Run the program again and enter a line number" \
68               f" between 1 and {length}.")
69
70     except Exception as excep:
71         # This code will be executed if some
72         # other type of exception occurs.
73         print()
74         print(type(excep).__name__, excep, sep=": ")
75
76
77 def read_list(filename):
78     """Read the contents of a text file into a list
79     and return the list that contains the lines of text.
80
81     Parameter filename: the name of the text file to read
82     Return: a list of strings
83     """
84
85     # Create an empty list named text_lines.
86     text_lines = []
87
88     # Open the text file for reading and store a reference
89     # to the opened file in a variable named text_file.
90     with open(filename, "rt") as text_file:
91
92         # Read the contents of the text
93         # file one line at a time.
94         for line in text_file:
95
96             # Remove white space, if there is any,
97             # from the beginning and end of the line.
98             clean_line = line.strip()
99
100            # Append the clean line of text
101            # onto the end of the list.
102            text_lines.append(clean_line)
103
104    # Return the list that contains the lines of text.
105    return text_lines
106
107
108    # If this file was executed like this:
109    # > python teach_solution.py
110    # then call the main function. However, if this file
111    # was simply imported, then skip the call to main.
112    if __name__ == "__main__":
113        main()

```

lesson10/accidents.csv

	Year	Fatalities	Injuries	Crashes	Fatal Crashes	Distraction Affected	Fatality Rate
2	2010	32999	2239000	5419000	30296	3064	423
3	2011	32479	2217000	5338000	29867	3111	426
4	2012	33782	2362000	5615000	31006	3167	440
5	2013	32893	2313000	5687000	30202	2972	411
6	2014	32744	2338000	6064000	30056	3018	387
7	2015	35484	2443000	6296000	32538	3313	453
8	2016	37806	3061000	6821000	34748	3252	453
9	2017	37133	2746000	6452000	34247	2994	401

10 Team Activity: Handling Exceptions

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Problem Statement

Every year in the United States, thousands of people are killed and millions more are injured in traffic accidents on the nation's roads and highways. The U.S. National Highway Traffic Safety Administration (NHTSA) maintains [data about the accidents](#), including a list of factors contributing to each accident.

The CSV file `accidents.csv` contains summary data from the NHTSA about the accidents that occurred on U.S. roads and highways during the years 2010 through 2017. `accidents.py` is a Python program that uses the data to estimate how many lives would have been saved and injuries avoided if drivers had stopped dangerous behavior such as speeding and using cell phones while driving.

Unfortunately, `accidents.py` does not contain any code to validate the input that the user enters while the program is running. Add exception handling code to `accidents.py` to validate the user's input and print helpful error messages when the user enters incorrect input.

Helpful Documentation

The [prepare content](#) for this lesson explains how to wrap `try` and `except` around a normal block of code. It also includes an example that shows how to [validate user input](#) within a loop.

This [video about error handling](#) (17 minutes) shows a BYU-Idaho faculty member adding a `try` block and `except` blocks to a program to handle errors that might occur while the program is executing.

[open function](#), [FileNotFoundException](#), and [PermissionError](#)

[float function](#) and [ValueError](#)

[ZeroDivisionError](#) and [csv.Error](#)

Assignment

Do the following:

1. Download and save both [`accidents.csv`](#) and [`accidents.py`](#) in the same folder.
2. Open `accidents.csv` in VS Code or another editor and notice that it contains ten columns and eight rows of data regarding traffic accidents in the United States.
3. Open `accidents.py` in VS Code and notice that lines 21–22 contain code to ask the user for a filename and then open that file. It is possible that when the computer executes line 21, the user may enter the name of a file that does not exist or may enter the name of a file that the user does not have permission to read. These two situations cause the computer to raise a `FileNotFoundException` or `PermissionError` when the computer tries to open the file at line 22.
4. Notice that lines 25–26 of `accidents.py` contains code to prompt the user for a number. It is possible that while the computer is executing lines 25–26, the user may enter text that is not a number, for example: "4r0". This situation causes the computer to raise a `ValueError`. It is also possible that the user may enter a number that is too low or too high, for example: -15 or 701234932. In this situation, the computer does not raise an exception, but this situation is still a mistake that the program should alert the user about.
5. Notice that lines 45–55 of `accidents.py` contain code to process each line of the CSV file. It is possible that the CSV file is formatted incorrectly which would cause the computer to raise `csv.Error`. It is also possible that one of the rows in the CSV file contains zero (0) in the "Fatal Crashes" column. This would cause the computer to raise `ZeroDivisionError` at line 75 within the `estimate_reduction` function.

Core Requirements

1. Add exception handling code that prints a useful error message when the computer raises `FileNotFoundException` or `PermissionError`.
2. Add exception handling code that prints a useful error message when the computer raises `ValueError` while it is converting the percentage entered by the user to a float.
3. Add code that prints a useful error message when the user enters a percentage that is less than 0 or greater than 100. Hint: you could add an `if` statement immediately after the line of code that gets the percentage from the user.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add exception handling code to handle `ZeroDivisionError` that might be raised by the `estimate_reduction` function. This exception handling code should print a useful error message that includes the filename and line number of the line that contains zero (0) in the "Fatal Crashes" column.
2. Add exception handling code to handle `csv.Error` that might be raised by the `csv` module. This exception handling code should print a useful error message that includes the filename and line number of the line that is formatted incorrectly.
3. Test *all* of your error handling code, including the code that handles `ZeroDivisionError` and `PermissionError`. To test `ZeroDivisionError`, you could temporarily change one of the numbers in the "Fatal Crashes" column of `accidents.csv` to zero (0). To test `PermissionError`, you could create a CSV file or a simple text file and then use File Explorer or Finder to change the permissions on that file so that your account doesn't have permission to read that file.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run the `accidents.py` program and enter the inputs shown below. Verify that the exception handling code that you added to `accidents.py` prints error messages as shown below.

```

> python accidents.py
Name of file that contains NHTSA data: dentiaccs.vsc
[Errno 2] No such file or directory: 'dentiaccs.vsc'
Please choose a different file.

> python accidents.py
Name of file that contains NHTSA data: unreadable.csv
[Errno 13] Permission denied: 'unreadable.csv'
Please choose a different file.

> python accidents.py
Name of file that contains NHTSA data: accidents.csv
Percent reduction of texting while driving [0, 100]: 4r0
Error: could not convert string to float: '4r0'

> python accidents.py
Name of file that contains NHTSA data: accidents.csv
Percent reduction of texting while driving [0, 100]: -5
Error: -5.0 is too low. Please enter a different number.

> python accidents.py
Name of file that contains NHTSA data: accidents.csv
Percent reduction of texting while driving [0, 100]: 110
Error: 110.0 is too high. Please enter a different number.

> python accidents.py
Name of file that contains NHTSA data: accidents.csv
Percent reduction of texting while driving [0, 100]: 50

With a 50.0% reduction in using a cell
phone while driving, approximately this
number of injuries and deaths would have
been prevented in the USA.

Year, Injuries, Deaths
2010, 15631, 230
2011, 15811, 232
2012, 16759, 240
2013, 15738, 224
2014, 15052, 211
2015, 17006, 247
2016, 19953, 246
2017, 16077, 217

```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to this [sample solution](#). Please ***do not look at the sample solutions*** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson10/accidents.py

```
1 # Import the csv module so that it can be used
2 # to read from the accidents.csv file.
3 import csv
4
5
6 # Column numbers from the accidents.csv file.
7 YEAR_COLUMN = 0
8 FATALITIES_COLUMN = 1
9 INJURIES_COLUMN = 2
10 CRASHES_COLUMN = 3
11 FATAL_CRASHES_COLUMN = 4
12 DISTRACT_COLUMN = 5
13 PHONE_COLUMN = 6
14 SPEED_COLUMN = 7
15 DUI_COLUMN = 8
16 FATIGUE_COLUMN = 9
17
18
19 def main():
20     # Prompt the user for a filename and open that text file.
21     filename = input("Name of file that contains NHTSA data: ")
22     with open(filename, "rt") as text_file:
23
24         # Prompt the user for a percentage.
25         perc_reduc = float(input(
26             "Percent reduction of texting while driving [0, 100]: "))
27
28         print()
29         print(f"With a {perc_reduc}% reduction in using a cell",
30               "phone while driving, approximately this",
31               "number of injuries and deaths would have",
32               "been prevented in the USA.", sep="\n")
33         print()
34         print("Year, Injuries, Deaths")
35
36         # Use the csv module to create a reader
37         # object that will read from the opened file.
38         reader = csv.reader(text_file)
39
40         # The first line of the CSV file contains column headings
41         # and not a student's I-Number and name, so this statement
42         # skips the first line of the CSV file.
43         next(reader)
44
45         # Process each row in the CSV file.
46         for row in reader:
47             year = row[YEAR_COLUMN]
48
49             # Call the estimate_reduction function.
50             injur, fatal = estimate_reduction(
51                 row, PHONE_COLUMN, perc_reduc)
52
53             # Print the estimated reductions
54             # in injuries and fatalities.
55             print(year, injur, fatal, sep=", ")
```

```
58 def estimate_reduction(row, behavior_key, perc_reduc):
59     """Estimate and return the number of injuries and deaths that
60     would not have occurred on U.S. roads and highways if drivers
61     had reduced a dangerous behavior by a given percentage.
62
63     Parameters
64         row: a CSV row of data from the U.S. National Highway Traffic
65             Safety Administration (NHTSA)
66         behavior_key: heading from the CSV file for the dangerous
67             behavior that drivers could reduce
68         perc_reduc: percent that drivers could reduce a dangerous
69             behavior
70     Return: The number of injuries and deaths that may have been
71             prevented
72     """
73     behavior = int(row[behavior_key])
74     fatal_crashes = int(row[FATAL_CRASHES_COLUMN])
75     ratio = perc_reduc / 100 * behavior / fatal_crashes
76
77     fatalities = int(row[FATALITIES_COLUMN])
78     injuries = int(row[INJURIES_COLUMN])
79
80     reduc_fatal = int(round(fatalities * ratio, 0))
81     reduc_injur = int(round(injuries * ratio, 0))
82     return reduc_injur, reduc_fatal
83
84
85 # If this file was executed like this:
86 # > python accidents.py
87 # then call the main function. However, if this file
88 # was simply imported, then skip the call to main.
89 if __name__ == "__main__":
90     main()
```

lesson10/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 # Import the csv module so that it can be used
4 # to read from the accidents.csv file.
5 import csv
6
7
8 # Column numbers from the accidents.csv file.
9 YEAR_COLUMN = 0
10 FATALITIES_COLUMN = 1
11 INJURIES_COLUMN = 2
12 CRASHES_COLUMN = 3
13 FATAL_CRASHES_COLUMN = 4
14 DISTRACT_COLUMN = 5
15 PHONE_COLUMN = 6
16 SPEED_COLUMN = 7
17 DUI_COLUMN = 8
18 FATIGUE_COLUMN = 9
19
20
21 def main():
22     try:
23         # Prompt the user for a filename and open that text file.
24         filename = input("Name of file that contains NHTSA data: ")
25         with open(filename, "rt") as infile:
26
27             # Get a percentage from the user.
28             perc_reduc = get_float("Percent reduction of texting"
29                                   " while driving [0, 100]: ", 0, 100)
30
31             print()
32             print(f"With a {perc_reduc}% reduction in using a cell",
33                  "phone while driving, approximately this",
34                  "number of injuries and deaths would have",
35                  "been prevented in the USA.", sep="\n")
36             print()
37             print("Year, Injuries, Deaths")
38
39             # Use the csv module to create a reader
40             # object that will read from the opened file.
41             reader = csv.reader(infile)
42
43             # The first line of the CSV file contains column
44             # headings and not a student's I-Number and name, so
45             # this statement skips the first line of the CSV file.
46             next(reader)
47
48             # Process each row in the CSV file.
49             for row in reader:
50                 year = row[YEAR_COLUMN]
51
52                 # Call the estimate_reduction function.
53                 injur, fatal = estimate_reduction(
54                     row, PHONE_COLUMN, perc_reduc)
55
56                 # Print the estimated reductions
57                 # in injuries and fatalities.
```

```

58             print(year, injur, fatal, sep=", ")
59
60     except (FileNotFoundException, PermissionError) as error:
61         print(error)
62         print("Please choose a different file.")
63
64     except ValueError as val_err:
65         print("Error:", val_err)
66
67     except (csv.Error, KeyError) as error:
68         print(f"Error: line {reader.line_num} of {infile.name}"
69               " is formatted incorrectly.")
70
71     except ZeroDivisionError as zero_div_err:
72         print(f"Error: line {reader.line_num} of {infile.name}"
73               " contains 0 in the 'Fatal Crashes' or"
74               "'Cell Phone Use' column.")
75
76
77 def get_float(prompt, lower_bound, upper_bound):
78     """Prompt the user for a number and return the number as a float.
79
80     Parameters
81         prompt: A string to display to the user.
82         lower_bound: The lowest (smallest) number
83                     that the user may enter.
84         upper_bound: The highest (largest) number
85                     that the user may enter.
86     Return: The number that the user entered.
87     """
88     number = None
89     while number == None:
90         try:
91             number = float(input(prompt))
92             if number < lower_bound:
93                 print(f"Error: {number} is too low. \
94                      f Please enter a different number.")
95             number = None
96             elif number > upper_bound:
97                 print(f"Error: {number} is too high. \
98                      f Please enter a different number.")
99                 number = None
100            except ValueError as val_err:
101                print("Error:", val_err)
102            print()
103            return number
104
105
106 def estimate_reduction(row, behavior_key, perc_reduc):
107     """Estimate and return the number of injuries and deaths that
108     would not have occurred on U.S. roads and highways if drivers
109     had reduced a dangerous behavior by a given percentage.
110
111     Parameters
112         row: a CSV row of data from the U.S. National Highway
113             Traffic Safety Administration (NHTSA)
114         behavior_key: heading from the CSV file for the dangerous
115             behavior that drivers could reduce
116         perc_reduc: percent that drivers could reduce a dangerous
117             behavior
118     Return: The number of injuries and deaths that may have been

```

```
119     prevented
120 """
121 behavior = int(row[behavior_key])
122 fatal_crashes = int(row[FATAL_CRASHES_COLUMN])
123 ratio = perc_reduc / 100 * behavior / fatal_crashes
124
125 fatalities = int(row[FATALITIES_COLUMN])
126 injuries = int(row[INJURIES_COLUMN])
127
128 reduc_fatal = int(round(fatalities * ratio, 0))
129 reduc_injur = int(round(injuries * ratio, 0))
130 return reduc_injur, reduc_fatal
131
132
133 # If this file was executed like this:
134 # > python teach_solution.py
135 # then call the main function. However, if this file
136 # was simply imported, then skip the call to main.
137 if __name__ == "__main__":
138     main()
```

10 Prove Assignment: Handling Exceptions

Purpose

Prove that you can write a Python program that handles exceptions, including `FileNotFoundException`, `PermissionError`, and `KeyError`.

Problem Statement

A local grocery store subscribes to an online service that enables its customers to order groceries online. After a customer completes an order, the online service sends a CSV file that contains the customer's requests to the grocery store. The store needs you to write a program that reads the CSV file and prints to the terminal window a receipt that lists the purchased items and shows the subtotal, the sales tax amount, and the total.

Assignment

During the prove milestone for the previous lesson, you wrote the part of this program that reads and processes two CSV files, one named `products.csv` that contains a catalog of products and one named `request.csv` that contains a customer's order. During this prove assignment, you will add code to finish printing a receipt and to handle any exceptions that might occur while your program is running. Specifically, your program must do the following:

1. Print the store name at the top of the receipt.
2. Print the list of ordered items.
3. Sum and print the number of ordered items.
4. Sum and print the subtotal due.
5. Compute and print the sales tax amount. Use 6% as the sales tax rate.
6. Compute and print the total amount due.
7. Print a thank you message.
8. Get the current date and time from your computer's operating system and print the current date and time.
9. Include a `try` block and `except` blocks to handle `FileNotFoundException`, `PermissionError`, and `KeyError`.

Helpful Documentation

- The [prove milestone](#) of the previous lesson describes the two CSV files that your program must process.
- The [prepare content](#) for this lesson explains how to handle exceptions.
- The [datetime.now\(\) method](#) from the standard Python `datetime` module will get the current date and time from your computer's operating system. Here is an excerpt from the official documentation for the `datetime.now` method:

```
datetime.now(tz=None)
    Return the current local date and time.

    tz is optional, but if it is not None, it must be a tzinfo (time zone
    information) object
```

These two Microsoft videos explain how to use methods from the standard `datetime` module.

[Date data types](#) (8 minutes)

[Demonstration: Dates](#) (9 minutes)

The following Python code imports the `datetime` class from the `datetime` module and calls the `datetime.now` method to get the current date and time from a computer's operating system. Then it uses an f-string to format and print the current date and time.

```
1 # Import the datetime class from the datetime
2 # module so that it can be used in this program.
3 from datetime import datetime
4
5 # Call the now() method to get the current
6 # date and time as a datetime object from
7 # the computer's operating system.
8 current_date_and_time = datetime.now()
9
10 # Print the current day of the week and the current time.
11 print(f"{current_date_and_time:%A %I:%M %p}")
```

After the computer executes line 8 in the above code, the variable `current_date_and_time` will hold the current date and time. Within the f-string at line 11, the string sequences that begin with the percent symbol (%) are called format codes. The [format codes and their meaning](#) are listed in the official Python `datetime` reference. When executed, the previous example code will print the current date and time to the terminal window like this:

```
> python example_1.py
Tuesday 01:23 PM
```

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and verify that it prints a receipt formatted similarly to the one shown below. Your program must print the current date and time with exactly the same formatting as shown below. Also, verify that your program computes the number of items, subtotal, sales tax, and total as shown below.

```
> python receipt.py
Inkom Emporium

wheat bread: 2 @ 2.55
1 cup yogurt: 4 @ 0.75
32 oz granola: 1 @ 3.21
twix candy bar: 2 @ 0.85
1 cup yogurt: 3 @ 0.75

Number of Items: 12
Subtotal: 15.26
Sales Tax: 0.92
Total: 16.18

Thank you for shopping at the Inkom Emporium.
Wed Nov 4 05:10:30 2020
```

2. Verify that the except block to handle `KeyError` that you added to your program works correctly by doing the following:

- a. Temporarily add the following line to the end of your `requests.csv` file and then save the file.

```
R002,5
```

- b. Run your program again and verify that it prints an error message like the one shown below.

```
> python receipt.py
Error: unknown product ID in the request.csv file
'R002'
```

Hint: if you wrote an except block in your program to handle `KeyError` and added "R002,5" to the `requests.csv` file and saved the `requests.csv` file and ran your program but your program isn't raising a `KeyError`, then look in your program to see if you wrote an `if` statement before the statement that finds a value in the `products` dictionary. Look for code similar to this:

```
if prod_num in products_dict:
    prod_info_list = products_dict[prod_num]
    :
```

If your program contains an `if` statement similar to the one above, then the `if` statement is probably preventing your program from raising a `KeyError`.

Delete the `if` statement and unindent the lines of code inside the `if` statement and test your program again.

3. Verify that the `except` block to handle `FileNotFoundException` that you added to your program works correctly by doing the following:

- a. Temporarily delete or rename the `products.csv` file.
- b. Run your program again and verify that it prints an error message like the one shown below.

```
> python receipt.py
Error: missing file
[Errno 2] No such file or directory: 'products.csv'
```

Exceeding the Requirements

If your program fulfills the requirements for this assignment as described in the previous prove milestone and the Assignment section above, your program will earn 93% of the possible points. In order to earn the remaining 7% of points, you will need to add one or more features to your program so that it exceeds the requirements. Here are a few suggestions for additional features that you could add to your program if you wish.

- Write code to discount the product prices by 10% if today is Tuesday or Wednesday.
- Write code to discount the product prices by 10% if the current time of day is before 11:00 a.m.
- Write code to print a coupon at the bottom of the receipt. Write the code so that it will always print a coupon for one of the products ordered by the customer.
- Write code to print at the bottom of the receipt an invitation for the customer to complete an online survey.

Submission

To submit your program, return to I-Learn and do these two things:

1. Upload your program (the `.py` file) for feedback.
2. Add a submission comment that specifies the grading category that best describes your program along with a one or two sentence justification for your choice. The grading criteria are:
 - a. Some attempt made
 - b. Developing but significantly deficient

- c. Slightly deficient
- d. Meets requirements
- e. Exceeds requirements

11 Prepare: Functional Programming

A **paradigm** is a way of thinking or a way of perceiving the world. There are at least four main paradigms for programming a computer: procedural, declarative, functional, and object-oriented. During previous lessons in CSE 110 and 111, you used procedural programming. During this lesson, you will be introduced to functional programming.

Procedural programming is a programming paradigm that focuses on the process or the steps to accomplish a task. This is the type of programming that you did in CSE 110 and in previous lessons of CSE 111.

Declarative programming is a programming paradigm that does *not* focus on the process or steps to accomplish a task. Instead, with declarative programming, a programmer focuses on what she wants from the computer, or in other words, she focuses on the desired results. The SQL programming language is a good example of a declarative language. If you have ever written SQL code, then you have used declarative programming. When writing SQL code, a programmer writes code to tell the computer what she wants in the results but not the steps the computer must follow to get those results.

Functional programming is a programming paradigm that focuses on functions and avoids shared state, mutating state, and side effects. There are many techniques and concepts that are part of functional programming. However, in this lesson we will focus on just three, namely:

1. We can pass a function into another function.
2. A nested function is a function defined inside another function.
3. A lambda function is a small anonymous function.

Concepts

Here are the functional programming concepts that you should learn during this lesson.

Passing a Function into another Function

The Python programming language allows a programmer to pass a function as an argument into another function. A function that accepts other functions in its parameters is known as a **higher-order function**. Higher-order functions are often used to process the elements in a list. Before seeing an example of using a higher-order function to process a list, first consider the program in example 1 that doesn't use a higher-order function but instead uses a `for` loop to convert a list of temperatures from Fahrenheit to Celsius.

```
1  # Example 1
```

```

2
3 def main():
4     fahr_temps = [72, 65, 71, 75, 82, 87, 68]
5
6     # Print the Fahrenheit temperatures.
7     print(f"Fahrenheit: {fahr_temps}")
8
9     # Convert each Fahrenheit temperature to Celsius and store
10    # the Celsius temperatures in a list named cels_temps.
11    cels_temps = []
12    for fahr in fahr_temps:
13        cels = cels_from_fahr(fahr)
14        cels_temps.append(cels)
15
16    # Print the Celsius temperatures.
17    print(f"Celsius: {cels_temps}")
18
19
20 def cels_from_fahr(fahr):
21     """Convert a Fahrenheit temperature to
22     Celsius and return the Celsius temperature.
23     """
24     cels = (fahr - 32) * 5 / 9
25     return round(cels, 1)
26
27
28 # Call main to start this program.
29 if __name__ == "__main__":
30     main()

```

```

> python example_1.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]

```

At lines 12–14 in example 1, there is a `for` loop that converts each Fahrenheit temperature to Celsius and then appends the Celsius temperature onto a new list. Writing a `for` loop like this is the traditional way to process all the elements in a list and doesn't use higher-order functions.

Python includes a built-in higher-order function named `map` that will process all the elements in a list and return a new list that contains the results. The [map function](#) accepts a function and a list as arguments and contains a loop inside it, so that when a programmer calls the `map` function, he doesn't need to write a loop. The `map` function is a higher-order function because it accepts a function as an argument. Consider the program in example 2 that produces the same results as example 1.

```

1 # Example 2
2
3 def main():
4     fahr_temps = [72, 65, 71, 75, 82, 87, 68]
5
6     # Print the Fahrenheit temperatures.
7     print(f"Fahrenheit: {fahr_temps}")
8
9     # Convert each Fahrenheit temperature to Celsius and store
10    # the Celsius temperatures in a list named cels_temps.

```

```

11     cels_temps = list(map(cels_from_fahr, fahr_temps))
12
13     # Print the Celsius temperatures.
14     print(f"Celsius: {cels_temps}")
15
16
17 def cels_from_fahr(fahr):
18     """Convert a Fahrenheit temperature to
19     Celsius and return the Celsius temperature.
20     """
21     cels = (fahr - 32) * 5 / 9
22     return round(cels, 1)
23
24
25 # Call main to start this program.
26 if __name__ == "__main__":
27     main()

```

```

> python example_2.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]

```

Notice that example 2, doesn't contain a `for` loop. Instead, at line 11, it contains a call to the `map` function. Remember that the `map` function has a loop inside it, so that the programmer who calls `map`, doesn't have to write the loop. Notice also at line 11 that the first argument to the `map` function is the name of the `cels_from_fahr` function. In other words, at line 11, we are passing the `cels_from_fahr` function into the `map` function, so that `map` will call `cels_from_fahr` for each element in the `fahr_temps` list.

The `map` function is just one example of a higher-order function. Python also includes the built-in higher-order [sorted](#) and [filter](#) functions and several higher-order functions in the [functools module](#).

Nested Functions

The Python programming language allows a programmer to define nested functions. A **nested function** is a function that is defined inside another function and is useful when we wish to split a large function into smaller functions and the smaller functions will be called by the containing function only. The program in example 3 produces the same results as examples 1 and 2, but it uses a nested function. Notice in example 3 at lines 5–10 that the `cels_from_fahr` function is nested inside the `main` function.

```

1 # Example 3
2
3 def main():
4
5     def cels_from_fahr(fahr):
6         """Convert a Fahrenheit temperature to
7         Celsius and return the Celsius temperature.
8         """
9         cels = (fahr - 32) * 5 / 9
10        return round(cels, 1)

```

```

11     fahr_temps = [72, 65, 71, 75, 82, 87, 68]
12
13     # Print the Fahrenheit temperatures.
14     print(f"Fahrenheit: {fahr_temps}")
15
16     # Convert each Fahrenheit temperature to Celsius and store
17     # the Celsius temperatures in a list named cels_temps.
18     cels_temps = list(map(cels_from_fahr, fahr_temps))
19
20     # Print the Celsius temperatures.
21     print(f"Celsius: {cels_temps}")
22
23
24
25     # Call main to start this program.
26     if __name__ == "__main__":
27         main()

```

```

> python example_3.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]

```

Lambda Functions

A Python **lambda function** is a small anonymous function, meaning a small function without a name. A lambda function is always a small function because the Python language restricts a lambda function to just one expression. Consider the program in example 4 which is yet another example program that converts Fahrenheit temperatures to Celsius. Notice the lambda function at line 12 of example 4. It takes one parameter named *fahr* and computes and returns the corresponding Celsius temperature. At line 16, the lambda function is passed into the `map` function.

```

1  # Example 4
2
3  def main():
4      fahr_temps = [72, 65, 71, 75, 82, 87, 68]
5
6      # Print the Fahrenheit temperatures.
7      print(f"Fahrenheit: {fahr_temps}")
8
9      # Define a lambda function that converts
10     # a Fahrenheit temperature to Celsius and
11     # returns the Celsius temperature.
12     cels_from_fahr = lambda fahr: round((fahr - 32) * 5 / 9, 1)
13
14     # Convert each Fahrenheit temperature to Celsius and store
15     # the Celsius temperatures in a list named cels_temps.
16     cels_temps = list(map(cels_from_fahr, fahr_temps))
17
18     # Print the Celsius temperatures.
19     print(f"Celsius: {cels_temps}")
20
21
22     # Call main to start this program.

```

```
23 if __name__ == "__main__":
24     main()
```

```
> python example_4.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]
```

Some students are confused by the statement that a lambda function is an anonymous function (a function without a name). Looking at the lambda function in example 4 at [line 12](#), it appears that the lambda function is named *cels_from_fahr*. However, *cels_from_fahr* is the name of a variable, not the name of the lambda function. The lambda function has no name. This distinction may seem trivial until we see an example of an inline lambda function. Notice in the next example that the lambda function is defined inside the parentheses for the call to the `map` function.

```
# Convert each Fahrenheit temperature to Celsius and store
# the Celsius temperatures in a list named cels_temps.
cels_temps = list(map(
    lambda fahr: round((fahr - 32) * 5 / 9, 1),
    fahr_temps))
```

To write a lambda function write code that follows this template:

```
lambda param1, param2, ... paramN: expression
```

As shown in the template, type the keyword `lambda`, then parameters separated by commas, then a colon (:), and finally an expression that performs arithmetic, modifies a string, or computes something else.

In Python, every lambda function can be written as a regular Python function. For example, the lambda function in example 4 can be rewritten as the `cels_from_fahr` function in examples 1, 2, and 3.

Example - Map and Filter

The [checkpoint](#) for lesson 9 required you to write a program that replaced all the occurrences of "AB" in a list with the name "Alberta" and then counted how many times the name "Alberta" appeared in the list.

Example 5 contains a program that uses the `map` and `filter` functions to complete the requirements of [checkpoint 9](#). The example program works by doing the following:

1. Calling the `map` function at [line 42](#) to convert all elements that are "AB" to "Alberta."
 2. Calling the `filter` function at [line 55](#) to remove all elements that are not "Alberta."
 3. Calling the `len` function at [line 63](#) to count the number of elements that remain in the filtered list.
-

```

1 # Example 5
2
3 def main():
4     # Create a list that contains the names of Canadian provinces.
5     provinces_list = [
6         "Alberta", "Ontario", "Prince Edward Island", "Ontario",
7         "Quebec", "Saskatchewan", "AB", "Nova Scotia", "Alberta",
8         "Northwest Territories", "Saskatchewan", "Nunavut",
9         "Nova Scotia", "Prince Edward Island", "Alberta",
10        "Nova Scotia", "Prince Edward Island", "Saskatchewan",
11        "Nova Scotia", "Newfoundland and Labrador", "Ontario",
12        "Ontario", "Ontario", "British Columbia", "Ontario",
13        "Nova Scotia", "Prince Edward Island", "Saskatchewan",
14        "Newfoundland and Labrador", "Ontario", "Ontario",
15        "Manitoba", "British Columbia", "Ontario", "Alberta",
16        "Saskatchewan", "Ontario", "Manitoba", "Ontario",
17        "New Brunswick", "Yukon", "British Columbia", "Yukon",
18        "Newfoundland and Labrador", "Manitoba", "Ontario",
19        "Yukon", "British Columbia", "Yukon", "Ontario", "AB",
20        "Newfoundland and Labrador", "Nova Scotia", "Yukon",
21        "Northwest Territories", "Nunavut", "Yukon", "Nunavut",
22        "Ontario", "British Columbia", "AB", "Saskatchewan",
23        "Prince Edward Island", "Saskatchewan",
24        "Prince Edward Island", "Alberta", "Ontario", "Alberta",
25        "Manitoba", "AB", "British Columbia", "Alberta"
26    ]
27
28     # As a debugging aid, print the entire list.
29     print("Original list of provinces:")
30     print(provinces_list)
31     print()
32
33     # Define a nested function that converts AB to Alberta.
34     def alberta_from_ab(province_name):
35         if province_name == "AB":
36             province_name = "Alberta"
37         return province_name
38
39     # Replace all occurrences of "AB" with "Alberta" by
40     # calling the map function and passing the alberta_from_ab
41     # function and provinces_list into the map function.
42     new_list = list(map(alberta_from_ab, provinces_list))
43     print("List of provinces after AB was changed to Alberta:")
44     print(new_list)
45     print()
46
47     # Define a nested function that returns True if
48     # province_name is Alberta and returns False otherwise.
49     def is_alberta(province_name):
50         return province_name == "Alberta"
51
52     # Filter the new list to only those provinces that
53     # are "Alberta" by calling the filter function and
54     # passing the is_alberta function and new_list.
55     filtered_list = list(filter(is_alberta, new_list))
56     print("List filtered to Alberta only:")
57     print(filtered_list)
58     print()
59
60     # Because all the elements in filtered_list are
61     # "Alberta", we can count how many elements are

```

```

62     # "Alberta" by simply calling the len function.
63     count = len(filtered_list)
64
65     print(f"Alberta occurs {count} times in the modified list.")
66
67
68 # Call main to start this program.
69 if __name__ == "__main__":
70     main()

```

```
> python example_5.py
```

Original list of provinces:

```
['Alberta', 'Ontario', 'Prince Edward Island', 'Ontario', 'Quebec',
'Saskatchewan', 'AB', 'Nova Scotia', 'Alberta', 'Northwest
Territories', 'Saskatchewan', 'Nunavut', 'Nova Scotia', 'Prince Edward
Island', 'Alberta', 'Nova Scotia', 'Prince Edward Island',
'Saskatchewan', 'Nova Scotia', 'Newfoundland and Labrador', 'Ontario',
'Ontario', 'Ontario', 'British Columbia', 'Ontario', 'Nova Scotia',
'Prince Edward Island', 'Saskatchewan', 'Newfoundland and Labrador',
'Ontario', 'Ontario', 'Manitoba', 'British Columbia', 'Ontario',
'Alberta', 'Saskatchewan', 'Ontario', 'Manitoba', 'Ontario', 'New
Brunswick', 'Yukon', 'British Columbia', 'Yukon', 'Newfoundland and
Labrador', 'Manitoba', 'Ontario', 'Yukon', 'British Columbia', 'Yukon',
'Ontario', 'AB', 'Newfoundland and Labrador', 'Nova Scotia', 'Yukon',
'Northwest Territories', 'Nunavut', 'Yukon', 'Nunavut', 'Ontario',
'British Columbia', 'AB', 'Saskatchewan', 'Prince Edward Island',
'Saskatchewan', 'Prince Edward Island', 'Alberta', 'Ontario',
'Alberta', 'Manitoba', 'AB', 'British Columbia', 'Alberta']
```

List of provinces after AB was changed to Alberta:

```
['Alberta', 'Ontario', 'Prince Edward Island', 'Ontario', 'Quebec',
'Saskatchewan', 'Alberta', 'Nova Scotia', 'Alberta', 'Northwest
Territories', 'Saskatchewan', 'Nunavut', 'Nova Scotia', 'Prince Edward
Island', 'Alberta', 'Nova Scotia', 'Prince Edward Island',
'Saskatchewan', 'Nova Scotia', 'Newfoundland and Labrador', 'Ontario',
'Ontario', 'Ontario', 'British Columbia', 'Ontario', 'Nova Scotia',
'Prince Edward Island', 'Saskatchewan', 'Newfoundland and Labrador',
'Ontario', 'Ontario', 'Manitoba', 'British Columbia', 'Ontario',
'Alberta', 'Saskatchewan', 'Ontario', 'Manitoba', 'Ontario', 'New
Brunswick', 'Yukon', 'British Columbia', 'Yukon', 'Newfoundland and
Labrador', 'Manitoba', 'Ontario', 'Yukon', 'British Columbia', 'Yukon',
'Ontario', 'Alberta', 'Newfoundland and Labrador', 'Nova Scotia',
'Yukon', 'Northwest Territories', 'Nunavut', 'Yukon', 'Nunavut',
'Ontario', 'British Columbia', 'Alberta', 'Saskatchewan', 'Prince
Edward Island', 'Saskatchewan', 'Prince Edward Island', 'Alberta',
'Ontario', 'Alberta', 'Manitoba', 'Alberta', 'British Columbia',
'Alberta']
```

List filtered to Alberta only:

```
['Alberta', 'Alberta', 'Alberta', 'Alberta', 'Alberta', 'Alberta',
'Alberta', 'Alberta', 'Alberta', 'Alberta']
```

Alberta occurs 11 times in the modified list.

Example - Sorting a Compound List

Python includes a built-in higher-order function named `sorted` that accepts a list as an argument and returns a new sorted list. Calling the `sorted` function is straightforward for a simple list such as a list of strings or a list of numbers as shown in example 6 and its output.

```
1 # Example 6
2
3 def main():
4     # Create a list that contains country names and print the list.
5     countries = ["Canada", "France", "Ghana", "Brazil", "Japan"]
6     print(countries)
7
8     # Sort the list. Then print the sorted list.
9     sorted_list = sorted(countries)
10    print(sorted_list)
11
12
13 # Call main to start this program.
14 if __name__ == "__main__":
15     main()
```

```
> python countries.py
['Mexico', 'France', 'Ghana', 'Brazil', 'Japan']
['Brazil', 'France', 'Ghana', 'Japan', 'Mexico']
```

A **compound list** is a list that contains lists. Sorting a compound list is more complex than sorting a simple list. Consider this compound list that contains data about some countries.

```
# Create a list that contains data about countries.
countries = [
    # [country_name, land_area, population, gdp_per_capita]
    ["Mexico", 1972550, 126014024, 21362],
    ["France", 640679, 67399000, 45454],
    ["Ghana", 239567, 31072940, 7343],
    ["Brazil", 8515767, 210147125, 14563],
    ["Japan", 377975, 125480000, 41634]
]
```

Perhaps we want the `countries` compound list sorted by country name or perhaps we want it sorted by population. The element that we want a list sorted by is known as the **key element**. If we want to use the `sorted` function to sort a compound list, we must tell the `sorted` function which element is the key element, which we do by passing a small function as an argument into the `sorted` function. This small function is called the **key function** and extracts the key element from a list as shown in example 7.

Notice at line 26 in example 7, there is a lambda function that extracts the population from a country. Then at line 29 that lambda function is passed to the `sorted` function so that the `sorted` function will sort the list of countries by the population.

```
1 # Example 7
2
3 def main():
```

```

4     # Create a list that contains data about countries.
5     countries = [
6         # [country_name, land_area, population, gdp_per_capita]
7         ["Mexico", 1972550, 126014024, 21362],
8         ["France", 640679, 67399000, 45454],
9         ["Ghana", 239567, 31072940, 7343],
10        ["Brazil", 8515767, 210147125, 14563],
11        ["Japan", 377975, 125480000, 41634]
12    ]
13
14     # Print the unsorted list.
15     print("Original unsorted list of countries")
16     for country in countries:
17         print(country)
18     print()
19
20     # Define a lambda function that will be used as the
21     # key function by the sorted function. The lambda
22     # function extracts the population data from a
23     # country so that the population will be used for
24     # sorting the list of countries.
25     POPULATION_INDEX = 2
26     popul_func = lambda country: country[POPULATION_INDEX]
27
28     # Sort the list of countries by the population.
29     sorted_list = sorted(countries, key=popul_func)
30
31     # Print the sorted list.
32     print("List of countries sorted by population")
33     for country in sorted_list:
34         print(country)
35
36
37     # Call main to start this program.
38     if __name__ == "__main__":
39         main()

```

```

> python countries.py
Original unsorted list of countries
['Mexico', 1972550, 126014024, 21362]
['France', 640679, 67399000, 45454]
['Ghana', 239567, 31072940, 7343]
['Brazil', 8515767, 210147125, 14563]
['Japan', 377975, 125480000, 41634]

List of countries sorted by population
['Ghana', 239567, 31072940, 7343]
['France', 640679, 67399000, 45454]
['Japan', 377975, 125480000, 41634]
['Mexico', 1972550, 126014024, 21362]
['Brazil', 8515767, 210147125, 14563]

```

By using a key function it's possible to sort a compound list with a key element that isn't in the list. Consider the compound list named *students* that contains data about various students in example 8. Within the list, each student's given name and surname are stored separately. It is common for a user to want such a list to be sorted by surname and then by given name. A simple way to do that is to write a key function that combines the

surname and given name elements and returns the combined name as the key that the sorted function will use for sorting.

Lines 21–22 in example 8 contain a lambda function that combines a student's surname and given name into a string that is used as the key by the sorted function at line 25. Notice in the output from example 8 that the students are sorted by surname and then by given name.

```
1 # Example 8
2
3 def main():
4     # Create a list that contains data about young students.
5     students = [
6         # [given_name, surname, reading_level]
7         ["Robert", "Smith", 6.7],
8         ["Annie", "Smith", 6.2],
9         ["Robert", "Lopez", 7.1],
10        ["Sean", "Li", 5.6],
11        ["Sofia", "Lopez", 5.3],
12        ["Lily", "Harris", 6.7],
13        ["Alex", "Harris", 5.8]
14    ]
15
16    GIVEN_INDEX = 0
17    SURNAME_INDEX = 1
18
19    # Define a lambda function that combines
20    # a student's surname and given name.
21    combine_names = lambda student_list: \
22        f"{student_list[SURNAME_INDEX]}, {student_list[GIVEN_INDEX]}"
23
24    # Sort the list by the combined key of surname, given_name.
25    sorted_list = sorted(students, key=combine_names)
26
27    # Print the list.
28    for student in sorted_list:
29        print(student)
30
31
32    # Call main to start this program.
33    if __name__ == "__main__":
34        main()
```

```
> python students.py
['Alex', 'Harris', 5.8]
['Lily', 'Harris', 6.7]
['Sean', 'Li', 5.6]
['Robert', 'Lopez', 7.1]
['Sofia', 'Lopez', 5.3]
['Annie', 'Smith', 6.2]
['Robert', 'Smith', 6.7]
```

Videos

If you wish to learn more about functional programming in Python, watch these videos by Dan Bader.

The [Basics of Functional Programming](#) in Python (19 minutes)

The Python [filter function](#) (16 minutes)

The Python [map function](#) (14 minutes)

The Python [reduce function](#) (18 minutes)

Additional Documentation

If you wish to learn even more details about functional programming, the following articles contain reference documentation for functional programming in Python.

Python [map function](#)

Python [filter function](#)

Python [reduce function](#)

Python [functools module](#)

A thorough tutorial about [lambda functions](#)

Summary

In this prepare content, you learned that functional programming is a programming paradigm that focuses on functions, and you learned these three concepts that are used in functional programming:

1. You can pass a function as an argument into another function.
2. A nested function is a function defined inside another function.
3. A lambda function is a small anonymous function.

11 Checkpoint: Functional Programming

Purpose

Reinforce in your mind the concept that a function can be passed as an argument into another function.

Problem Statement

Phone numbers in the U.S. are often stored as ten digits and two dashes in this format: "ddd-ddd-dddd" where each *d* is a digit. The first three digits are called the area code. For example, for many years, 208 was the area code for all phone numbers in the state of Idaho and 801 and 435 were the two area codes for phone numbers in the state of Utah. Some people, when entering their phone number, will not enter the area code and assume that other people will know the correct area code.

Helpful Documentation

The Reading Files article explains how to setup VS Code so that your Python program can [read from a text file](#).

The [prepare content for lesson 9](#) explains how to read the contents of a text file into a list.

The [prepare content for this lesson](#) explains higher-order functions, nested functions, lambda functions, and how to call the [map function](#).

Assignment

Write a program named `add_area_code.py` that reads phone numbers from a text file. Your program must add the area code "208-" to all phone numbers that don't have an area code. Do the following:

1. Download these two files: [phone_numbers.txt](#) and [add_area_code.py](#) and save them in the same folder.
2. Open the `phone_numbers.txt` file in VS Code and examine the phone numbers. Notice that some of the phone numbers do not have an area code and contain only seven digits.

3. Open the `add_area_code.py` file in VS Code. Read the comments in the program and then add code to the program so that it does the following:
 - a. Reads the phone numbers in `phone_numbers.txt` into a list.
 - b. Adds the area code "208-" to the phone numbers that do not have an area code.
 - c. Prints the list with the corrected phone numbers.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Download, examine, and run this Python file named [`test_add_area_code.py`](#) that contains `pytest` test functions. Verify that all the test functions pass. If they don't pass, fix the mistakes in your `add_area_code` function until they pass.
2. Run your program, which should print the list of corrected phone numbers as shown below. Examine the corrected phone numbers printed by your program and verify that all of them have an area code.

```
> python add_area_code.py
['208-318-1623', '208-384-7712', '986-314-1378', '307-377-4921',
 '208-671-7536', '986-228-7414', '208-693-6706', '208-274-2098',
 '208-854-3221', '208-361-9817', '208-422-3492', '208-631-1374',
 '986-533-9667', '208-158-9497', '208-892-9830', '208-477-5323',
 '406-204-8756', '208-805-2538', '208-913-9942', '208-216-9589',
 '406-138-9226', '801-644-6761', '208-814-4444', '208-822-6738',
 '208-892-1093', '208-936-6389', '435-656-5821', '435-605-8294',
 '208-271-1868', '208-894-6391', '208-232-4118', '208-132-2121',
 '208-515-8227', '208-848-2334', '208-426-1557', '208-952-4330',
 '208-998-2171', '208-743-9219', '208-944-4381', '208-589-7767',
 '208-407-3555', '208-928-9293', '208-957-5808', '208-940-6629',
 '208-825-9259', '208-123-7580', '208-374-8128', '208-847-4807',
 '986-159-8293', '208-515-9963', '406-241-6676', '208-907-1851',
 '208-297-6367', '208-936-8173', '208-376-7010', '208-681-1558',
 '208-800-4436', '208-590-8866', '208-788-1866', '208-163-7631',
 '208-928-8793', '406-595-2898', '208-330-3155', '208-884-7294',
 '208-120-4949', '801-343-8776', '208-362-3779', '208-482-2107',
 '208-366-5618', '435-957-8250', '208-441-5628', '307-494-6001',
 '208-571-5297', '208-366-7389', '801-730-8010', '208-134-6240',
 '208-402-3658', '208-151-9828', '208-485-7973', '208-506-2683',
 '208-518-9635', '208-769-7551', '208-776-1574', '208-604-4304',
 '208-119-9734', '208-917-6174', '208-773-1167', '307-749-5991',
 '208-334-8971', '208-995-8602', '208-193-5073', '208-345-4015',
 '208-367-8633', '208-289-6515', '208-265-8998', '208-800-3565',
 '208-211-2706', '208-408-3950', '208-928-3119', '208-691-9409']
```

Ponder

In the `main` function, look at the line of code that calls the `map` function. Think about the fact that the `map` function will call the `add_area_code` function once for each element in the `phone_numbers` list, and the `add_area_code` function will add the area code "208-" at the beginning of each phone number if necessary.

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson11/phone_numbers.txt

1	208-318-1623
2	384-7712
3	986-314-1378
4	307-377-4921
5	208-671-7536
6	986-228-7414
7	693-6706
8	208-274-2098
9	208-854-3221
10	208-361-9817
11	422-3492
12	631-1374
13	986-533-9667
14	158-9497
15	208-892-9830
16	208-477-5323
17	406-204-8756
18	805-2538
19	208-913-9942
20	208-216-9589
21	406-138-9226
22	801-644-6761
23	814-4444
24	822-6738
25	208-892-1093
26	208-936-6389
27	435-656-5821
28	435-605-8294
29	208-271-1868
30	208-894-6391
31	232-4118
32	208-132-2121
33	515-8227
34	208-848-2334
35	208-426-1557
36	208-952-4330
37	998-2171
38	743-9219
39	208-944-4381
40	208-589-7767
41	208-407-3555
42	928-9293
43	208-957-5808
44	940-6629
45	208-825-9259
46	208-123-7580
47	208-374-8128
48	847-4807
49	986-159-8293
50	208-515-9963
51	406-241-6676
52	907-1851
53	208-297-6367
54	208-936-8173
55	208-376-7010
56	208-681-1558
57	208-800-4436

58	208-590-8866
59	788-1866
60	208-163-7631
61	928-8793
62	406-595-2898
63	208-330-3155
64	884-7294
65	208-120-4949
66	801-343-8776
67	208-362-3779
68	482-2107
69	366-5618
70	435-957-8250
71	441-5628
72	307-494-6001
73	208-571-5297
74	208-366-7389
75	801-730-8010
76	134-6240
77	208-402-3658
78	208-151-9828
79	208-485-7973
80	506-2683
81	208-518-9635
82	208-769-7551
83	776-1574
84	208-604-4304
85	208-119-9734
86	917-6174
87	208-773-1167
88	307-749-5991
89	208-334-8971
90	995-8602
91	208-193-5073
92	208-345-4015
93	208-367-8633
94	289-6515
95	208-265-8998
96	208-800-3565
97	211-2706
98	208-408-3950
99	208-928-3119
100	208-691-9409

lesson11/add_area_code.py

```
1 def main():
2     try:
3         # Open the file phone_numbers.txt for reading and read all
4         # of the phone numbers into a list named phone_numbers.
5         phone_numbers = read_list("phone_numbers.txt")
6
7         # Print the list of phone numbers which will show that
8         # some of the phone number don't have an area code.
9         print(phone_numbers)
10
11        # Some of the phone numbers in phone_numbers.txt do not start
12        # with an area code. Replace each short phone number with a
13        # phone number that begins with the area code "208-". To do this
14        # call the map function and pass the add_area_code function and
15        # the list of phone numbers as arguments to the map function.
16        pass
17
18        # Print the list with the corrected phone numbers.
19        print(new_numbers)
20
21    except (FileNotFoundException, PermissionError) as error:
22        print(type(error).__name__, error, sep=": ")
23
24
25    def add_area_code(phone_number):
26        """Phone numbers in the U.S. are often stored as ten digits and
27        two dashes in this format: "ddd-ddd-dddd" where each d is a digit.
28        If the length of phone_number is less than 12 characters, add the
29        area code "208-" at the beginning of phone_number and return
30        phone_number.
31
32        Parameter phone_number: a string of digits formatted as
33        "ddd-dddd" or "ddd-ddd-dddd"
34        Return: a string of digits formated as "ddd-ddd-dddd"
35        """
36        pass
37
38
39    def read_list(filename):
40        """Read the contents of a text file into a list and
41        return the list. Each element in the list will contain
42        one line of text from the text file.
43
44        Parameter filename: the name of the text file to read
45        Return: a list of strings
46        """
47        # Create an empty list named text_list.
48        text_list = []
49
50        # Open the text file for reading and store a reference
51        # to the opened file in a variable named text_file.
52        with open(filename, "rt") as text_file:
53
54            # Read the contents of the text
55            # file one line at a time.
56            for line in text_file:
57
```

```
58     # Remove white space, if there is any,
59     # from the beginning and end of the line.
60     clean_line = line.strip()
61
62     # Append the clean line of text
63     # onto the end of the list.
64     text_list.append(clean_line)
65
66 # Return the list that contains the lines of text.
67 return text_list
68
69
70 # If this file is executed like this:
71 # > python add_area_code.py
72 # then call the main function. However, if this file is simply
73 # imported (e.g. into a test file), then skip the call to main.
74 if __name__ == "__main__":
75     main()
76
```

lesson11/test_add_area_code.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 from add_area_code import add_area_code
4 import pytest
5
6 def test_add_area_code():
7     """Verify that the add_area_code function works correctly.
8     Parameters: none
9     Return: nothing
10    """
11    old_phone = "656-4771"
12    new_phone = add_area_code(old_phone)
13
14    # Verify that the add_area_code function returned a string.
15    assert isinstance(new_phone, str), \
16        f"add_area_code function failed to return a string for {old_phon
17
18    # Verify that the add_area_code function
19    # returned the correct value.
20    assert new_phone == "208-656-4771"
21
22    old_phone = "801-412-3210"
23    new_phone = add_area_code(old_phone)
24
25    # Verify that the add_area_code function returned a string.
26    assert isinstance(new_phone, str), \
27        f"add_area_code function failed to return a string for {old_phon
28
29    # Verify that the add_area_code function
30    # returned the correct value.
31    assert new_phone == old_phone
32
33
34    # Call the main function that is part of pytest so that the
35    # computer will execute the test functions in this file.
36    pytest.main(["-v", "--tb=line", "-rN", __file__])
```

lesson11/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def main():
4     try:
5         # Open the file phone_numbers.txt for reading and read all
6         # of the phone numbers into a list named phone_numbers.
7         phone_numbers = read_list("phone_numbers.txt")
8
9         # Print the list of phone numbers which will show that
10        # some of the phone number don't have an area code.
11        print(phone_numbers)
12
13        # Some of the phone numbers in phone_numbers.txt do not start
14        # with an area code. Replace each short phone number with a
15        # phone number that begins with the area code "208-". To do this
16        # call the map function and pass the add_area_code function and
17        # the list of phone numbers as arguments to the map function.
18        new_numbers = list(map(add_area_code, phone_numbers))
19
20        # Print the list with the corrected phone numbers.
21        print(new_numbers)
22
23    except (FileNotFoundException, PermissionError) as error:
24        print(type(error).__name__, error, sep=": ")
25
26
27 def add_area_code(phone_number):
28     """Phone numbers in the U.S. are often stored as ten digits and
29     two dashes in this format: "ddd-ddd-dddd" where each d is a digit.
30     If the length of phone_number is less than 12 characters, add the
31     area code "208-" at the beginning of the phone_number and return
32     phone_number.
33
34     Parameter phone_number: a string of digits formatted as
35         "ddd-dddd" or "ddd-ddd-dddd"
36     Return: a string of digits formatted as "ddd-ddd-dddd"
37     """
38     if len(phone_number) < 12:
39         phone_number = "208-" + phone_number
40     return phone_number
41
42
43 def read_list(filename):
44     """Read the contents of a text file into a list and
45     return the list. Each element in the list will contain
46     one line of text from the text file.
47
48     Parameter filename: the name of the text file to read
49     Return: a list of strings
50     """
51     # Create an empty list named text_list.
52     text_list = []
53
54     # Open the text file for reading and store a reference
55     # to the opened file in a variable named text_file.
56     with open(filename, "rt") as text_file:
```

```
58     # Read the contents of the text
59     # file one line at a time.
60     for line in text_file:
61
62         # Remove white space, if there is any,
63         # from the beginning and end of the line.
64         clean_line = line.strip()
65
66         # Append the clean line of text
67         # onto the end of the list.
68         text_list.append(clean_line)
69
70     # Return the list that contains the lines of text.
71     return text_list
72
73
74 # If this file is executed like this:
75 # > python check_solution.py
76 # then call the main function. However, if this file is simply
77 # imported (e.g. into a test file), then skip the call to main.
78 if __name__ == "__main__":
79     main()
```

11 Team Activity: Functional Programming

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Purpose

Reinforce in your mind the concept that a function can be passed as an argument into another function.

Assignment

The CSV file named `pupils.csv` contains data about 100 students. Unfortunately, the data is not sorted. As a team, write a program that reads the contents of `pupils.csv` into a compound list, sorts the list by birthday from oldest to youngest, and then prints the list with the data about each student on a separate line.

Helpful Documentation

The Reading Files article explains how to setup VS Code so that your Python program can [read from a text file](#).

The [prepare content for lesson 7](#) explains how to use compound lists.

The [prepare content for lesson 9](#) explains how to read the contents of a text file into a list.

The [prepare content for this lesson](#) explains higher-order functions, nested functions, lambda functions and how to call the Python built-in [sorted function](#).

Steps

Do the following:

1. Download these two files: [pupils.csv](#) and [pupils.py](#) and save them in the same folder.
2. Open the `pupils.csv` file in VS Code and notice that it has three columns: `givenName`, `surname`, and `birthdate`.
3. Open the `pupils.py` file in VS Code. At the bottom of the file write a function named `print_list` that takes a list as a parameter and prints each element of the list on a separate line. In other words, this `print_list` function should include a `for` loop that prints each element on a separate line.
4. Near the top of `pupils.py`, below the three indexes, write the `main` function. Inside the `main` function, write statements to do the following:
 - a. Call the `read_compound_list` function to read the `pupils.csv` file into a list named `students_list`.
 - b. Write a lambda function that will extract the birthdate from a student.
 - c. Write a call to the Python built-in `sorted` function that will sort the `students_list` by birthdate from oldest to youngest.
 - d. Print the `students_list` by calling the `print_list` function.
5. At the bottom of the `pupils.py` file write a call to the `main` function.

Core Requirements

Your program must contain the following:

1. A function named `print_list` that takes a list as a parameter and prints the list with each element of the list on a separate line.
2. A function named `main` that calls `read_compound_list` and `print_list`.
3. Statements in the `main` function that sort the `students_list` by birthdate from oldest to youngest.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Within the `main` function, replace the code that sorts the `students_list` by birthdate, with code that sorts the `students_list` by given name.
2. Within the `main` function, replace the code that sorts the `students_list` by birthdate, with code that sorts the `students_list` by birth month and day. In other words, the

code should sort the `students_list` by birthdate but ignore the year when a student was born.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and ensure that your program's output is sorted correctly as shown below.

```
> python pupils.py
Ordered from Oldest to Youngest
[('Cody', 'Gjoni', '2008-01-14')]
[('Jakob', 'Moore', '2008-08-09')]
[('Dylan', 'Bradford', '2008-10-11')]
[('Chih-Yang', 'Olson', '2008-12-23')]
[('Camdon', 'Radke', '2009-01-31')]
[('Jacob', 'Ortiz', '2009-02-04')]
[('Tanner', 'McAllister', '2009-02-13')]
[('Aoi', 'Lee', '2009-02-22')]
[('Colton', 'Kent', '2009-02-25')]
[('Marco', 'Zeng', '2009-06-25')]
:

Ordered by Given Name
[('Adam', 'Chase', '2009-08-04')]
[('Aidan', 'Havens', '2014-08-12')]
[('Alexander', 'Bingham', '2015-01-30')]
[('Amarsanaa', 'Cromar', '2010-07-15')]
[('Ammon', 'Reeder', '2014-11-22')]
[('Andrea', 'Omokoh', '2014-11-08')]
[('Aoi', 'Lee', '2009-02-22')]
[('Aranza', 'Billman', '2012-12-08')]
[('Benjamin', 'Rojas', '2010-12-24')]
[('Brody', 'Wilson', '2013-10-29')]
:

Ordered by Birth Month and Day
[('Mitchel', 'Elliott', '2010-01-03')]
[('Nathan', 'Bowman', '2014-01-07')]
[('Christian', 'White', '2015-01-09')]
[('Manoel', 'Gonzalez', '2014-01-10')]
[('Cody', 'Gjoni', '2008-01-14')]
[('Curtis', 'Loveridge', '2011-01-14')]
[('Alexander', 'Bingham', '2015-01-30')]
[('Camdon', 'Radke', '2009-01-31')]
[('Jacob', 'Ortiz', '2009-02-04')]
[('Tanner', 'McAllister', '2009-02-13')]
:
```

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#) or the [stretch solution](#). Please ***do not look at the sample solution*** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson11/pupils.csv

1	givenName,surname,birthdate
2	Gavin,Tyra,2012-11-25
3	Zachary,Danso,2009-08-17
4	Ethan,Balls,2014-07-06
5	Justin,Otis,2012-12-25
6	Mitchel,Elliott,2010-01-03
7	Daniel,Ririe,2010-07-24
8	Dawson,Tomlinson,2013-06-12
9	Carolina,Rowley,2014-03-22
10	Ryan,Knudson,2010-08-07
11	Manoel,Gonzalez,2014-01-10
12	Thomas,Cantarero,2011-08-17
13	Alexander,Bingham,2015-01-30
14	Cody,Gjoni,2008-01-14
15	Tyler,Moreira,2009-10-16
16	Camron,Jensen,2015-03-01
17	Lora,de Jong,2014-08-03
18	Giuliana,Child,2014-12-09
19	Benjamin,Rojas,2010-12-24
20	Jonathan,Hibbert,2012-04-09
21	Lora,Tuckett,2014-09-04
22	Joshua,Bingham,2015-05-22
23	Krystiane,Cornelius,2013-08-05
24	Dylan,Bradford,2008-10-11
25	Raymond,Cooper,2014-09-28
26	Chase,McGettigan,2011-10-07
27	Bryson,Cheuk,2015-03-21
28	Aoi,Lee,2009-02-22
29	Joshua,Bringhurst,2011-07-18
30	Amarsanaa,Cromar,2010-07-15
31	Camdon,Radke,2009-01-31
32	Curtis,Loveridge,2011-01-14
33	Kaden,Potter,2013-02-21
34	Chase,Grecco,2015-06-18
35	Robert,Vasiuk,2010-10-20
36	Ammon,Reeder,2014-11-22
37	Daniel,Carroll,2012-03-17
38	Tyler,Gubler,2012-10-01
39	Jonah,Skinner,2010-09-18
40	Matthew,Knutzen,2013-09-16
41	Crescencio,Burke,2013-10-26
42	Carl,Smith,2015-06-14
43	Garrett,McMullens,2011-04-22
44	Jacob,Ortiz,2009-02-04
45	Jonathan,Weatherston,2013-12-30
46	Aidan,Havens,2014-08-12
47	Adam,Chase,2009-08-04
48	Joshua,Vredenburg,2010-05-22
49	Jakob,Moore,2008-08-09
50	Chih-Yang,Olson,2008-12-23
51	Tanner,McAllister,2009-02-13
52	Trevor,Johnson,2009-11-19
53	Cierra,Mikami,2013-10-30
54	Star,Resendiz,2011-12-19
55	John,Tsuchiya,2010-06-30
56	Nathan,Johnson,2013-03-03
57	Caleb,Aitchison,2011-10-03

58	Spencer,Duffin,2010-02-26
59	Shiekahadd,Autran,2012-03-22
60	Emmanuel,Johnson,2010-09-06
61	Marco,Zeng,2009-06-25
62	Colton,Kent,2009-02-25
63	Brody,Wilson,2013-10-29
64	Jefferson,Brush,2010-07-30
65	Cristians,Cannon,2010-10-28
66	Christopher,Beck,2012-08-12
67	Ryan,Bishop,2014-11-10
68	Christopher,Johnson,2011-04-27
69	Daniel,Rowley,2009-10-16
70	Jonathan,Martell,2011-08-25
71	Jared,Jacobs,2013-10-18
72	Spencer,Biziyabal,2009-07-09
73	Juan,Miller,2012-06-07
74	Jordan,White,2012-06-16
75	Kaleb,Young,2012-06-07
76	Nathan,Bowman,2014-01-07
77	Steven,Bond,2013-12-28
78	Kjellden,Merrell,2009-11-03
79	Christian,White,2015-01-09
80	Chaim,Coglianese,2009-11-14
81	Dallin,Brown,2013-12-17
82	Raquel,White,2010-05-14
83	Jason,Smith,2009-11-21
84	Ernest,Adams,2013-05-15
85	Aranza,Billman,2012-12-08
86	Noah,Flora,2011-03-05
87	Madison,Russell,2012-08-30
88	Julia,Sorenson,2010-12-26
89	Andrea,Omokoh,2014-11-08
90	Virginia,Banta,2010-10-11
91	Melissa,Littlefield,2009-07-21
92	Louis,Littlefield,2013-10-21
93	Diana,Brooks,2012-10-26
94	Joyce,Cox,2014-07-30
95	Wayne,Orchard,2010-02-16
96	Gregory,Kofford,2011-07-22
97	Richard,Ali,2013-05-29
98	Jack,Reynolds,2013-12-03
99	Deborah,Davison,2012-05-10
100	Larry,Omokoh,2011-03-16
101	Nancy,Moore,2014-08-02

lesson11/pupils.py

```
1 import csv
2
3
4 # Each row in the pupils.csv file contains three elements.
5 # These are the indexes of the elements in each row.
6 GIVEN_NAME_INDEX = 0
7 SURNAME_INDEX = 1
8 BIRTHDATE_INDEX = 2
9
10
11 def read_compound_list(filename):
12     """Read the text from a CSV file into a compound list.
13     The compound list will contain small lists. Each small
14     list will contain the data from one row of the CSV file.
15
16     Parameter
17         filename: the name of the CSV file to read.
18     Return: the compound list
19     """
20
21     # Create an empty list.
22     compound_list = []
23
24     # Open the CSV file for reading.
25     with open(filename, "rt") as csv_file:
26
27         # Use the csv module to create a reader
28         # object that will read from the opened file.
29         reader = csv.reader(csv_file)
30
31         # The first line of the CSV file contains column headings
32         # and not a student's I-Number and name, so this statement
33         # skips the first line of the CSV file.
34         next(reader)
35
36         # Process each row in the CSV file.
37         for row in reader:
38
39             # Append the current row at the end of the compound list.
40             compound_list.append(row)
41
42     return compound_list
```

lesson11/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 import csv
4
5
6 # Each row in the pupils.csv file contains three elements.
7 # These are the indexes of the elements in each row.
8 GIVEN_NAME_INDEX = 0
9 SURNAME_INDEX = 1
10 BIRTHDATE_INDEX = 2
11
12
13 def main():
14     try:
15         # Read the pupils.csv file into a compound list.
16         students_list = read_compound_list("pupils.csv")
17
18         # As a debugging aid, print the students_list.
19         #print(students_list)
20
21         # Define a lambda function that extracts a student's birthdate.
22         extract_birthdate = lambda student: student[BIRTHDATE_INDEX]
23
24         # Call the Python built-in sorted function to sort
25         # the list of students by their birthdate and pass
26         # the lambda function to the sorted function.
27         sorted_list = sorted(students_list, key=extract_birthdate)
28
29         # Call the print_list function to print the sorted list.
30         print_list(sorted_list)
31
32     except (FileNotFoundException, PermissionError) as error:
33         print(type(error).__name__, error, sep=": ")
34
35
36 def read_compound_list(filename):
37     """Read the text from a CSV file into a compound list.
38     The compound list will contain small lists. Each small
39     list will contain the data from one row of the CSV file.
40
41     Parameter
42         filename: the name of the CSV file to read.
43     Return: the compound list
44     """
45
46     # Create an empty list.
47     compound_list = []
48
49     # Open the CSV file for reading.
50     with open(filename, "rt") as csv_file:
51
52         # Use the csv module to create a reader
53         # object that will read from the opened file.
54         reader = csv.reader(csv_file)
55
56         # The first line of the CSV file contains column headings
57         # and not a student's I-Number and name, so this statement
58         # skips the first line of the CSV file.
```

```
58     next(reader)
59
60     # Process each row in the CSV file.
61     for row in reader:
62
63         # Append the current row at the end of the compound list.
64         compound_list.append(row)
65
66     return compound_list
67
68
69 def print_list(compound_list):
70     """Print the elements in a compound list to
71     the terminal window for the user to see.
72
73     Parameter
74         compound_list: a list that contains other lists
75     Return: nothing
76     """
77     for element in compound_list:
78         print(element)
79     print()
80
81
82 # If this file is executed like this:
83 # > python teach_solution.py
84 # then call the main function. However, if this file is simply
85 # imported (e.g. into a test file), then skip the call to main.
86 if __name__ == "__main__":
87     main()
```

11 Prove Milestone: Student Chosen Program

Purpose

Prove that you can write a significant Python program that solves a real-world problem and is well organized with functions.

Assignment

During this lesson and the next two lessons, write a Python program that you choose. Your program must include multiple functions that each perform a single task. You must write test functions for at least two of your program functions and run your test functions with `pytest`.

Ideally your program will solve a real-world problem or become a tool in a subject area that interests you, such as chemistry, physics, geography, fashion, family history, linguistics, social work, food science, carpentry, machinery, engineering, mathematics, computing, etc. Before you begin writing your program, you must ***propose your program to your instructor*** to ensure that the program is neither too easy nor too difficult for the time remaining in this course. If you finish your program more quickly than you or your instructor expected, and there is time remaining in this course, then choose and write another program.

Proposal

To write a proposal for your program, download and save this [`proposal.txt`](#) file. Open the downloaded file in VS Code. Answer each of the questions and save your file.

Possible Programs

If you find it difficult to choose a program to write, you may choose to write one of the following programs:

1. A program that performs calculations that are useful in your major or hobby.
2. A program with a complete graphical user interface (GUI), including windows, frames, labels, text fields, and buttons. Use either the `tkinter` or `kivy` module. The team assignment in the next lesson will show you how to write code for a simple GUI.

3. A program that reads a text file and counts how often each word occurs in the file.
4. A program that reads English text from a file and converts strangely spelled words to a simpler spelling that more closely matches their pronunciation. Consider just a few of the strangely spelled English words: comb, lamb, have, done, two, Wednesday, should, night ("ight" is a blight on the English language). For more information, see the [English-language spelling reform](#) article at Wikipedia.
5. A program to reorganize or rename all the files in a collection of files, such as collection of Microsoft Word documents, Excel spreadsheets, music (mp3) files, pictures (jpg) files. If you choose to write this program, we **strongly** suggest that you make a copy of the files that your program will reorganize or rename and run your program on the copy. This is because it is highly likely that you will make mistakes when writing this program, and such mistakes may cause the files to be deleted. Really.
6. A program that graphically simulates the spread of a disease like the simulations in this [Washington Post article](#). Your program should include variables that can be changed, such as movement rate, probability of transmission, length of contagiousness, length of illness, and probability of death.
7. A program that retrieves data from a server on the internet. The Python [requests](#) module contains functions that retrieve data from a server.
8. A program that uses the [pandas](#) module to process a data set and produces charts that help people visualize information about the data set.
9. A program that retrieves data from a relational database. The MySQL Python connector is a Python module that enables Python programs to read from and write to a MySQL relational database. It was written by the same developers who wrote the MySQL server. You can download it from the [MySQL downloads page](#) and install it on your Windows, Mac OS, or Linux computer.
10. A program that retrieves data from a Firebase database.

These three websites list other ideas for Python projects:

- » [42 Exciting Python Projects for Beginners](#)
- » [45 Fun Python Project Ideas for Easy Learning](#)
- » [Python Projects for Beginners](#)

If you don't understand the description of one of these possible projects, ask your CSE 111 team members or your instructor to explain the project to you.

Organization

As you develop your program, be certain to divide the program into functions. Each function should perform a single task and be appropriately named. Remember that the most reusable and easily testable functions don't get user input and don't print results but instead have parameters and return a result. Functions that get user input and print results are important and do useful work but are not easily reusable and testable. It's fine if some of the functions in your program have no parameters, return nothing, get user input, and print results, but it's bad if all of your program functions are like that.

Testing Procedure

You should follow good coding practices by writing test functions for many of your program functions. Run your test functions with pytest to verify that your program functions work correctly. Many students believe that the program functions they write cannot be tested with test functions and pytest. Such claims are simply false. Here are just some of the reasons that students have claimed make their program functions untestable and example code that shows such claims are false.

My function can't be tested because it:

- Calculates numbers

```
def payment(amt, ar, y, ppy):
    """Computes and returns the payment amount
    for a loan with a fixed annual interest rate.
    """
    r = ar / ppy
    n = y * ppy
    p = amt * r / (1 - (1 + r) ** -n)
    return round(p, 2)
```

```
def test_payment():
    """Verify that the payment function works correctly."""
    assert payment(10_000, 0.07, 4, 12) == 239.46
    assert payment(80_000, 0.04, 15, 4) == 1779.56
```

- Processes text instead of numbers

```
def prefix(s1, s2):
    """Return the prefix, if any, that appears in both s1
    and s2. In other words, return a string of the
    characters that appear at the beginning of both s1 and
    s2. For example, if s1 is "inconceivable" and s2 is
    "inconvenient", this function will return "incon".
    """
    s1 = s1.lower()
    s2 = s2.lower()
    i = 0
    limit = min(len(s1), len(s2))
    while i < limit:
        if s1[i] != s2[i]:
            break
```

```
i += 1  
return s1[0:i]
```

```
def test_prefix():  
    """Verify that the prefix function works correctly."""  
    assert prefix("", "") == ""  
    assert prefix("", "correct") == ""  
    assert prefix("clear", "") == ""  
    assert prefix("happy", "funny") == ""  
    assert prefix("cat", "catalog") == "cat"  
    assert prefix("dogmatic", "dog") == "dog"  
    assert prefix("jump", "joyous") == "j"  
    assert prefix("unwise", "ungrateful") == "un"  
    assert prefix("Disable", "dIstasteful") == "dis"
```

- Gets user input from a terminal

```
def get_int(prompt, min, max):  
    """Prompt the user for an integer, verify that it is  
    between min and max inclusive, and return the integer.  
    """  
    done = False  
    while not done:  
        try:  
            text = input(prompt)  
            number = int(text)  
            if number < min:  
                print(f"Invalid input: number must be {min} or greater.")  
            elif number > max:  
                print(f"Invalid input: number must be {max} or less.")  
            else:  
                done = True  
        except ValueError as val_err:  
            print(f"Invalid integer: {text}. Please try again.")  
    return number
```

```
from io import StringIO  
import re  
import sys  
  
def test_get_int(monkeypatch):  
    """Verify that the get_int function works correctly."""  
    stdin = StringIO("1r\n3\n12\n8")  
    stdout = StringIO()  
    monkeypatch.setattr(sys, "stdin", stdin)  
    monkeypatch.setattr(sys, "stdout", stdout)  
  
    min = 1  
    max = 10  
    prompt = f"Enter an integer between {min} and {max}: "  
    num = get_int(prompt, min, max)  
  
    monkeypatch.undo()  
    pattern = prompt + "\\\s*" \  
        + "Invalid integer: 1r\\. Please try again\\.\\s*" \  
        + prompt + "\\\s*" \  
        + f"Invalid input: number must be {min} or greater\\.\\s*" \  
        + prompt + "\\\s*" \  
        + "
```

```

        + f"Invalid input: number must be {max} or less\\.\s*"
        + prompt + "\s*"
    output = stdout.getvalue()
    assert re.compile(pattern).match(output) != None
    assert num == 8

```

- Gets user input from a GUI

```

"""
In order to test a program that gets user input from a
graphical user interface, separate the event functions (the
ones that are executed when a user types a value in a text
field) into two functions. Move the calculations out of the
event function and into a calculation function that takes
parameters, performs a calculation, and returns a result.
Then write a test function for the new calculation function.
"""

# This function is called each time the user releases a key.
def calc(event):
    try:
        # Get the user input.
        w = txtWidth.get()
        a = txtRatio.get()
        d = txtDiam.get()

        # Compute the tire volume.
        v = tire_volume(w, a, d)

        # Display the volume for the user to see.
        lblResult.config(text=f"{v:.1}")

    except ValueError:
        # When the user deletes all the digits in one
        # of the text fields, clear the result labels.
        lblResult.config(text="")

def tire_volume(width, ratio, diam):
    """Compute and return the approximate volume of a tire."""
    vol = (math.pi * width * width * ratio *
           (width * ratio + 2540 * diam)) / 10_000_000
    return vol

```

```

def test_tire_volume():
    """Verify that the tire_volume function works correctly."""
    assert tire_volume(185, 60, 14) == approx(30101.58, 0.01)
    assert tire_volume(205, 60, 15) == approx(39924.49, 0.01)

```

- Writes to a file

```

def write_file(filename, lines):
    """Create a file and write lines of text to it.
    Parameter
        lines: a list of strings.
    Return: nothing
    """
    with open(filename, "wt") as outfile:
        for text in lines:
            print(text, file=outfile)

```

```

import os

def test_write_file():
    """Verify that the write_file function works correctly."""
    lines = [
        "Beware of false prophets, who come to you in sheep's",
        "clothing, but inwardly they are ravening wolves. Ye",
        "shall know them by their fruits. Do men gather grapes",
        "of thorns, or figs of thistles? Even so every good tree",
        "bringeth forth good fruit; but a corrupt tree bringeth",
        "forth evil fruit. A good tree cannot bring forth evil",
        "fruit, neither can a corrupt tree bring forth good fruit.",
        "Every tree that bringeth not forth good fruit is hewn",
        "down, and cast into the fire. Wherefore, by their fruits",
        "ye shall know them."
    ]
    filename = "fruits.txt"

    # Call the write_file function to
    # write a file named fruits.txt.
    write_file(filename, lines)

    # Read the contents of the fruits.txt file.
    with open(filename, "rt") as infile:

        # Read all the characters in the file into a string.
        string = infile.read()

        # Split the string into a list of strings named
        # written. Each line of text from the text file
        # will be stored in its own element in the list.
        written = string.splitlines()

        # Delete the fruits.txt file.
        os.remove(filename)

    # Verify that write_file correctly wrote the fruits.txt file.
    assert lines == written

```

- Reads from a file

```

def read_file(filename):
    """Reads and returns the contents
    of a text file as a list of strings.
    """
    lines = None
    with open(filename, "rt") as infile:

        # Read all the characters in the file into a string.
        string = infile.read()

        # Split the string into a list of strings named
        # lines. Each line of text from the text file
        # will be stored in its own element in the list.
        lines = string.splitlines()

    return lines

```

```

import os

```

```

def test_read_file():
    """Verify that the read_file function works correctly."""
    # Write a sample file with three lines
    filename = "lines.txt"
    lines = ["first", "second", "third"]
    with open(filename, "wt") as outfile:
        print(*lines, sep="\n", file=outfile)

    # Call read_file to read the sample file.
    read = read_file(filename)

    # Delete the lines.txt file.
    os.remove(filename)

    # Verify that read_file read the file correctly.
    assert read == lines

```

- Uses the pandas module

```

def filter_for_meter(df, meter_number):
    """Return a new DataFrame that contains only the rows
    where the meterNumber column equals the parameter
    meter_number.

    Parameters
        df: The DataFrame that this function will filter.
        meter_number: df will be filtered so that all the rows
                      in the new DataFrame will have this meter number.
    Return: A new DataFrame.
    """
    filter = (df["meterNumber"] == meter_number)
    filtered_df = df[filter]
    return filtered_df

```

```

import pandas as pd

def test_filter_for_meter():
    """Verify that the filter_for_meter
    function works correctly.
    """

    # Read the water.csv file and convert the
    # readDate column from a string to a datetime64.
    df = pd.read_csv(FILENAME, parse_dates=["readDate"])

    # Call the filter_for_meter function.
    meter_num = "M4103"
    filtered_df = filter_for_meter(df, meter_num)

    # Verify that the filtered data
    # frame contains at least one row.
    assert len(filtered_df.index) > 0

    # Verify that all the meter numbers in the
    # filtered data frame are the correct number.
    for value in filtered_df["meterNumber"]:
        assert value == meter_num

```

- My code is so good that it doesn't need to be tested.

Good luck with that attitude, hot shot. In all seriousness, how do you know your code is good? Besides, no matter how good of a programmer you are, you still make mistakes. It is better to find those mistakes yourself by writing and running test functions than it is to make users of your software find your mistakes.

Submission

At the end of this lesson and lessons 12 and 13, you must submit a description of your work, and your teacher or teaching assistant will grade your work according to the following rubrics.

Lesson 11 Rubric

1. Proposal—40%: Does your proposal adequately answer the questions in the template?
2. Time—50%: Did you spend at least three hours on your proposal and program during the current lesson?
3. Description—10%: Is the description of your work for this lesson complete and easily understandable? At the end of each lesson, you will submit a description of your work. This description should follow normal English rules of spelling and grammar. It must include the following:
 - a. The amount of time that you spent working on your program during the current lesson.
 - b. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
 - c. A description or list of the work that you finished on your program.

Lesson 12 Rubric

1. Time—50%: Did you spend at least three hours on your Python program or test functions during the current lesson?
2. Organization—20%:
 - a. Is your program organized into multiple functions?
 - b. Does each function in your program perform just one task?
3. Progress—20%: Did you complete some significant part of your program during the current week?
4. Description—10%: Is the description of your work for this lesson complete and easily understandable? Your description should include the following:
 - a. A list of the function names in your program.

- b. A list of the test function names in your test code.
- c. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
- d. A description or list of the work that you finished on your program.

Lesson 13 Rubric

1. Time—50%: Did you spend at least six hours on your Python program or test functions during the current lesson?
2. Description—10%: Is the description of your work for this lesson complete and easily understandable? Your description should include the following:
 - a. A list of the function names in your program.
 - b. A list of the test function names in your test code.
 - c. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
 - d. A description or list of the work that you finished on your program.
3. Python program file—25%: Upload your Python program. Your teacher will evaluate it according to these criteria:
 - a. Your program is divided into functions and each function performs one task only.
 - b. Your program effectively uses existing Python modules such as math, random, requests, pandas, and tkinter.
 - c. Your program performs a significant real world task.
4. Python test file—15%: Upload your Python test file. Your teacher will evaluate it according to these criteria:
 - a. Each testable program function is covered (tested) by one test function.
(Some functions, especially main and those that create GUIs are difficult to test and don't need to have a corresponding test function.)
 - b. Each test function completely exercises (tests) its corresponding program function. In other words, the test function calls the program function multiple times with different arguments, including unusual or unexpected values.

lesson11/proposal.txt

```
1 CSE 111 Proposal for a Student Chosen Program
2
3     (This is a proposal. All proposals are written with our best
4      knowledge at the beginning of a project. As we progress from the
5      beginning to the end of a project, we gain more knowledge, and we
6      change some of our proposed ideas. This is part of completing any
7      project and is fine. However, we still must write a proposal so
8      that our teachers or supervisors know what we intend to do. Please
9      answer each of the following questions to the best of your
10     knowledge.)
11
12 1. What is the title of your program?
13
14 2. What real-world problem will your program address or help to solve?
15
16 3. What will you learn from developing this program?
17
18 4. What Python modules will your program use?
19
20     (Some examples are: csv, datetime, functools, matplotlib, math,
21      pandas, pytest, random, requests, and tkinter.)
22
23 5. Will you separate your Python program into functions that each
24     perform a single task?
25
26     (Remember that the most reusable functions don't get user input and
27      don't print results but instead have parameters and return a result.
28      Functions that get user input and print results are important and do
29      useful work but are not easily reusable.)
30
31     (Remember also that it's hard to test functions that get user input
32      and print results. It's easy to test functions that don't get user
33      input and don't print results but instead have parameters and return
34      a result. Therefore, you should write most of your program function
35      to have parameters and return a result.)
36
37 6. Will you write test functions to test at least two of your program
38     functions?
39
```

12 Prepare: Using Objects

A **paradigm** is a way of thinking or a way of perceiving the world. There are at least four main paradigms for programming a computer: procedural, declarative, functional, and object-oriented. During most of CSE 110 and 111, you used procedural programming. During the previous lesson, you encountered functional programming. During this lesson, you will be introduced to object-oriented programming.

Programming Paradigms

Procedural Programming

Procedural programming is a way of programming that focuses on the process or the steps to accomplish a task. For example, if we had 100 numbers and wanted to know the average value of those 100 numbers, we could add the numbers and then divide by 100. This is one process to compute the average of numbers: add them and divide by the quantity of numbers. A Python procedural program for computing the average is shown in example 1.

```
# Example 1

def main():
    numbers = [87, 95, 72, 92, 95, 88, 84]
    total = 0
    for x in numbers:
        total += x
    average = total / len(numbers)
    print(f"average: {average:.2f}")

# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_1.py
average: 87.57
```

Notice that with procedural programming, we must write the process or the steps that are necessary to complete a task. Procedural programming is the type of programming that you did most often in CSE 110 and 111.

Declarative Programming

When we use declarative programming to program a computer, we do not focus on the process or steps to accomplish a task, but rather we focus on what we want from the task, or in other words, we focus on the desired result. Continuing the example of the average, with declarative programming, we focus on exactly what numbers we want averaged and tell the computer to compute that average for us. SQL is a declarative programming language used with relational databases. Example 2 contains SQL code that causes the computer to compute the average of a column of numbers.

```
-- Example 2  
SELECT AVG(numbers) FROM table;
```

Notice in example 2, that the code does not contain the steps required to compute the average. Someone else already wrote the code that contains those steps. Instead, the SQL code contains a command that tells the computer to compute the average of a column named *numbers*. The term "declarative programming" means that we write or declare what we want the computer to do. We do not tell the computer how to compute something. We declare what we want the computer to do, and the computer determines how to do it and then does it.

Functional Programming

When we use functional programming to program a computer, we focus on the functions necessary to accomplish a task. Mathematicians often find functional programming natural for them because they are accustomed to using functions while studying mathematics. In functional programming, functions are so important that we often pass functions into other functions. You did this in the checkpoint and team activity for lesson 11. Example 3 contains a functional programming solution to computing the average in Python.

```
# Example 3  
from functools import reduce  
  
def main():  
    numbers = [87, 95, 72, 92, 95, 88, 84]  
    func_add = lambda a, b: a + b  
    total = reduce(func_add, numbers)  
    average = total / len(numbers)  
    print(f"average: {average:.2f}")  
  
# Call main to start this program.  
if __name__ == "__main__":  
    main()
```

```
> python example_3.py  
average: 87.57
```

Notice how example 3 uses three functions: a lambda function, the reduce function, and the len function. Notice also that the lambda function is passed into the reduce function. Passing a function into a function is one of the marks of functional programming.

Object-Oriented Programming

Object-oriented programming is a programming paradigm based on the concept of objects. An **object** is a piece of a program that contains both data (also known as attributes) and functions (also known as methods).

When we write an object-oriented program, we combine data and functions together into objects. For example, if we were writing a registration program used by students to register for courses at a university, we would write code to create Student objects and Course objects. Each Student object would have data such as *given_name*, *family_name*, and *phone_number* and would have functions such as *register*, *enroll*, *drop*, and *withdraw*. Each Course object would have data such as *course_code*, *title*, *description*, and *list_of_students* and would have functions such as *get_students* and *take_role*.

Python includes many built-in and standard objects that a programmer can use to write programs. In fact, you have already used many objects in your programs. Python lists and dictionaries are objects and have attributes and methods. Readers and Writers from the csv module are also objects.

One of the marks of object-oriented programming is selecting attributes and calling methods using the **dot operator** (a period). The official name of the dot operator is **component selector**, but almost no one calls it that because the term "dot" is much easier to say than "component selector." The code in example 4 uses the dot operator (.) to call the append method.

```
# Example 4

def main():
    numbers = [87, 95, 72, 92, 95, 88, 84]
    numbers.append(78)
    numbers.append(72)
    print(numbers)

# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_4.py
[87, 95, 72, 92, 95, 88, 84, 78, 72]
```

There are several types of commands that are commonly found in object-oriented programs. These types of commands are so common, that a programmer must be able to recognize and write them. Three of these types of commands are:

1. Creating objects, for example:

```
obj = datetime.now()
```

2. Accessing the attributes of an object using the dot operator (.), for example:

```
year = obj.year
```

3. Calling the methods of an object using the dot operator (.), for example:

```
new_obj = obj.replace(year=2035)  
day_of_week = obj.weekday()
```

Python Lists Are Objects

In Python, lists are objects with attributes and methods, and a programmer can modify a list by calling those methods. The list methods are documented in a section of the Python Tutorial titled [More on Lists](#).

Example 5 contains a program that is similar to [example 2](#) in the prepare content of lesson 7. Now that you know what an object is, that objects have methods, and that Python lists are objects, this example code should make more sense than it did in lesson 7. Notice that the append method is called on lines 8–10, insert is called on line 13, index is called on line 17, pop is called on line 24, and remove is called on line 27.

```
1 # Example 5  
2  
3 def main():  
4     # Create an empty list that will hold fabric names.  
5     fabrics = []  
6  
7     # Add three elements at the end of the fabrics list.  
8     fabrics.append("velvet")  
9     fabrics.append("denim")  
10    fabrics.append("gingham")  
11  
12    # Insert an element at the beginning of the fabrics list.  
13    fabrics.insert(0, "chiffon")  
14    print(fabrics)  
15  
16    # Get the index where velvet is stored in the fabrics list.  
17    i = fabrics.index("velvet")  
18  
19    # Replace velvet with taffeta.  
20    fabrics[i] = "taffeta"  
21    print(fabrics)  
22  
23    # Remove the last element from the fabrics list.  
24    fabrics.pop()  
25  
26    # Remove denim from the fabrics list.  
27    fabrics.remove("denim")  
28    print(fabrics)  
29
```

```
30
31 # Call main to start this program.
32 if __name__ == "__main__":
33     main()
```

```
> python example_5.py
['chiffon', 'velvet', 'denim', 'gingham']
['chiffon', 'taffeta', 'denim', 'gingham']
['chiffon', 'taffeta']
```

Python Dictionaries Are Objects

Python dictionaries are objects with attributes and methods, and a programmer can modify a dictionary by calling those methods. There doesn't seem to be an official Python web page that documents the dictionary methods, so here is a list of the built-in dictionary methods:

Method	Description
d.clear()	Removes all the elements from the dictionary <i>d</i> .
d.copy()	Returns a copy of the dictionary <i>d</i> .
d.get(<i>key</i>)	Returns the value of the specified <i>key</i> . Calling the get method is almost equivalent to using square brackets ([and]) to find a key in a dictionary.
d.items()	Returns a list that contains the key value pairs that are in the dictionary <i>d</i> .
d.keys()	Returns a list that contains the keys that are in the dictionary <i>d</i> .
d.pop(<i>key</i>)	Removes the element with the specified <i>key</i> from the dictionary <i>d</i> .
d.update(<i>other</i>)	Updates the dictionary <i>d</i> with the key value pairs that are in the <i>other</i> dictionary.
d.values()	Returns a list that contains the values that are in the dictionary <i>d</i> .

The following example code, which is similar to [example 1](#) from the prepare content of lesson 8, calls dictionary methods at lines 20, 28, and 37.

```
1 # Example 6
2
3 def main():
4     # Create a dictionary with student IDs as
5     # the keys and student names as the values.
6     students = {
7         "42-039-4736": "Clint Huish",
8         "61-315-0160": "Amelia Davis",
9         "10-450-1203": "Ana Soares",
10        "75-421-2310": "Abdul Ali",
11        "07-103-5621": "Amelia Davis",
12        "81-298-9238": "Sama Patel"
13    }
14
15    # Get a student ID from the user.
```

```

16     id = input("Enter a student ID: ")
17
18     # Lookup the student ID in the dictionary and
19     # retrieve the corresponding student name.
20     name = students.get(id)
21
22     if name:
23         # Print the student name.
24         print(name)
25
26         # Remove the student that the user
27         # specified from the dictionary.
28         students.pop(id)
29     else:
30         print("No such student")
31     print()
32
33     # Use a for loop to print each key value pair
34     # in the dictionary. Of course, the code in
35     # the body of a loop can do much more with
36     # each key value pair than simply print it.
37     for key, value in students.items():
38         print(key, value)
39
40
41 # Call main to start this program.
42 if __name__ == "__main__":
43     main()

```

```

> python example_6.py
Enter a student ID: 81-298-9238
Sama Patel

```

```

42-039-4736 Clint Huish
61-315-0160 Amelia Davis
10-450-1203 Ana Soares
75-421-2310 Abdul Ali
07-103-5621 Amelia Davis

```

Summary

This lesson introduces you to object-oriented programming. You are learning that an object has data (attributes) and functions (methods) and that a programmer uses the dot operator (.) to access the attributes and call the methods in an object. Python lists and dictionaries are objects and contain attributes and methods.

12 Checkpoint: Using Objects

Purpose

Improve your ability to write object-oriented code.

Problem Statement

There are several types of commands that are commonly found in object oriented programs. These types of commands are so common, that a programmer must be able to recognize and write them. One of these types of commands is calling the methods of an object using the dot operator (.) as shown in this template:

```
variable = object.method(arg1, arg2, ...)
```

Helpful Documentation

In Python, lists are objects with attributes and methods, and a programmer can modify a list by calling those methods. The list methods are documented in a web page titled [More on Lists](#).

Assignment

Write a small Python program named `fruit.py` that demonstrates object oriented programming by modifying a list. Do the following:

1. Open a new blank file in VS Code and save it as `fruit.py`
2. Copy and paste this code at the top of your `fruit` program:

```
def main():
    # Create and print a list named fruit.
    fruit_list = ["pear", "banana", "apple", "mango"]
    print(f"original: {fruit_list}")
```
3. Add code to reverse and print `fruit_list`.
4. Add code to append "orange" to the end of `fruit_list` and print the list.
5. Add code to find where "apple" is located in `fruit_list` and insert "cherry" before "apple" in the list and print the list.

6. Add code to remove "banana" from *fruit_list* and print the list.
7. Add code to pop the last element from *fruit_list* and print the popped element and the list.
8. Add code to sort and print *fruit_list*.
9. Add code to clear and print *fruit_list*.
10. At the bottom of your program write a call to the `main` function.

Testing Procedure

Verify that your program works correctly by following each step in this testing procedure:

1. Run your program and ensure that your program's output is the same as the output shown below.

```
> python fruit.py
original: ['pear', 'banana', 'apple', 'mango']
reversed: ['mango', 'apple', 'banana', 'pear']
append orange: ['mango', 'apple', 'banana', 'pear', 'orange']
insert cherry: ['mango', 'cherry', 'apple', 'banana', 'pear', 'orange']
remove banana: ['mango', 'cherry', 'apple', 'pear', 'orange']
pop orange: ['mango', 'cherry', 'apple', 'pear']
sorted: ['apple', 'cherry', 'mango', 'pear']
cleared: []
```

Ponder

After you finish your program, examine the code in `main` and realize that you wrote object oriented code when you used the *fruit_list* and the dot operator (`.`) to call the `reverse`, `append`, `insert`, `remove`, `pop`, `sort`, and `clear` methods.

Sample Solution

When your program is finished, view the [sample solution](#) for this assignment to compare your solution to that one. Before looking at the sample solution, you should work to complete this checkpoint program. However, if you have worked on it for at least an hour and are still having problems, feel free to use the sample solution to help you finish your program.

Submission

When complete, report your progress in the associated I-Learn quiz.

lesson12/check_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 def main():
4     try:
5         # Create and print a list named fruit_list.
6         fruit_list = ["pear", "banana", "apple", "mango"]
7         print(f"original: {fruit_list}")
8
9         # Reverse and print the fruit_list list.
10        fruit_list.reverse()
11        print(f"reversed: {fruit_list}")
12
13        # Append "orange" to the end of the fruit_list and
14        # print the list.
15        fruit_list.append("orange")
16        print(f"append orange: {fruit_list}")
17
18        # Find where "apple" is located in the fruit_list and insert
19        # "cherry" before "apple" in the list and print the list.
20        index = fruit_list.index("apple")
21        fruit_list.insert(index, "cherry")
22        print(f"insert cherry: {fruit_list}")
23
24        # Remove "banana" from the fruit_list and print the list.
25        fruit_list.remove("banana")
26        print(f"remove banana: {fruit_list}")
27
28        # Pop (remove) the last element from the fruit_list
29        # and print the popped element and the list.
30        last = fruit_list.pop()
31        print(f"pop {last}: {fruit_list}")
32
33        # Sort and print the fruit_list.
34        fruit_list.sort()
35        print(f"sorted: {fruit_list}")
36
37        # Clear and print the fruit_list.
38        fruit_list.clear()
39        print(f"cleared: {fruit_list}")
40
41    except IndexError as index_err:
42        print(type(index_err).__name__, index_err, sep=": ")
43
44
45    # If this file is executed like this:
46    # > python check_solution.py
47    # then call the main function. However, if this file is simply
48    # imported (e.g. into a test file), then skip the call to main.
49    if __name__ == "__main__":
50        main()
```

12 Team Activity: Using Objects

Instructions

Work as a team as explained in the instructions for the [lesson 2 team activity](#).

Problem Statement

Almost all of the programs that you wrote for this course receive input from and print results to a terminal window. However, most users prefer to interact with a program through a graphical user interface (GUI) that contains icons, text fields, drop-down lists, buttons, etc. Within a GUI, the individual components (icons, text fields, etc.) are called widgets. Most libraries for creating GUIs use object oriented programming because each widget is an object with attributes and methods.

Assignment

As a team, write a Python program named `gui.py` that gets user input from a GUI, performs a simple calculation, and displays the result in a GUI.

Helpful Documentation

[GUI Programming with Tkinter](#)

Official documentation for the [tkinter module](#)

Steps

Do the following:

1. Download these two Python files: [heart_rate.py](#) and [number_entry.py](#) and save them in the same folder where you will save your Python program.
2. Open the `heart_rate.py` file in VS Code and run it. Notice how running the program opens a GUI where a user can enter his age and see heart rates for exercising. Experiment with the GUI by entering different ages, seeing the results, and clicking the "Clear" button.

3. Read the comments and examine the code in the `heart_rate.py` program. Notice the following:
 - a. To create its GUI, the `heart_rate` program uses the `tkinter` module which is imported at line 3.
 - b. The `main` function begins at line 7.
 - c. The `main` function creates a `tk.TK` root object and uses that object to create the main window for the program.
 - d. Beginning at line 49, the `populate_main_window` function creates labels, text entry boxes, and buttons and places them in a grid.
 - e. Inside the `populate_main_window` function, there are two nested functions named `calculate` and `clear`. The `calculate` function is called each time the user enters a digit in the age text field. The `clear` function is called each time the user clicks the "Clear" button.
4. Choose a simple calculation or formula for your program to calculate. You could choose one of these:
 - a. Area of a circle: $a = \pi r^2$
 - b. Swing time of a pendulum: $t = 2\pi\sqrt{\frac{n}{9.81}}$
 - c. Area of a rectangle: $a = wh$
 - d. Volume of a tire: $v = \frac{\pi w^2 a (w a + 2,540 d)}{10,000,000,000}$
5. Use the `heart_rate.py` program as a reference (you could even copy and paste parts of it or all of it) and write a Python program with a GUI that calculates the formula that you chose.

Core Requirements

1. Your program must include a GUI that opens when you run your program.
2. The GUI must allow a user to enter input.
3. When the user enters valid input, your program must compute correct results and display those results in the GUI.

Stretch Challenges

If your team finishes the core requirements in less than an hour, complete one or more of these stretch challenges. Note that the stretch challenges are optional.

1. Add a "Clear" button to your GUI that clears all inputs and outputs when the user clicks it.

2. Add a label that acts as a status bar at the bottom of your GUI. Your program should display an error message in the status bar when the user enters invalid input. Your program should clear the status bar when the user enters valid input.

Testing Procedure

Run your program and enter various inputs, including invalid values. Verify that your program doesn't crash, behaves as expected, and displays correct results.

Ponder

After you finish your program, examine the code in the `populate_main_window` function and realize that you wrote object oriented code when you used objects such as `lbl_age`, `lbl_width`, `txt_age`, and `txt_width` and the dot operator to call methods such as `grid`, `config`, `get`, and `clear`.

Sample Solution

Please work diligently with your team for the one hour meeting. After the meeting is over, please compare your approach to the [sample solution](#). Please **do not look at the sample solution** until you have either finished the program or diligently worked for at least one hour. At the end of the hour, if you are still struggling to complete the assignment, you may use the sample solution to help you finish.

Submission

When you have finished the activity, please report your progress via the associated I-Learn quiz. When asked about which of the requirements you completed, feel free to include any work done during the team meeting or after the meeting, including work done with the help of the sample solution, if necessary. In short, report on what you were able to accomplish, regardless of when you completed it or if you needed help from the sample solution.

lesson12/heart_rate.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 import tkinter as tk
4 import number_entry as nent
5
6
7 def main():
8     # Create the Tk root object.
9     root = tk.Tk()
10
11    # Create the main window. In tkinter,
12    # a window is also called a frame.
13    frm_main = tk.Frame(root)
14    frm_main.master.title("Heart Rate")
15    frm_main.pack(padx=4, pady=3, fill=tk.BOTH, expand=1)
16
17    # Call the populate_main_window function, which will add
18    # labels, text entry boxes, and buttons to the main window.
19    populate_main_window(frm_main)
20
21    # Start the tkinter loop that processes user events
22    # such as key presses and mouse button clicks.
23    root.mainloop()
24
25
26    # The controls in a graphical user interface (GUI) are called widgets,
27    # and each widget is an object. Because a GUI has many widgets and
28    # each widget is an object, the code to make a GUI usually has many
29    # variables to store the many objects. Because there are so many
30    # variable names, programmers often adopt a naming convention to help
31    # a programmer keep track of all the variables. One popular naming
32    # convention is to type a three letter prefix in front of the names
33    # of all variables that store GUI widgets, according to this list:
34    #
35    # frm: a frame (window) widget
36    # lbl: a label widget that displays text for the user to see
37    # ent: an entry widget where a user will type text or numbers
38    # btn: a button widget that the user will click
39
40
41 def populate_main_window(frm_main):
42     """Populate the main window of this program. In other words, put
43     the labels, text entry boxes, and buttons into the main window.
44
45     Parameter
46         frm_main: the main frame (window)
47     Return: nothing
48     """
49     # Create a label that displays "Age:"
50     lbl_age = tk.Label(frm_main, text="Age:")
51
52     # Create a integer entry box where the user will enter her age.
53     ent_age = nent.IntEntry(frm_main, 12, 90, width=5)
54
55     # Create a label that displays "Rates:"
56     lbl_rates = tk.Label(frm_main, text="Rates:")
```

```

58     # Create labels that will display the results.
59     lbl_slow = tk.Label(frm_main, width=4)
60     lbl_fast = tk.Label(frm_main, width=4)
61
62     # Create the Clear button.
63     btn_clear = tk.Button(frm_main, text="Clear")
64
65     # Layout all the labels, entry boxes, and buttons in a grid.
66     lbl_age.grid( row=0, column=0, padx=3, pady=3)
67     ent_age.grid( row=0, column=1, padx=3, pady=3)
68     lbl_rates.grid(row=0, column=2, padx=(30,3), pady=3)
69     lbl_slow.grid( row=0, column=3, padx=3, pady=3)
70     lbl_fast.grid( row=0, column=4, padx=3, pady=3)
71     btn_clear.grid(row=1, column=0, padx=3, pady=3, colspan=5, stick
72
73
74     # This function will be called each time the user releases a key.
75     def calculate(event):
76         """Compute and display the user's slowest
77         and fastest beneficial heart rates.
78         """
79         try:
80             # Get the user's age.
81             age = ent_age.get()
82
83             # Compute the user's maximum heart rate.
84             max_rate = 220 - age
85
86             # Compute the user's slowest and
87             # fastest beneficial heart rates.
88             slow = max_rate * 0.65
89             fast = max_rate * 0.85
90
91             # Display the slowest and fastest benificial
92             # heart rates for the user to see.
93             lbl_slow.config(text=f"{slow:.0f}")
94             lbl_fast.config(text=f"{fast:.0f}")
95
96         except ValueError:
97             # When the user deletes all the digits in the age
98             # entry box, clear the slowest and fastest labels.
99             lbl_slow.config(text="")
100            lbl_fast.config(text="")
101
102
103    # This function will be called each time
104    # the user presses the "Clear" button.
105    def clear():
106        """Clear all the inputs and outputs."""
107        ent_age.delete(0, tk.END)
108        lbl_slow.config(text="")
109        lbl_fast.config(text="")
110        ent_age.focus()
111
112
113    # Bind the calculate function to the age entry box
114    # so that the calculate function will be called when
115    # the user changes the text in the entry box.
116    ent_age.bind("<KeyRelease>", calculate)
117
118    # Bind the clear function to the clear button so

```

```
119      # that the clear function will be called when the
120      # user clicks the clear button.
121      btn_clear.config(command=clear)
122
123      # Give the keyboard focus to the age entry box.
124      ent_age.focus()
125
126
127      # If this file is executed like this:
128      # > python heart_rate.py
129      # then call the main function. However, if this file is simply
130      # imported (e.g. into a test file), then skip the call to main.
131      if __name__ == "__main__":
132          main()
```

lesson12/number_entry.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 """This module contains two classes, IntEntry and FloatEntry,
4 that allow a user to enter an integer or a floating point number.
5 """
6 import tkinter as tk
7 from numbers import Number
8
9 """
10 Can this user input become a number within the range [lower, upper]
11
12 All possibilities
13 [ < 0, < 0 ]    require -
14 [ < 0, 0 ]       allow -
15 [ < 0, > 0 ]    allow -
16 [ 0, > 0 ]      disallow -
17 [ > 0, > 0 ]    disallow -
18 """
19
20 class IntEntry(tk.Entry):
21     """An Entry widget that accepts only
22     integers between a lower and upper bound.
23     """
24     def __init__(self, parent, lower_bound, upper_bound, **kwargs):
25         super().__init__(parent)
26         assert isinstance(lower_bound, int), "lower_bound must be an in
27         assert isinstance(upper_bound, int), "upper_bound must be an in
28         assert lower_bound < upper_bound, \
29             "lower_bound must be less than upper_bound"
30
31         self.lower_bound = lower_bound
32         self.upper_bound = upper_bound
33         self.lower_entry = lower_bound if lower_bound <= 1 else 1
34         self.upper_entry = upper_bound if upper_bound >= -1 else -1
35         if "justify" not in kwargs:
36             kwargs["justify"] = "right"
37         if "width" not in kwargs:
38             kwargs["width"] = max(len(str(lower_bound)), len(str(upper_
39         kwargs["validate"] = "key"
40         kwargs["validatecommand"] = (parent.register(self.validate), "%
41         self.config(**kwargs)
42
43
44     def validate(self, value_if_allowed):
45         valid = False
46         try:
47             i = int(value_if_allowed)
48             valid = (str(i) == value_if_allowed and
49                     self.lower_entry <= i <= self.upper_entry)
50         except:
51             valid = (len(value_if_allowed) == 0 or
52                     (self.lower_entry < 0 and value_if_allowed == "-"))
53         return valid
54
55
56     def get(self):
57         """Return the integer that the user entered.""""
```

```

58     value = int(super().get())
59     if value < self.lower_bound or self.upper_bound < value:
60         raise ValueError("number must be between"
61                         f" {self.lower_bound} and {self.upper_bound}")
62     return value
63
64
65     def set(self, n):
66         """Display an integer for the user to see."""
67         assert isinstance(n, int), "n must be an integer"
68         assert self.lower_bound <= n <= self.upper_bound, \
69             f"n must be between {self.lower_bound} and {self.upper_bound}"
70         self.delete(0, tk.END)
71         self.insert(0, str(n))
72
73
74     class FloatEntry(tk.Entry):
75         """An Entry widget that accepts only
76         numbers between a lower and upper bound.
77         """
78         def __init__(self, parent, lower_bound, upper_bound, **kwargs):
79             super().__init__(parent)
80             assert isinstance(lower_bound, Number), "lower_bound must be a"
81             assert isinstance(upper_bound, Number), "upper_bound must be a"
82             assert lower_bound < upper_bound, \
83                 "lower_bound must be less than upper_bound"
84
85             self.lower_bound = lower_bound
86             self.upper_bound = upper_bound
87             self.lower_entry = lower_bound if lower_bound <= 0 else 0
88             self.upper_entry = upper_bound if upper_bound >= 0 else 0
89             vcmd = (parent.register(self.validate), "%P")
90             if not "justify" in kwargs:
91                 kwargs["justify"] = "right"
92             if not "width" in kwargs:
93                 kwargs["width"] = max(len(str(lower_bound)), len(str(upper_bound)))
94             self.config(validate="key", validatecommand=vcmd, **kwargs)
95
96         def validate(self, value_if_allowed):
97             valid = False
98             try:
99                 i = float(value_if_allowed)
100                valid = (self.lower_entry <= i <= self.upper_entry)
101            except:
102                valid = (len(value_if_allowed) == 0 or
103                         (self.lower_entry < 0 and value_if_allowed == "-"))
104            return valid
105
106
107     def get(self):
108         """Return the number that the user entered."""
109         return float(super().get())
110         if value < self.lower_bound or self.upper_bound < value:
111             raise ValueError("number must be between"
112                             f" {self.lower_bound} and {self.upper_bound}")
113         return value
114
115
116     def set(self, n):
117         """Display a number for the user to see."""
118         assert isinstance(n, Number), "n must be an integer"

```

```
119     assert self.lower_bound <= n <= self.upper_bound, \
120         f"n must be between {self.lower_bound} and {self.upper_bound}"
121     self.delete(0, tk.END)
122     self.insert(0, str(n))
```

lesson12/teach_solution.py

```
1 # Copyright 2020, Brigham Young University-Idaho. All rights reserved.
2
3 import math
4 import tkinter as tk
5 import number_entry as nent
6
7
8 def main():
9     # Create the Tk root object.
10    root = tk.Tk()
11
12    # Create the main window. In tkinter,
13    # a window is also called a frame.
14    frm_main = tk.Frame(root)
15    frm_main.master.title("Tire Volume")
16    frm_main.pack(padx=4, pady=3, fill=tk.BOTH, expand=1)
17
18    # Call the populate_main_window function, which will add
19    # labels, text entry boxes, and buttons to the main window.
20    populate_main_window(frm_main)
21
22    # Start the tkinter loop that processes user events
23    # such as key presses and mouse button clicks.
24    root.mainloop()
25
26
27    # The controls in a graphical user interface (GUI) are called widgets,
28    # and each widget is an object. Because a GUI has many widgets and
29    # each widget is an object, the code to make a GUI usually has many
30    # variables to store the many objects. Because there are so many
31    # variable names, programmers often adopt a naming convention to help
32    # a programmer keep track of all the variables. One popular naming
33    # convention is to type a three letter prefix in front of the names
34    # of all variables that store GUI widgets, according to this list:
35    #
36    # frm: a frame (window) widget
37    # lbl: a label widget that displays text for the user to see
38    # ent: an entry widget where a user will type text or numbers
39    # btn: a button widget that the user will click
40
41
42 def populate_main_window(frm_main):
43     """Populate the main window of this program. In other words, put
44     the labels, text entry boxes, and buttons into the main window.
45
46     Parameter
47         frm_main: the main frame (window)
48     Return: nothing
49     """
50
51     # Create labels for the text fields and the results.
52     lbl_width = tk.Label(frm_main, text="Width (mm):")
53     lbl_ratio = tk.Label(frm_main, text="Aspect Ratio:")
54     lbl_diam = tk.Label(frm_main, text="Diameter (in):")
55     lbl_vol = tk.Label(frm_main, text="Volume (liters):")
56
57     # Create three text fields.
58     ent_width = nent.IntEntry(frm_main, 80, 300, width=5)
```

```

58     ent_ratio = nent.FloatEntry(frm_main, 30, 90, width=5)
59     ent_diam = nent.FloatEntry(frm_main, 10, 30, width=5)
60
61     # Create a label to display the result.
62     lbl_result = tk.Label(frm_main, width=8, anchor="w")
63
64     # Create the Clear button.
65     btn_clear = tk.Button(frm_main, text="Clear")
66
67     # Layout all the labels, text fields, and buttons in a grid.
68     lbl_width.grid( row=0, column=0, padx=3, pady=2, sticky="e")
69     ent_width.grid( row=0, column=1, padx=3, pady=2, sticky="w")
70     lbl_ratio.grid( row=1, column=0, padx=3, pady=2, sticky="e")
71     ent_ratio.grid( row=1, column=1, padx=3, pady=2, sticky="w")
72     lbl_diam.grid( row=2, column=0, padx=3, pady=2, sticky="e")
73     ent_diam.grid( row=2, column=1, padx=3, pady=2, sticky="w")
74     lbl_vol.grid( row=3, column=0, padx=3, pady=2, sticky="e")
75     lbl_result.grid(row=3, column=1, padx=3, pady=2, sticky="w")
76     btn_clear.grid( row=3, column=2, padx=3, pady=2)
77
78
79     # This function is called each time the user releases a key.
80     def calculate(event):
81         """Compute the approximate volume of a tire in liters."""
82         try:
83             # Get the user input.
84             w = ent_width.get()
85             a = ent_ratio.get()
86             d = ent_diam.get()
87
88             # Compute the tire volume in liters.
89             v = (math.pi * w * w * a * (w * a + 2540 * d)) / 10_000_000
90
91             # Display the volume rounded to one place after
92             # the decimal for the user to see.
93             lbl_result.config(text=f"{v:.1f}")
94
95         except ValueError:
96             # When the user deletes all the digits in one
97             # of the text fields, clear the result labels.
98             lbl_result.config(text="")
99
100
101    # This function is called each time
102    # the user clicks the "Clear" button.
103    def clear():
104        """Clear all the inputs and outputs."""
105        ent_width.delete(0, tk.END)
106        ent_ratio.delete(0, tk.END)
107        ent_diam.delete(0, tk.END)
108        lbl_result.config(text="")
109        ent_width.focus()
110
111
112    # Bind the calculate function to the three text fields
113    # so that the calculate function will be called when
114    # the user changes the text in the text fields.
115    ent_width.bind("<KeyRelease>", calculate)
116    ent_ratio.bind("<KeyRelease>", calculate)
117    ent_diam.bind("<KeyRelease>", calculate)
118
```

```
119 # Bind the clear function to the clear button so
120 # that the clear function will be called when the
121 # user clicks the clear button.
122 btn_clear.config(command=clear)
123
124 # Give the keyboard focus to the width text field.
125 ent_width.focus()
126
127
128 # If this file is executed like this:
129 # > python teach_solution.py
130 # then call the main function. However, if this file is simply
131 # imported (e.g. into a test file), then skip the call to main.
132 if __name__ == "__main__":
133     main()
```

12 Prove Milestone: Student Chosen Program

Purpose

Prove that you can write a significant Python program that solves a real-world problem and is well organized with functions.

Assignment

Continue developing the program that you started in the [prove assignment for lesson 11](#). Your program must include multiple functions that you verify are correct with test functions and pytest.

Submission

At the end of this lesson and lesson 13, you must submit a description of your work, and your teacher or teaching assistant will grade your work according to the following rubrics.

Lesson 12 Rubric

1. Time—50%: Did you spend at least three hours on your Python program or test functions during the current lesson?
2. Organization—20%:
 - a. Is your program organized into multiple functions?
 - b. Does each function in your program perform just one task?
3. Progress—20%: Did you complete some significant part of your program during the current week?
4. Description—10%: Is the description of your work for this lesson complete and easily understandable? Your description should include the following:
 - a. A list of the function names in your program.
 - b. A list of the test function names in your test code.
 - c. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
 - d. A description or list of the work that you finished on your program.

Lesson 13 Rubric

1. Time—50%: Did you spend at least six hours on your Python program or test functions during the current lesson?
2. Description—10%: Is the description of your work for this lesson complete and easily understandable? Your description should include the following:
 - a. A list of the function names in your program.
 - b. A list of the test function names in your test code.
 - c. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
 - d. A description or list of the work that you finished on your program.
3. Python program file—25%: Upload your Python program. Your teacher will evaluate it according to these criteria:
 - a. Your program is divided into functions and each function performs one task only.
 - b. Your program effectively uses existing Python modules such as math, random, requests, pandas, and tkinter.
 - c. Your program performs a significant real world task.
4. Python test file—15%: Upload your Python test file. Your teacher will evaluate it according to these criteria:
 - a. Each testable program function is covered (tested) by one test function.
(Some functions, especially main and those that create GUIs are difficult to test and don't need to have a corresponding test function.)
 - b. Each test function completely exercises (tests) its corresponding program function. In other words, the test function calls the program function multiple times with different arguments, including unusual or unexpected values.

13 Prove Assignment: Student Chosen Program

Purpose

Prove that you can write a significant Python program that solves a real-world problem and is well organized with functions.

Assignment

Finish developing the program that you started in the [prove assignment for lesson 11](#). Your program must include multiple functions that you verify are correct with test functions and pytest.

Submission

At the end of this lesson, you must submit a description of your work, and your teacher or teaching assistant will grade your work according to the following rubric.

Lesson 13 Rubric

1. Time—50%: Did you spend at least six hours on your Python program or test functions during the current lesson?
2. Description—10%: Is the description of your work for this lesson complete and easily understandable? Your description should include the following:
 - a. A list of the function names in your program.
 - b. A list of the test function names in your test code.
 - c. A list of the documentation that you read, the videos that you watched, and the coding experiments that you tried.
 - d. A description or list of the work that you finished on your program.
3. Python program file—25%: Upload your Python program. Your teacher will evaluate it according to these criteria:
 - a. Your program is divided into functions and each function performs one task only.
 - b. Your program effectively uses existing Python modules such as math, random, requests, pandas, and tkinter.
 - c. Your program performs a significant real world task.

4. Python test file—15%: Upload your Python test file. Your teacher will evaluate it according to these criteria:
 - a. Each testable program function is covered (tested) by one test function.
(Some functions, especially main and those that create GUIs are difficult to test and don't need to have a corresponding test function.)
 - b. Each test function completely exercises (tests) its corresponding program function. In other words, the test function calls the program function multiple times with different arguments, including unusual or unexpected values.

14 Prepare: Conclusion

Congratulations! You successfully made it to the final lesson of CSE 111.

During this course you learned to write programs organized with functions to solve significant problems in a variety of domains, which was the major outcome of this course. In addition, you learned to do the following:

- Research modules, objects, and functions written by others and use them in your programs.
- Write programs that can detect and recover from invalid conditions (exceptions).
- Follow good practices in designing, writing, and debugging functions, including writing each function so that it performs one task only and writing test functions to verify that program functions are correct.

Each of these skills will transfer to other programming languages that you learn in your future. Also, researching modules, objects, and functions written by others gave you important practice in lifelong learning. Even though the work of this course may have been challenging at times, you showed skill in thinking analytically, breaking down problems into manageable pieces, working with others, and debugging abstract problems. These are skills that will transfer to any field of study or career path that you choose.

The final two tasks in CSE 111 are:

1. Write a personal reflection document.
2. Complete a one question survey.

Both of these two final tasks are in I-Learn. Please find them in the Lesson 14 module and complete both of them. May you never confuse Python tuples (), lists [], sets {}, and dictionaries {}:, and in all seriousness, may the Lord bless all your righteous endeavors.