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, including quizzes and due dates.

Overview

Syllabus

Help

Lesson 1 - Getting Started

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
Prepare		user-defined function, header, parameter, body, return, main
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
Prepare		syntax error, logic error, error messages, print statements, test functions, debugger
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
Prepare		text files, open, for each line, CSV files, csv module
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

Activity	Domain Topics	Computing Topics
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

Activity	Domain Topics	Computing Topics
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 project	

Lesson 12 - Using Objects

Activity	Domain Topics	Computing Topics
Prepare		creating an object, dot operator, attributes, methods, lists and dictionaries as objects
Prepare: Checkpoint	fruit	modify a list using object oriented programming
Teach: Team Activity	GUI	creating objects, accessing attributes, calling methods
Prove: Milestone (block) (semester)	student chosen project	

Lesson 13 - Student Chosen Project

Activity	Domain Topics	Computing Topics
No preparation content, checkpoint, or team activity. Students work on their own chosen project.		
Prove: Assignment (block) (semester)	student chosen project	

Lesson 14 - Conclusion

Activity	Domain Topics	Computing Topics

Search CSE 111 Content

Enter search text Search

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
- variable scope, arguments, named arguments

- exception handling
- lists, dictionaries
- text files, CSV files, csv Reader
- testing using pytest
- nested functions, lambda functions, higher order functions

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 preparation 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 Range		**Letter Grade**
93%	100.00%	A
90%	92.99%	A−
87%	89.99%	B+
83%	86.99%	B
80%	82.99%	B−
77%	79.99%	C+
73%	76.99%	C
70%	72.99%	C−
67%	69.99%	D+
63%	66.99%	D
60%	62.99%	D−
0%	59.99%	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 true***.

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. 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. Additional information about Disability Services resources can be found at www.byui.edu/disabilities.

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.

```
# This is a comment because it has
# hash symbols at the beginning.
```

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.

```
length = 5
time = 7.2
in_flight = True
first_name = "Cho"
```

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:

  ```
  greeting = "Hello"
  text = "23"
  ```

- 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.

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.

```
text = input("Please enter your name: ")
color = input("What is your favorite color? ")
```

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).

```
print(f"Heart rate: {rate}")
```

The Python program in example 1 creates ten different variables. Some of the variables are of type `str`, some 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.

```
# Example 1

# Create variables of different data types and then
# print the variable names, data types, and values.

a = "Her name is "  # string
b = "Isabella"      # string
c = a + b           # string plus string makes string
print(f"a: {type(a)} {a}")
print(f"b: {type(b)} {b}")
print(f"c: {type(c)} {c}")
print()
```

```

d = False  # boolean
e = True   # boolean
print(f"d: {type(d)} {d}")
print(f"e: {type(e)} {e}")
print()

f = 15     # int
g = 7.62   # float
h = f + g  # int plus float makes float
print(f"f: {type(f)} {f}")
print(f"g: {type(g)} {g}")
print(f"h: {type(h)} {h}")
print()

i = "True"   # string because of the surrounding quotes
j = "2.718"  # string because of the surrounding quotes
print(f"i: {type(i)} {i}")
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
```

```python
# The int and float functions convert a string to a number.
p = int(input("Please enter a number: "))        # int
q = float(input("Please enter another number: "))  # float
r = p + q                   # int plus float makes float
print(f"p: {type(p)} {p}")
print(f"q: {type(q)} {q}")
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	|
* / // %	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.

```python
 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:

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal to
!=	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:
```

```python
    rate = 0.10
elif cost < 250:
    rate = 0.15
elif cost < 400:
    rate = 0.18
else:
    rate = 0.20

# Compute the discount amount
# and the discounted cost.
discount = cost * rate
cost -= discount

# Print the discounted cost for the user to see.
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 246.00
```

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:

```python
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`

02 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
```

Notice in the previous code example that to call the `input` function, the programmer wrote the name of the function, `input`, followed by parentheses. The programmer did the same to call the `print` function.

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

1. The name of the function
2. The parameters that the function accepts
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 accepts 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 accepts 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 preparation 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 argument is 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.

```python
 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.
```

Video

The following video shows a BYU-Idaho student writing Python code that calls built-in functions.

> Calling Built-in Functions (11 minutes)

Tutorial

If you are uncertain about any of the concepts in the Concepts 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)
   ```

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`:

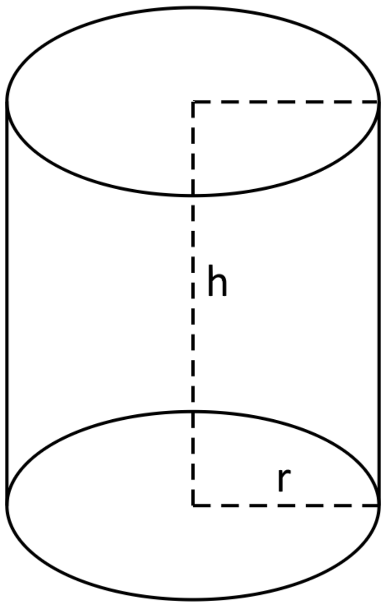
```
def draw_circle(x, y, radius):
```

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.

```python
# Example 4

import math

# 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}")
```

```
> python example_4.py
Enter the radius in centimeters: 3
Enter the height in centimeters: 8
Volume: 226.19
```

In a large program, writing statements outside a function can lead to poor organization. Professional software developers write statements inside a function whenever possible. Beginning with this lesson, you will do the following in each program:

1. Write nearly all statements inside a user-defined function.
2. Write a user-defined function named `main`, which contains the beginning statements of your program.
3. Write one or more user-defined functions that have parameteters, perform calculations and other useful work, and return a result to the call point.
4. Write a call to the `main` function at the bottom of your program.

Example 5 contains the same Python program as example 4 except most of the statements are inside a user-defined function named `main`.

```python
# Example 5

```

```
 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: 3
Enter the height in centimeters: 8
Volume: 226.19
```

Notice the call to the `main` function in example 5 at line 19. 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 write 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
```

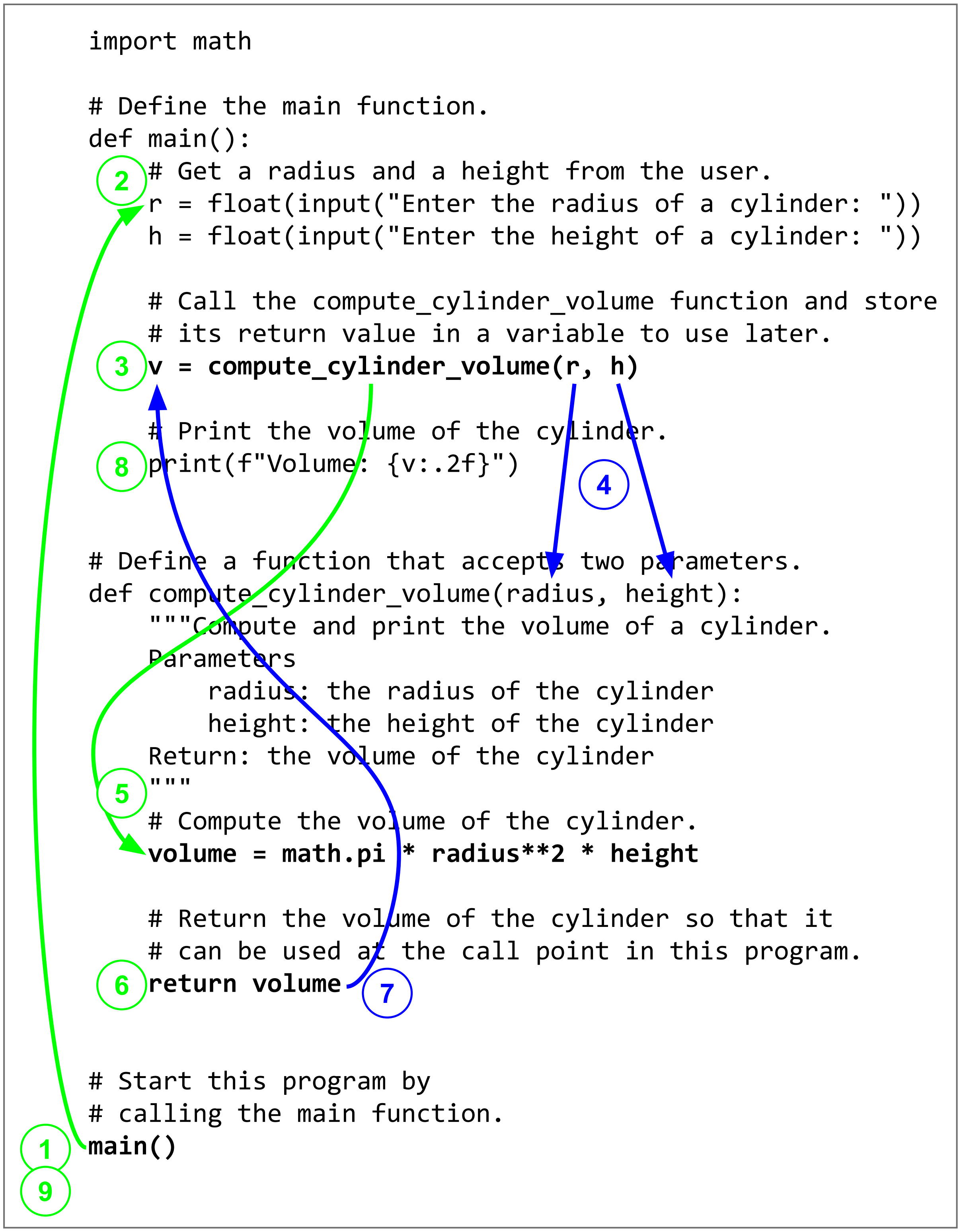
```python
import math

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

    # Print the volume of the cylinder.
    print(f"Volume: {volume:.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 the
    # volume can be used somewhere else in the program.
    # The returned result will be available wherever
    # this function was called.
    return volume


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



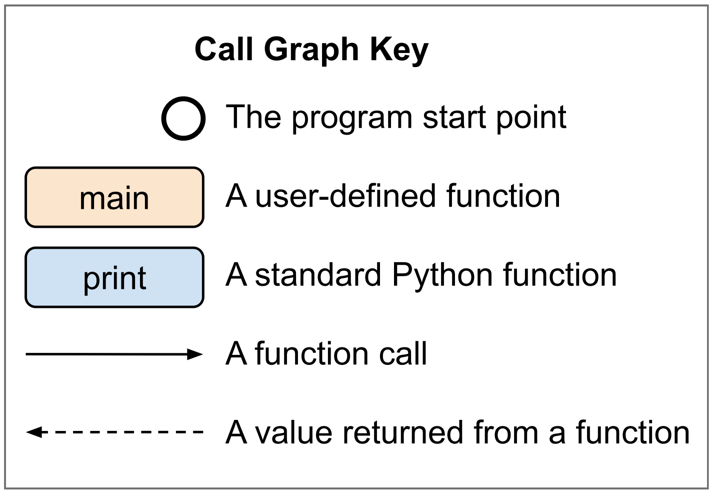
```python
import math

# 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:

A. 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).
B. 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.

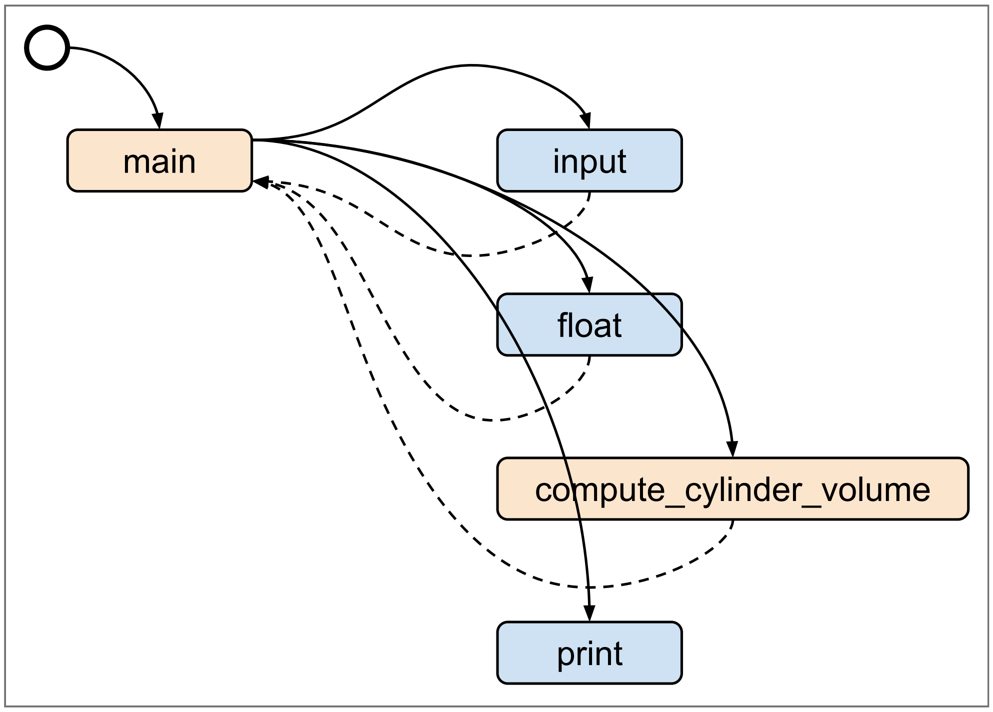
Call Graphs

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 preparation content, if the concepts still seem confusing or vague to you, pray and ask Heavenly Father to help you understand the concepts. Then watch the videos and read the concepts ***again***.

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.

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 0100000000110111101100110011001100110011001100110011001100110011. This binary number is actually 23.699999999999999928945726424 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:

```
# test_weather.py

from weather import cels_from_fahr
from pytest import approx
import pytest

def test_cels_from_fahr():
    """Test the cels_from_fahr function by calling it and
    comparing the values it returns to the expected values.
    Notice this test function uses pytest.approx to compare
    floating point numbers.
    """
```

```
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.

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.

```python
# Example 1

def main():
    print("Are you surprised, Clark?)

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

```
8  if __name__ == "__main__":
9      main()
```

```
> python surprise.py
  File "C:\Users\cse111\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.

```python
# Example 2

def main():
    print("This program computes and prints the remaining")
    print("balance for a loan with a fixed annual percentage")
    print("rate and a fixed number of payments per year.")
    print()
    print("Please enter the following five values.")

    principal = float(input("Principal amount: "))
    annual_rate = float(input("Annual percentage rate: "))
    years = int(input("Number of years in the life of the loan: "))
    payments_per_year = int(input("Number of payments per year: "))
    number_paid = int(input("Number of payments already paid: "))

    balance = compute_balance(principal, annual_rate, years,
            payments_per_year, number_paid)

    print()
    print(f"Balance remaining: {balance}")


def compute_balance(princ, ar, years, ppy, ptd):
    """Compute and return the balance remaining for a loan."""
    payment = compute_payment(princ, ar, years, ppy)

    print()
    print(f"compute_balance({princ}, {ar}, {years}, {ppy}, {ptd})")

    rate = ar / ppy
    power = (1 + rate) ** ptd
    print(f"    payment: {payment}  rate: {rate}  power: {power}")

    balance = princ * power - payment * (power - 1) / rate
    print(f"    balance: {balance:.2f}")

    return round(balance, 2)


def compute_payment(princ, ar, years, ppy):
    """Compute and return the payment per period for a loan."""
    print()
    print(f"compute_payment({princ}, {ar}, {years}, {ppy})")

    rate = ar / ppy
    n = years * ppy
    print(f"    rate: {rate}  n: {n}")

    payment = princ * rate / (1 - (1 + rate) ** -n)
    print(f"    payment: {payment:.2f}")

    return round(payment, 2)


```

```python
# Start this program by
# calling the main function.
if __name__ == "__main__":
    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 (17 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.

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.

```python
# 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.

```python
# 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

A programmer can cause a computer to repeat a group of statements by writing `for` and `while` loops.

for loop

A `for` loop iterates over a sequence, such as a list. This means a `for` loop causes the computer to repeatedly execute the statements in the body of the `for` loop, once for each element in the sequence. In example 3, consider the list of colors at line 5 and the `for` loop at lines 8–9. Notice how the `for` loop causes the computer to repeat line 9 once for each element in the *colors* list. Of course, the code in the body of a loop can do much more with each element than simply print it.

```python
 1  # Example 3
 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
12  # Call main to start this program.
13  if __name__ == "__main__":
14      main()
```

```
> python example_3.py
red
orange
yellow
green
blue
```

Notice in example 3 at lines 8–9 that just like `if` statements in Python, the body of a loop starts and ends with indentation.

range function

The Python built-in range function creates and returns a sequence of numbers. The `range` function accepts one, two, or three parameters as shown in example 4 and its output. Many programmers use the `range` function in a `for` loop to cause the computer to repeat code once for each number in a range of numbers. Example 4 shows four `for` loops that iterate over a range of numbers.

```python
# Example 4

def main():
    # Count from zero to nine by one.
    for i in range(10):
        print(i)
    print()

    # Count from five to nine by one.
    for i in range(5, 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_4.py
0
1
2
⋮
8
9

5
6
7
8
9

0
2
4
6
8

100
97
94
 ⋮
73
70
```

In example 5 at lines 8–9 and lines 15–17, there are two `for` loops. Both loops print each element from a list named *colors*. The first loop iterates over the elements in the *colors* list. The second loop uses the built-in `len` and `range` functions to iterate 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 5
 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          # Use the index i to retrieve
17          # an element from the list.
18          color = colors[i]
19
20          print(color)
21
22
23  # Call main to start this program.
24  if __name__ == "__main__":
25      main()
```

```
> python example_5.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 statement

A `break` statement causes a loop to end early. In example 6 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 6
 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_6.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 loop

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 7 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 7
 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:
```

```python
        print(f"list1 and list2 differ at index {index}")


def compare_lists(list1, list2):
    """Compare the contents of two lists. If the contents
    of the two lists are not equal, return the index of
    the first difference. If the contents of the two lists
    are equal, return -1.

    Parameters
        list1: a list
        list2: another list
    Return: an index or -1
    """
    # Get the length of the shortest list.
    length1 = len(list1)
    length2 = len(list2)
    limit = min(length1, length2)

    # Begin at the first index (0) and repeat until the
    # computer finds two elements that are not equal or
    # until the computer reaches the end of the shortest
    # list, whichever comes first.
    i = 0
    while i < limit:
        # Retrieve one element from each list.
        element1 = list1[i]
        element2 = list2[i]

        # If the two elements are not
        # equal, quit the while loop.
        if element1 != element2:
            break

        # Add one to the index variable.
        i += 1

    # If the length of both lists are equal and the
    # computer verified that all elements are equal,
    # set i to -1 to indicate that the contents of
    # the two lists are equal.
    if length1 == length2 == i:
        i = -1

    return i


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_7.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 8 shows how to create a compound list, retrieve one inner list from the compound list, and retrieve an individual number from the inner list.

```python
# Example 8

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_8.py
height: 2.3
```

Example 9 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.

```python
1   # Example 9
2
3   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]
    ]

    total_fruit_amount = 0

    # This loop will repeat once for each inner list
    # in the apple_tree_data compound list.
    for inner_list in apple_tree_data:

        # Retrieve the fruit amount from
        # the current inner list.
        fruit_amount = inner_list[FRUIT_AMOUNT_INDEX]

        # Print the fruit amount for the current tree.
        print(fruit_amount)

        # Add the current fruit amount to the total.
        total_fruit_amount += fruit_amount

    # Print the total fruit amount.
    print(f"Total fruit amount: {total_fruit_amount:.1f}")


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_9.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 10 and the output of that program. The program in example 10 contains two integer variables named *x* and *y*. The program in example 10 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 10
 2
 3  def main():
 4      x = 17
 5      y = x
 6      print(f"Before changing x: x {x}  y {y}")
 7      x += 1
 8      print(f"After changing x:  x {x}  y {y}")
 9
10  # Call main to start this program.
11  if __name__ == "__main__":
12      main()
```

```
> python example_10.py
Before changing x: x 17  y 17
After changing x:  x 18  y 17
```

Example 11 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 11, 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 11 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 11
 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_11.py
Before changing lx: lx [7, -2]  ly [7, -2]
After changing lx:  lx [7, -2, 5]  ly [7, -2, 5]
```

From examples 10 and 11, 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 12 with two functions named `main` and `modify_args`. The program in example 12 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 12
 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_12.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 12, 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 preparation 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.

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	|
10-450-1203	Ana Soares	5 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.

```python
# Example 1

def main():
    # 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"
    }

    # Add an item to the dictionary.
    students["81-298-9238"] = "Sama Patel"

    # Remove an item from the dictionary.
    students.pop("61-315-0160")

    # Get the number of items in the dictionary.
    length = len(students)
    print(f"length: {length}")

    # Print the entire dictionary.
    print(students)
    print()

    # 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")


# Call main to start this program.
if __name__ == "__main__":
    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.
29
```

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.

```
# Example 4

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]
    }

    # These are the indexes of the elements in the value lists.
    GIVEN_NAME_INDEX = 0
    SURNAME_INDEX = 1
    EMAIL_INDEX = 2
    CREDITS_INDEX = 3

    # 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 value, which is a list.
        value = students[id]

        # Retrieve the student's given name (first name) and
        # surname (last name or family name) from the list.
        given_name = value[GIVEN_NAME_INDEX]
        surname = value[SURNAME_INDEX]

        # Print the student's name.
        print(f"{given_name} {surname}")

    else:
        print("No such student")


# Call main to start this program.
if __name__ == "__main__":
    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],
```

```python
        "81-298-9238": ["Sama", "Patel", "pat21004@byui.edu", 8]
    }

    # These are the indexes of the elements in the value lists.
    GIVEN_NAME_INDEX = 0
    SURNAME_INDEX = 1
    EMAIL_INDEX = 2
    CREDITS_INDEX = 3

    total = 0

    # For each item in the list add the number
    # of credits that the student has earned.
    for key, value in students.items():

        # Retrieve the number of credits from the value list.
        credits = value[CREDITS_INDEX]

        # Add the number of credits to the total.
        total += credits

    print(f"Total credits earned by all students: {total}")


# Call main to start this program.
if __name__ == "__main__":
    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 compares lists to dictionaries and categorizes their traits as same, similar, or different.

	Lists	Dictionaries
Similar	A list can store many *elements*.	A dictionary can store many *items*.
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.
	Lists were designed for efficiently storing elements. Lists use less memory than dictionaries. However,	Dictionaries were designed for quickly finding items. Finding an item in a dictionary is fast. However,

	Lists	**Dictionaries**
	finding an element in a list is relatively slow.	dictionaries use more memory than lists.
	A programmer uses square brackets ([and]) to create a list.	A programmer uses curly braces ({ and }) to create a dictionary.

```
# Create a list of cities.
cities_list = ["Delhi", "Lagos", "Dallas"]

# Create a dictionary of people.
people_dict = {
    "P203": "Ignacio Torres",
    "P445": "Whitney Nelson",
    "P128": "Yasmin Li"
}
```

Same	Lists are mutable, which means a program can add and remove elements after a list is created.	Dictionaries are mutable, which means a program can add and remove items after a dictionary is created.
Different	A programmer calls the `insert` and `append` methods to add an element to a list.	A programmer uses square brackets ([and]) to add an item to a dictionary.

```
# Add two cities to the cities list.
cities_list.insert(1, "Paris")
cities_list.append("Tokyo")

# Add two people to the people dictionary.
people_dict["P205"] = "Liam Myers"
people_dict["P317"] = "Davina Patel"
```

Same	To cause the computer to check whether an element is in a list, a programmer uses the `in` keyword.	To cause the computer to check whether an element is in a dictionary, a programmer uses the `in` keyword.

```
if "Paris" in cities_list:
    print("Paris is in the list of cities.")
```

	Lists	Dictionaries

```
if "P203" in people_dict:
    print("P203 is in the dictionary of people.")
```

Different	A programmer uses the `index` method to find an element in a list.	A programmer uses square brackets ([and]) and a key to find an item in a list.

```
# Find Dallas in the cities list.
index = cities_list.index("Dallas")

# Find person P128 in the people dictionary.
person_name = people_dict["P128"]
```

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.

```
# Retrieve the element stored at
# index 2 in the cities list.
city_name = cities_list[2]

# Find person P128 in the people dictionary
# and retrieve the corresponding value.
person_name = people_dict["P128"]
```

	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.

```
# Change the city name at index 2 to London.
cities_list[2] = "London"

# Change the name of person P205 to Finn Meyers.
people_dict["P205"] = "Finn Myers"
```

	A programmer can use a `for` loop to	A programmer can use a `for` loop to

	Lists	**Dictionaries**
	process all the elements in a list.	process all the items in a dictionary.

```
# Process all the elements in the cities list.
for city_name in cities_list:
    print(city_name)

# Process all the items in the people dictionary.
for person_key, person_name in people_dict.items():
    print(person_name)
```

Same	A programmer uses the pop method to remove an element from a list.	A programmer uses the pop method to remove an item from a dictionary.

```
# Remove the element at index 3
# from the cities list.
cities_list.pop(3)

# Remove the key "P203" and its
# value from the people dictionary.
people_dict.pop("P203")
```

	Lists are passed by reference into a function.	Dictionaries are passed by reference into a function.

```
# Call the draw_chart function and pass
# the citites list to that function.
draw_chart(cities_list)

# Call the hire_people function and pass
# the people dictionary to that function.
hire_people(people_dict)
```

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.

```python
# Example 7

def main():
    # Create a list that contains five student numbers.
    numbers = ["42-039-4736", "61-315-0160",
            "10-450-1203", "75-421-2310", "07-103-5621"]

    # Create a list that contains five student names.
    names = ["Clint Huish", "Amelia Davis",
            "Ana Soares", "Abdul Ali", "Amelia Davis"]

    # Convert the numbers list and names list into a dictionary.
    student_dict = dict(zip(numbers, names))

    # Print the entire student dictionary.
    print("Dictionary:", student_dict)
    print()

    # Convert the student dictionary into
    # two lists named keys and values.
    keys = list(student_dict.keys())
    values = list(student_dict.values())

    # Print both lists.
    print("Keys:", keys)
    print()
    print("Values:", values)


# Call main to start this program.
if __name__ == "__main__":
    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.

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      # Create an empty list that will store
21      # the lines of text from the text file.
22      text_list = []
23  
24      # Open the text file for reading and store a reference
25      # to the opened file in a variable named text_file.
26      with open(filename, "rt") as text_file:
27  
28          # Read the contents of the text
29          # file one line at a time.
30          for line in text_file:
31  
32              # Remove white space, if there is any,
33              # from the beginning and end of the line.
34              clean_line = line.strip()
35  
36              # Append the clean line of text
37              # onto the end of the list.
38              text_list.append(clean_line)
39  
40      # Return the list that contains the lines of text.
41      return text_list
42  
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]
```

```python
            num_employees = int(row_list[NUM_EMPS_INDEX])
            num_patients = int(row_list[NUM_PATIENTS_INDEX])

            # Compute the number of patients per
            # employee for the current dental office.
            patients_per_emp = num_patients / num_employees

            # If the current dental office has more
            # patients per employee than the running
            # maximum, assign running_max and most_office
            # to be the current dental office.
            if patients_per_emp > running_max:
                running_max = patients_per_emp
                most_office = company

    # Print the results for the user to see.
    print(f"{most_office} has {running_max:.1f}"
            " patients per employee")


# Call main to start this program.
if __name__ == "__main__":
    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.

```python
# Example 4

import csv

def main():
    # Read the contents of the dentists.csv file
    # into a compound list.
    dentists_list = read_compound_list("dentists.csv")

    # Print the entire list.
    print(dentists_list)


```

```python
def read_compound_list(filename):
    """Read the contents of a CSV file into a compound
    list and return the list. Each element in the
    compound list will be a small list that contains
    the values from one row of the CSV file.

    Parameter filename: the name of the CSV file to read
    Return: a list of lists that contain strings
    """
    # Create an empty list that will
    # store the data from the CSV file.
    compound_list = []

    # Open the CSV file for reading and store a reference
    # to the opened file in a variable named csv_file.
    with open(filename, "rt") as csv_file:

        # Use the csv module to create a reader object
        # that will read from the opened CSV file.
        reader = csv.reader(csv_file)

        # Read the rows in the CSV file one row at a time.
        # The reader object returns each row as a list.
        for row_list in reader:

            # If the current row is not blank,
            # append it to the compound_list.
            if len(row_list) != 0:

                # Append one row from the CSV
                # file to the compound list.
                compound_list.append(row_list)

    # Return the compound list.
    return compound_list


# Call main to start this program.
if __name__ == "__main__":
    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 progam 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.

```python
 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_dictionary("dentists.csv", PHONE_INDEX)
15
16      # Print the dentists compound dictionary.
17      print(dentists_dict)
18
19
20  def read_dictionary(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      # Create an empty dictionary that will
32      # store the data from the CSV file.
33      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)
```

```python

        # The first row of the CSV file contains column
        # headings and not data, so this statement skips
        # the first row of the CSV file.
        next(reader)

        # Read the rows in the CSV file one row at a time.
        # The reader object returns each row as a list.
        for row_list in reader:

            # If the current row is not blank, add the
            # data from the current to the dictionary.
            if len(row_list) != 0:

                # From the current row, retrieve the data
                # from the column that contains the key.
                key = row_list[key_column_index]

                # Store the data from the current
                # row into the dictionary.
                dictionary[key] = row_list

    # Return the dictionary.
    return dictionary


# Call main to start this program.
if __name__ == "__main__":
    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…]}
```

Additional Information

The following tutorials contain additional information that you may find helpful. You are not required to read these tutorials.

Python "for" Loops

Writing Text 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.

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. You will need to write code in your programs to handle exceptions.

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 9:

```
except ZeroDivisionError as zero_div_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 (line 19) and a bare `except` block (line 23).

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 preparation 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 to the `round` function. This causes the computer to raise a `TypeError` because the `round` function cannot round a string to an integer. It can round only a number to an integer. The output below example 2 shows that the computer raised a `TypeError`.

```
# 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`.

```python
# 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.

```python
# 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`.

```python
# 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:

```python
    # 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")
```

FileNotFoundError

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 `FileNotFoundError`. Example 8 contains code where such an error might occur.

```
# Example 8

def main():
    try:
        with open("products.vcs", "rt") as products_file:
            for row in products_file:
                print(row)

    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 row in contacts_file:
                print(row)

    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:
```

```python
        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 made today: 350.4
Error: invalid literal for int() with base 10: '350.4'
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`.

```python
1   # Example 11
2
3   import csv
4
5   DATE_INDEX = 0
6   START_MILES_INDEX = 1
```

```python
END_MILES_INDEX = 2
GALLONS_INDEX = 3


def main():
    try:
        # Open the fuel_usage.csv file.
        filename = "fuel_usage.csv"
        with open(filename, "rt") as usage_file:

            # Use the standard csv module to get
            # a reader object for the CSV file.
            reader = csv.reader(usage_file)

            # The first line of the CSV file contains
            # headings and not fuel usage data, so this
            # statement skips the first line of the file.
            next(reader)

            # Print headers for the three columns.
            print("Date,Start,End,Gallons,Miles/Gallon")

            # Process each row in the CSV file.
            for row_list in reader:

                # From the current row of the CSV file, get
                # the date, the starting and ending odometer
                # readings, and the number of gallons used.
                date = row_list[DATE_INDEX]
                start_miles = float(row_list[START_MILES_INDEX])
                end_miles = float(row_list[END_MILES_INDEX])
                gallons = float(row_list[GALLONS_INDEX])

                # Call the miles_per_gallon function.
                mpg = miles_per_gallon(
                        start_miles, end_miles, gallons)

                # Display the results for one row.
                mpg = round(mpg, 1)
                print(date, start_miles, end_miles,
                        gallons, mpg, sep=",")

    except FileNotFoundError as not_found_err:
        print(f"Error: cannot open {filename}"
                " because it doesn't exist.")

    except PermissionError as perm_err:
        print(f"Error: cannot read from {filename}"
                " because you don't have permission.")

    except ZeroDivisionError as zero_div_err:
        print(f"Error: {filename} contains a"
                " zero in the gallons column.")

```

```python

def miles_per_gallon(start_miles, end_miles, gallons):
    """Compute and return the average number of miles
    that a vehicle traveled per gallon of fuel.

    Parameters
        start_miles: starting odometer reading in miles.
        end_miles: ending odometer reading in miles.
        gallons: amount of fuel used in U.S. gallons.
    Return: miles per gallon
    """
    mpg = abs(end_miles - start_miles) / gallons
    return mpg


# Call main to start this program.
if __name__ == "__main__":
    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, there is a `try` block at line 29. 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.

```python
 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.
```

```

    Parameters
        prompt: A string to display to the user.
        lower_bound: The smallest number that the user may enter.
        upper_bound: The largest number that the user may enter.
    Return: The number entered by the user.
    """
    while True:
        try:
            text = input(prompt)
            number = float(text)
            if number < lower_bound:
                print(f"{number} is too small.")
                print("Please enter another number.")
            elif number > upper_bound:
                print(f"{number} is too large.")
                print("Please enter another number.")
            else:
                # If the computer gets to this line of code,
                # the user entered a valid number between
                # lower_bound and upper_bound, so exit the loop.
                break

        except ValueError as val_err:
            # The user entered at least one character that is
            # not part of a floating point number, so print a
            # message asking the user to enter a number.
            print(f"{text} is not a number.")
            print("Please enter a number.")

    return number


def basal_metabolic_rate(gender, weight, height, age):
    """Calculate and return a person's basal metabolic rate
    in calories per day. weight must be in kilograms, height
    must be in centimeters, and age must be in years.
    """
    if gender.upper() == "F":
        bmr = 447.593 + 9.247 * weight \
                + 3.098 * height - 4.330 * age
    else:
        bmr = 88.362 + 13.397 * weight \
                + 4.799 * height - 5.677 * age
    return bmr


# Call main to start this program.
if __name__ == "__main__":
    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. Write 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`, `FileNotFoundError`, and `PermissionError`. When writing code that writes to or reads from a file, a programmer usally writes `try` and `except` blocks to handle `FileNotFoundError` 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.

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.

```python
 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.

```python
 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
12      fahr_temps = [72, 65, 71, 75, 82, 87, 68]
13
14      # Print the Fahrenheit temperatures.
15      print(f"Fahrenheit: {fahr_temps}")
16
17      # Convert each Fahrenheit temperature to Celsius and store
18      # the Celsius temperatures in a list named cels_temps.
19      cels_temps = list(map(cels_from_fahr, fahr_temps))
20
21      # Print the Celsius temperatures.
22      print(f"Celsius: {cels_temps}")
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.

```python
 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.

```python
    # 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 the lesson 9 checkpoint. The example program works by doing the following:

1. Calling the `read_list` function at line 6 to read all the provinces from a text file into a list. (The `read_list` function is in the preparation content for lesson 9.)
2. Calling the `map` function at line 22 to convert all elements that are "AB" to "Alberta."
3. Calling the `filter` function at line 34 to remove all elements that are not "Alberta."
4. Calling the `len` function at line 42 to count the number of elements that remain in the filtered list.

```
 1  # Example 5
 2
 3  def main():
 4      # Read a file that contains a list
 5      # of Canadian province names.
 6      provinces_list = read_list("provinces.txt")
 7
 8      # As a debugging aid, print the entire list.
 9      print("Original list of provinces:")
10      print(provinces_list)
11      print()
12
13      # Define a nested function that converts AB to Alberta.
14      def alberta_from_ab(province_name):
15          if province_name == "AB":
16              province_name = "Alberta"
17          return province_name
18
```

```python
    # Replace all occurrences of "AB" with "Alberta" by
    # calling the map function and passing the ablerta_from_ab
    # function and provinces_list into the map function.
    new_list = list(map(alberta_from_ab, provinces_list))
    print("List of provinces after AB was changed to Alberta:")
    print(new_list)
    print()

    # Define a lambda function that returns True if a
    # province's name is Alberta and returns False otherwise.
    is_alberta = lambda name: name == "Alberta"

    # Filter the new list to only those provinces that
    # are "Alberta" by calling the filter function and
    # passing the is_alberta function and new_list.
    filtered_list = list(filter(is_alberta, new_list))
    print("List filtered to Alberta only:")
    print(filtered_list)
    print()

    # Because all the elements in filtered_list are
    # "Alberta", we can count how many elements are
    # "Alberta" by simply calling the len function.
    count = len(filtered_list)

    print(f"Alberta occurs {count} times in the modified list.")


# Call main to start this program.
if __name__ == "__main__":
    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', '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']

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', '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',
'Alberta', 'Nova Scotia', 'Newfoundland and Labrador', 'Yukon',
'Nunavut', 'Northwest Territories', 'Nunavut', 'Yukon',
'British Columbia', 'Ontario', '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']

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.

```python
# Example 6

def main():
    # Create a list that contains country names
    # and print the list.
    countries = [
        "Canada", "France", "Ghana", "Brazil", "Japan"
    ]
    print(countries)

    # Sort the list. Then print the sorted list.
    sorted_list = sorted(countries)
    print(sorted_list)


# Call main to start this program.
```

```
if __name__ == "__main__":
    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)
```

```
    print()

    # Define a lambda function that will be used as the
    # key function by the sorted function. The lambda
    # function extracts the population data from a
    # country so that the population will be used for
    # sorting the list of countries.
    POPULATION_INDEX = 2
    popul_func = lambda country: country[POPULATION_INDEX]

    # Sort the list of countries by the population.
    sorted_list = sorted(countries, key=popul_func)

    # Print the sorted list.
    print("List of countries sorted by population")
    for country in sorted_list:
        print(country)


# Call main to start this program.
if __name__ == "__main__":
    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.

```python
# Example 8

def main():
    # Create a list that contains data about young students.
    students = [
        # [given_name, surname, reading_level]
        ["Robert", "Smith", 6.7],
        ["Annie", "Smith", 6.2],
        ["Robert", "Lopez", 7.1],
        ["Sean", "Li", 5.6],
        ["Sofia", "Lopez", 5.3],
        ["Lily", "Harris", 6.7],
        ["Alex", "Harris", 5.8]
    ]

    GIVEN_INDEX = 0
    SURNAME_INDEX = 1

    # Define a lambda function that combines
    # a student's surname and given name.
    combine_names = lambda student_list: \
        f"{student_list[SURNAME_INDEX]}, {student_list[GIVEN_INDEX]}

    # Sort the list by the combined key of surname, given_name.
    sorted_list = sorted(students, key=combine_names)

    # Print the list.
    for student in sorted_list:
        print(student)


# Call main to start this program.
if __name__ == "__main__":
    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 preparation 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.

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.

```python
# 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;
```

```
   AVG(numbers)
-----------------
87.57142857142857
```

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.

```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 below contains a program that is similar to example 2 in the preparation 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():
```

```python
    # 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)

    # Get the index where velvet is stored in the fabrics list.
    i = fabrics.index("velvet")

    # Replace velvet with taffeta.
    fabrics[i] = "taffeta"
    print(fabrics)

    # Remove the last element from the fabrics list.
    fabrics.pop()

    # Remove denim from the fabrics list.
    fabrics.remove("denim")
    print(fabrics)


# Call main to start this program.
if __name__ == "__main__":
    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 *d*.
d.copy()	Returns a copy of the dictionary *d*.

Method	Description
d.get(*key*)	Returns the value of the specified `key`. 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 *d*.
d.keys()	Returns a list that contains the keys that are in the dictionary *d*.
d.pop(*key*)	Removes the element with the specified *key* from the dictionary *d*.
d.update(*other*)	Updates the dictionary *d* with the key value pairs that are in the *other* dictionary.
d.values()	Returns a list that contains the values that are in the dictionary *d*.

The following example code, which is similar to example 1 from the preparation 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
```

```python
    # in the dictionary. Of course, the code in
    # the body of a loop can do much more with
    # each key value pair than simply print it.
    for key, value in students.items():
        print(key, value)


# Call main to start this program.
if __name__ == "__main__":
    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.

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.