1. INTRODUCTION TO PYTHON

1.1 Programming general

* Program consists of instructions executing one at a time
  + Basic instruction types are INPUT, PROCESS, OUTPUT

1.2 Programming using python

* The python interpreter is a computer program that executes code written in the Python programming language. An interactive interpreter is a program that allows the user to execute one line of code at a time.
* Code is a common word for the textual representation of a program.
* Line is a row of text

1.3 Basic input and output

* The value of a variable can be printed out via print(variable\_name) without needing quotations.
* The addition of end= ‘ ‘ allows the output to be printed on the same line with a space after halt
* Output can be moved to the next line using the newline character “\n”
  + Example - print(‘1\n2\n3’) print 1 on the first line, 2 on the second line and 3 on the 3rd line
* **Many useful programs allow a user to enter values such as typing a number, a name, etc.**
  + The input() function is used to read input from a user. The statement

best\_friend = input()

* Reading from input always results in a string type
* If a string contains only numbers line ‘123’, then **int()** function can be used to convert that string to the integer 123

1.4 Errors

| **Error type** | **Description** |
| --- | --- |
| SyntaxError | The program contains invalid code that cannot be understood. |
| IndentationError | The lines of the program are not properly indented. |
| ValueError | An invalid value is used, which can occur if giving letters to int(). |
| NameError | The program tries to use a variable that does not exist. |
| TypeError | An operation uses incorrect types, which can occur if adding an integer to a string. |

1. VARIABLES AND EXPRESSIONS

2.1 Variables and assignments

* In a program, a variable is a named intel, such as x or num\_people, that holds a value
* An assignment statement assigns a variable with a value such as x=5. That statement means x is assigned with 5, and x keeps that value during subsequent statements, until x is assigned again.
  + Assignment statement on the left side is variable. Right side is an expression

2.2 Identifiers

* Identifiers always start with a letter or an underscore. Never a number. Never a space

2.3 Experimenting with objects

* Use type() to print an objects type
* Use id() to print an object's identity - memory address where the object is stored.

2.4 Numeric types: Floating-point

* Floating point numbers are numbers like 98.6, 0.0001, -5555.555. Decimal is floating in the number. This data type is called **float**
* A floating point literal is written with the fractional part. - like - 1.0, 0.0, 99.0
* If a number is too big in a float, then it becomes an overflow error.

2.5 Arithmetic expressions

* Double \*\* means to the power or - ex: x\*\*y = x to the power of y

2.6 Python expressions

2.7 Division and modulo

* A division operator performs a division and returns a floating point number.
* The floor division operator // can be used to round down the result of a floating-point division to the closest smaller whole number value.

2.8 Module basics

* A module is a file containing Python code that can be used by other modules or scripts. A module is made available for use via the **import** statement Once a module is imported, any object defined in that module can be accessed using **dot notation** 
  + A variable speed\_of\_light defined in universe.py is accessed via universe.speed\_of\_light.

2.9 Math module

* Import **math**
  + Used for more complex math such as finding square roots and such.
    - EXAMPLE
      * Import math
      * Num = 49
      * Num\_sqrt = math.sqrt(num)
* A function is a list of statements that can be executed simply by referring to the function's name.

2.10 Random numbers

* The random() method returns a random floating point value whenever the function is called, in the range 0(inclusive) to 1(exclusive)
* randrange() method generated random integers within a specified range.

2.11 Representing text

1. Types

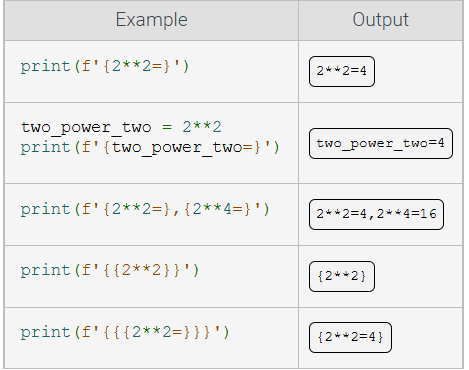
3.1 String basics

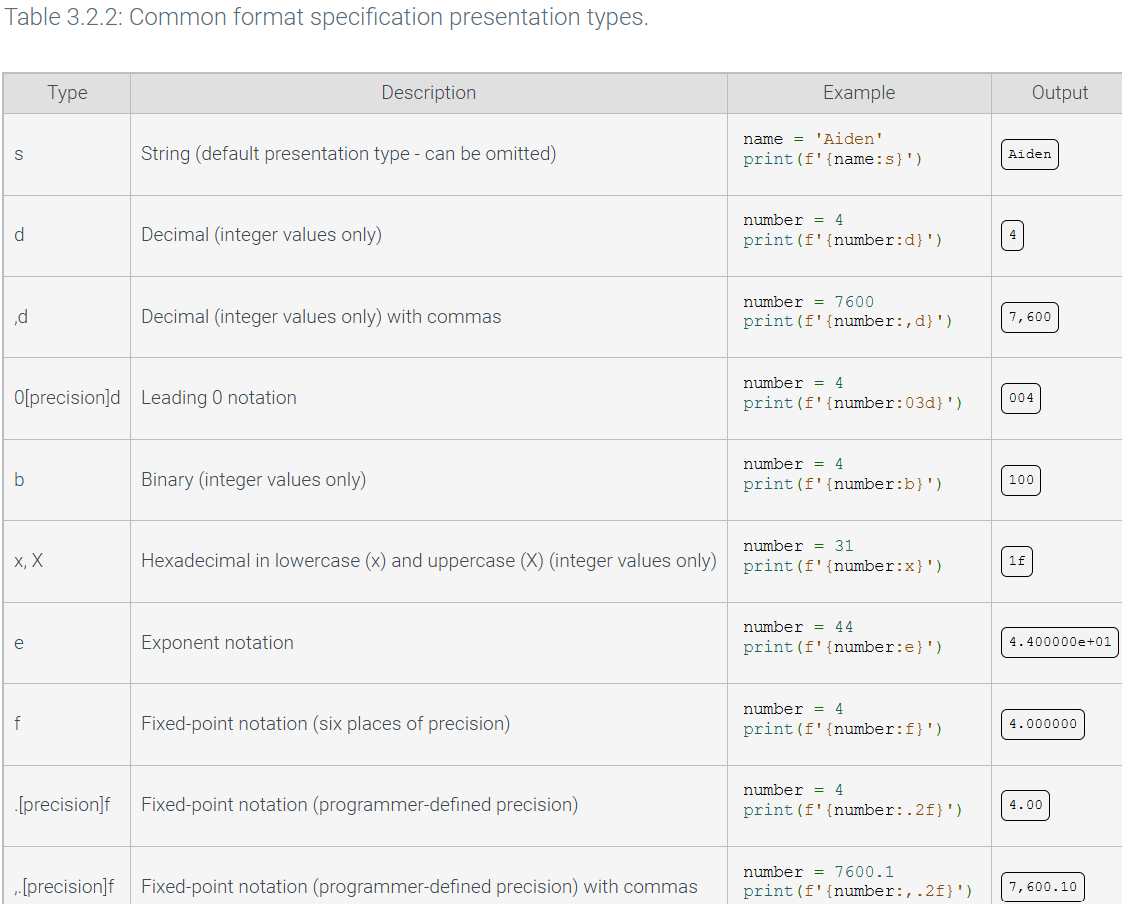
* Printing the length of a string - print(len((variable))
* Finding characters within a string. Ex: if you want to find the first character of string my\_country,
  + Answer: my\_country[0]
* Example: Assign my\_var with the last character in my\_str. Use a negative index
  + Answer: my\_var = my\_str[-1] -------- -2 would be the character to the left of -1
* Modification of string variables is not allowed. Reassign the string to the same variable with the full corrections
* The + operator concatenates two strings together
* Example problem:
  + First, read strings fav\_drink1 and fav\_drink2 from input and assume fav\_drink1 is always longer than fav\_drink2. Then, output the following, all separated by spaces:
    - The value of fav\_drink2
    - 'Is'
    - The length of fav\_drink1 minus the length of fav\_drink2
    - 'character(s) shorter than'
    - The value of fav\_drink1

Answer:

* + 

3.2 String formatting

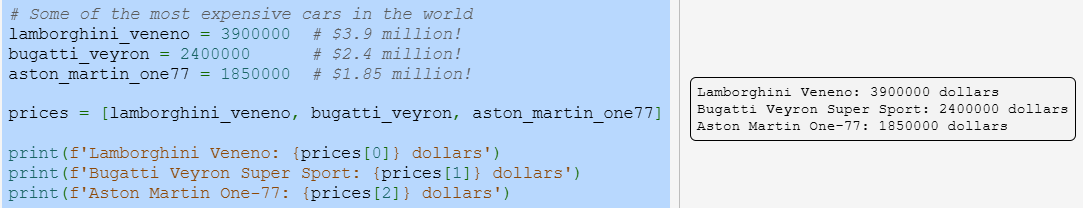
* A formatted string literal or f string allows a programmer to create a string with placeholder expressions that are evaluated as the program executes.
  + An f-string starts with an f character before the starting quote and uses curly braces { } to denote the placeholder expression.
    - Example: num\_items = 3, cost\_taco = 1.25 —-------- (((I need 2 items please)))
      * Answer: print(f’I need {num\_items} items please’)
    - Example: ((( 3 tacos cost 3.75)))
      * Answer: print(f’ {num\_items} tacos cost {cost\_taco \* num\_items} ’)
  + An equals sign is provided after the expression in a replacement field to print both the expression and its result.
    - Example: (output = f’{2\*2}’ — output is 4… ) (output = f’ {2\*2=} ‘ — output is 2\*2=4)
    - 



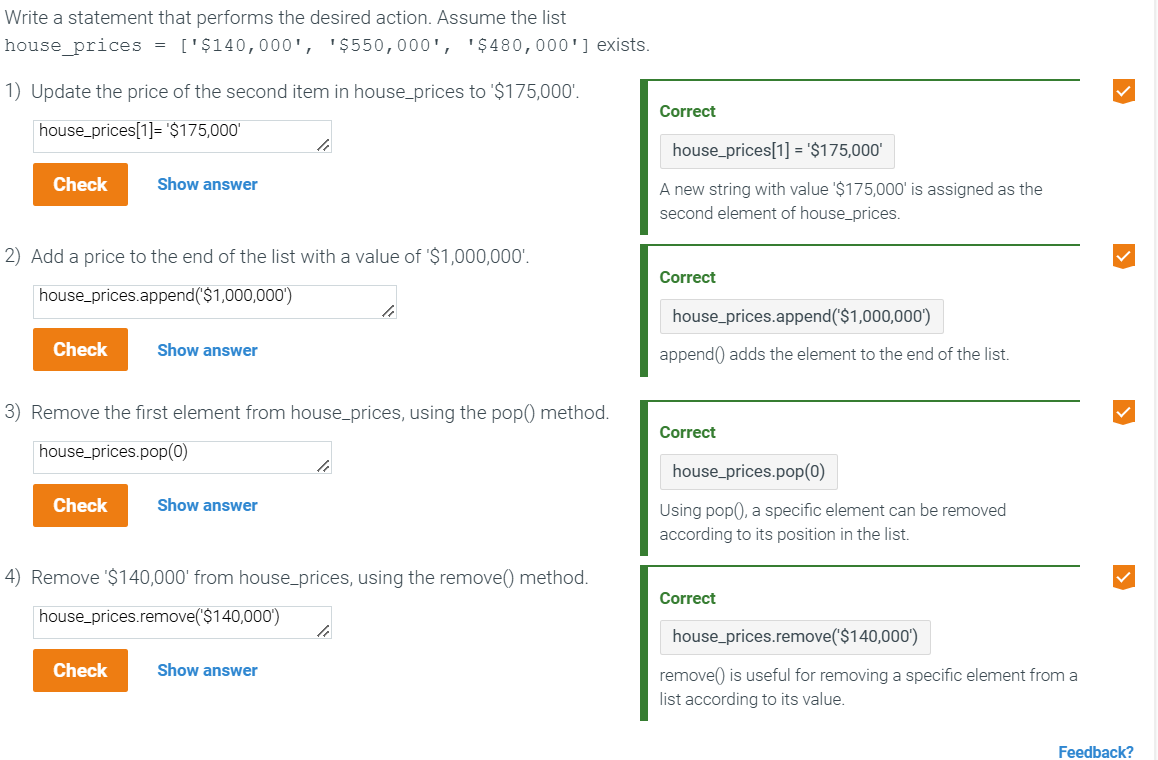
* Example:
  + Number = 10
  + Print (f’ {number:d}, {number:04d}, {number: .3f)’)
* Example: Mariana’s loan rate is 4.0%
  + print(f’{borrowers\_name} loan rate is {loan\_rate \* 100:1f} %.’)
* Example: input\_num is read from input as an integer. Using format specifications, output the following:
  + input\_num as a binary value
    - Answer: print(f’{input\_num:b}’)
  + Input\_num in fixed-point notation with four places of precision
    - Answer: print(f’{input\_num:.4f}’)

3.3 List basics

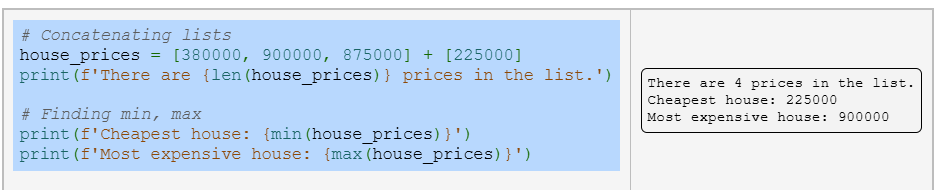
* A container is a construct used to group related values together and contains references to other objects instead of data.
  + A list is a contained created by surrounding a sequence of variables or literals with brackets [ ]
    - Ex: my\_list = [10, ‘abc’] —------------------ This list contains **10 and ‘abc**’ as the data
  + Lists are useful for reducing the number of variables in a program. Instead of having a separate variable for the name of every student in the class, a single list can store an entire collection of related variables.
    - A list index must be an integer. The index cannot be a floating-point type, even if the value is a whole number like 0.0 or 1.0.

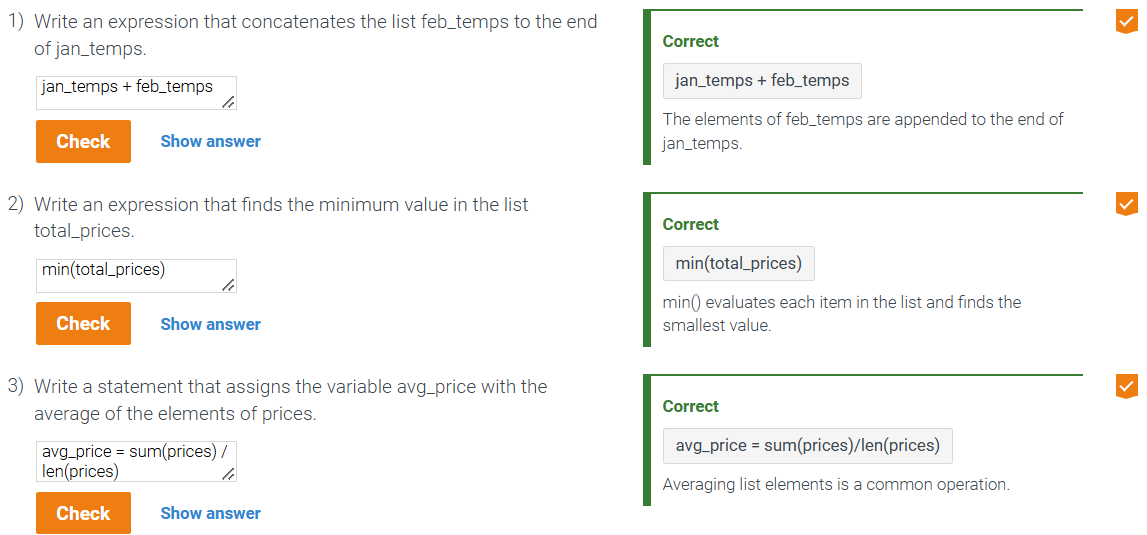


* Example: Lists are mutable, meaning that a programmer can change a list’s contents.
  + My\_nums = [5, 12, 20]
  + # update a list element
  + My\_nums [1] = -28
  + Print (my\_nums)
* A **Method** instructs an object to perform some action, and is executed by specifying the method name following a “.” symbol and an object.
* The **append() list method is used to add new elements to a list**
* Elements can be removed using the **pop()** or **remove()** methods.
  + Adding elements to a list:
    - list.append(value): adds value to the end of list — Ex: my\_list.append(‘abc’)
  + Removing elements from a list:
    - list.pop(i): Removes the element at index i from list. Ex:my\_list.pop(1)
    - list.remove(v): Removes the first element whose value is v. Ex: my\_list.remove(‘abc’)



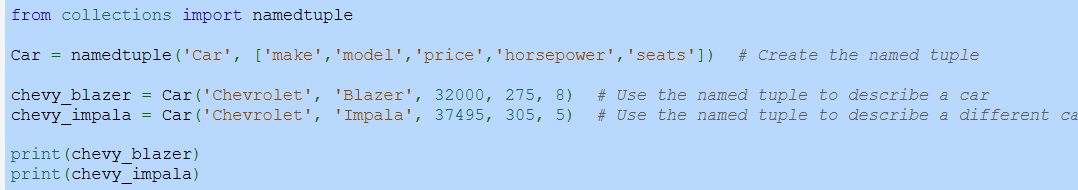


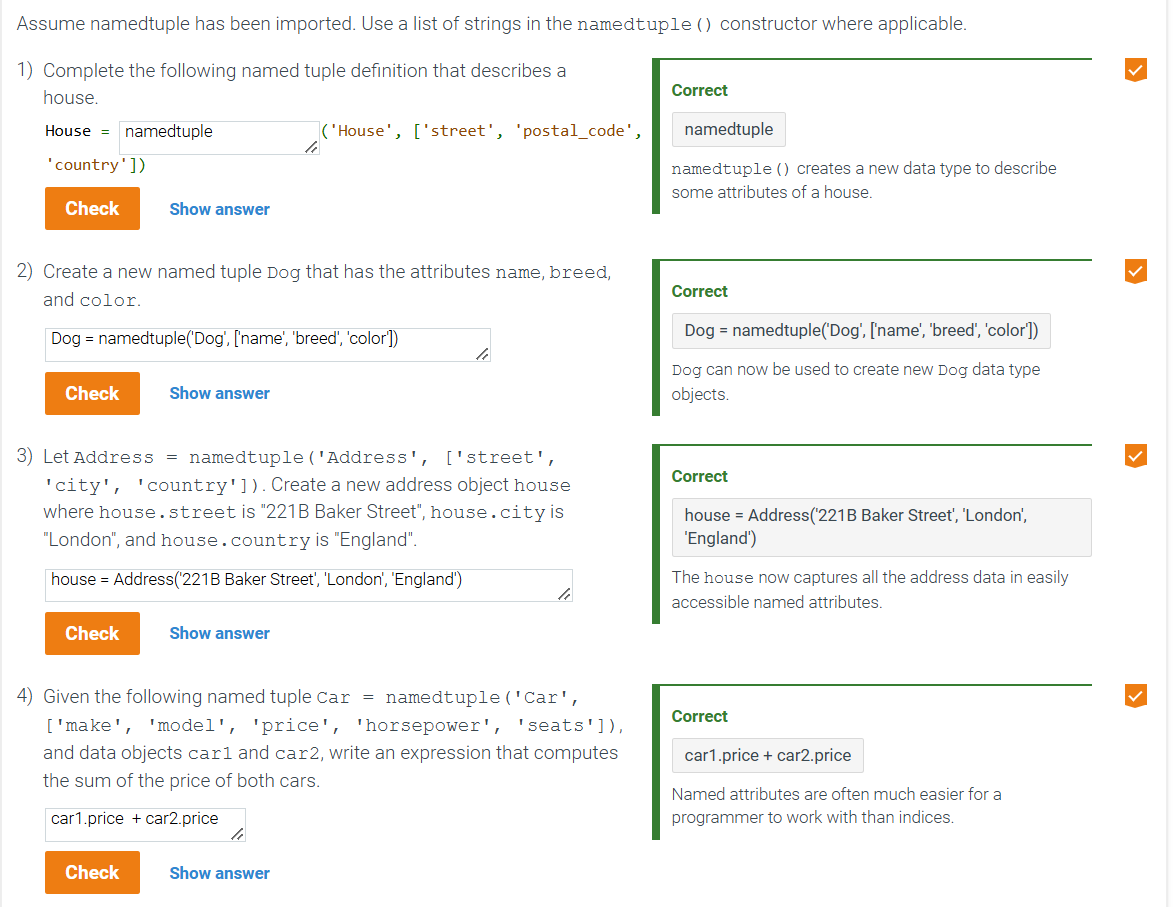




3.4 Tuple basics

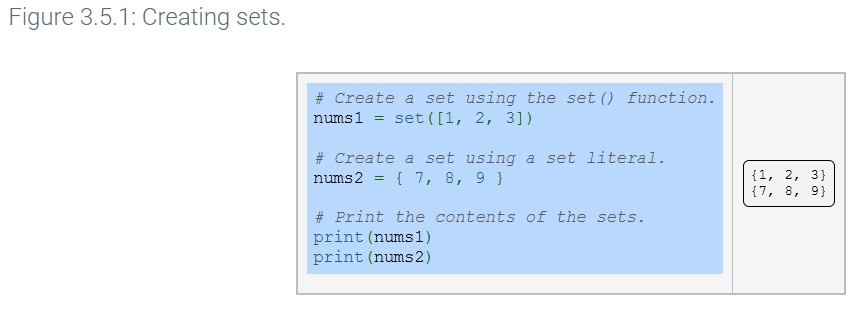
* A **tuple** stores a collection of data, like a list, but is immutable - once created, the tuple’s elements cannot be changed.
  + Printing a tuple always displays surrounding parentheses
  + Tuple is a list that cannot change
  + Example:
    - Create a new variable point that is a tuple containing the strings 'X string' and 'Y string'.
      * Answer: point = ('X string', 'Y string')
* Examples of names tuples
  + Car.price, car.horsepower, car.make, car.model …
* FOR NAMED TUPLES TO WORK, YOU NEED TO IMPORT IT
  + Example: from collections import namedtuple



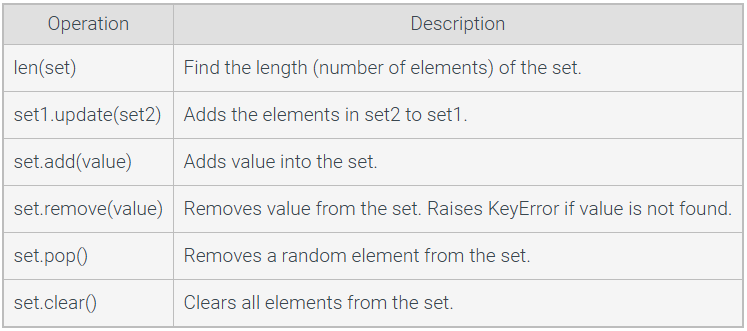


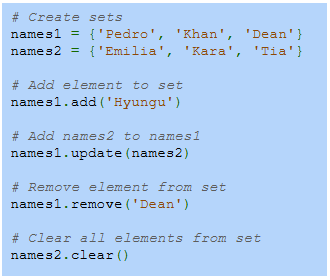
3.5 Set basics

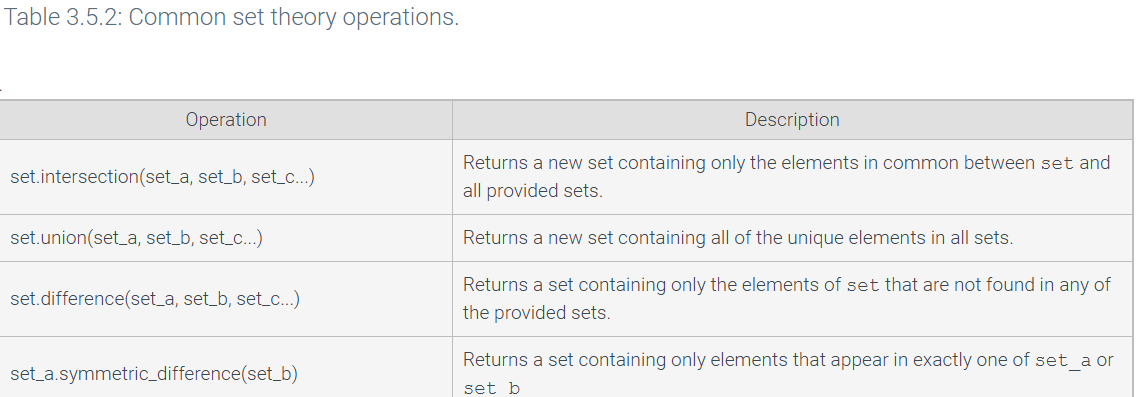
* A set is an unordered collection of unique elements. A set has the following properties:
  + Elements are unordered: Elements in the set do not have a position or index
  + Elements are unique: No elements in the set share the same value
* A set can be created using a **set()** function, which only accepts a sequence-type irritable object(list,tuples, strings, etc)
* A **set literal** can be written using curly braces { } with commas separating set elements.

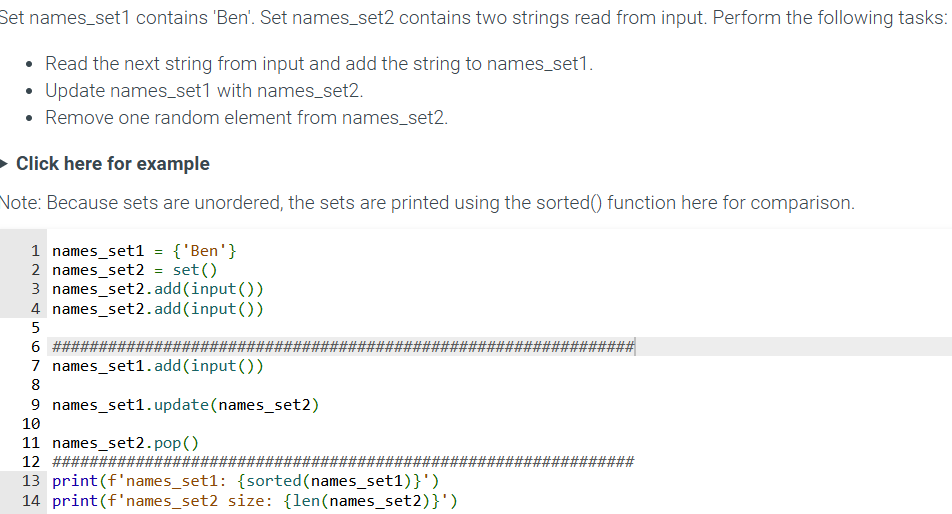


* A set is often used to reduce a list of items that potentially contains duplicates into a collection of unique values. Simply passing a list into set () will cause any duplicates to be omitted in the created set.



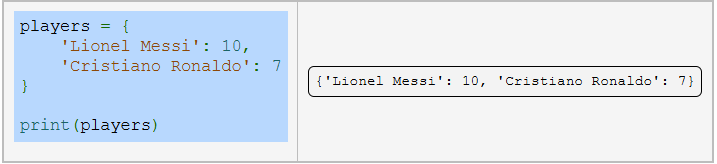






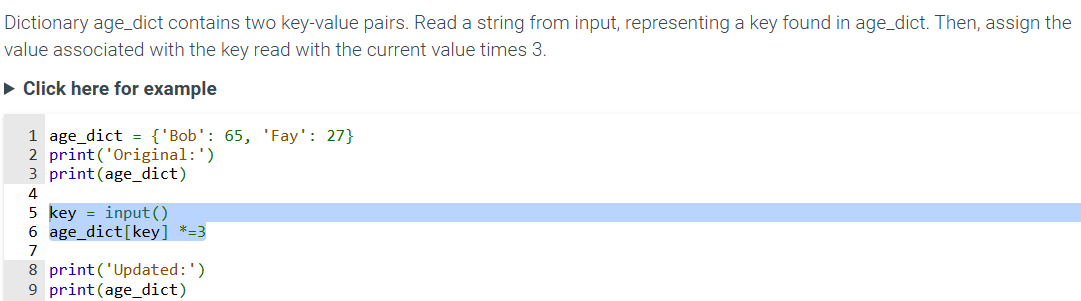
3.6 Dictionary basics

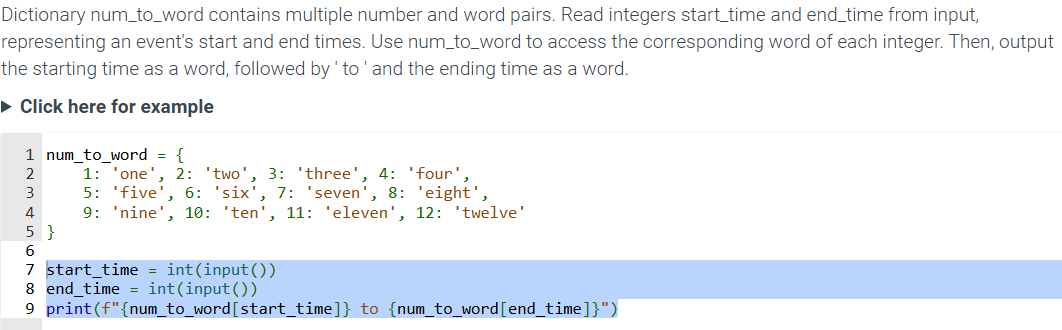
* A dictionary is represented by the **dict** object type.
* A **dict** object is created using **curly braces** { } to surround the **key:value pairs** that comprise the dictionary contents.
  + Ex: players = {‘Lionel Messi’ : 10, ‘Cristiano Ronaldo’ : 7} creates a dictionary called players with two keys: ‘Lionel Messi’ and ‘Cristiano Ronaldo’ , associated with values 10 and 7 (their jersey numbers).
* An empty dictionary is created with the expression players = { }
* **Dictionaries are typically used in place of lists when an associative relationship exists. If a program contains a collection of anonymous student test scores, those scores should be stored in a list. However, if each score is associated with a student name, a dictionary could be used to associate students' names with their scores. Other examples of associative relationships include last names and addresses, car models and prices, or student Id number and university email address.**



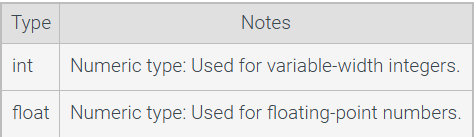
* A dictionary is mutable, so entries can be added, modified, and deleted as necessary by a programmer.
  + A new dictionary entry is added by using brackets to specify the key: prices [‘banana’] = 1.49
  + The **del** keyword is used to remove entries from a dictionary: del prices [‘papaya’]

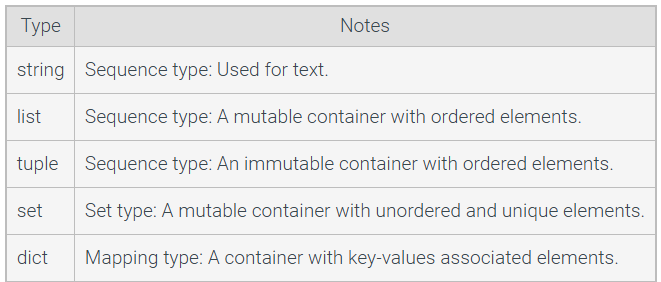






3.7 Common Data types

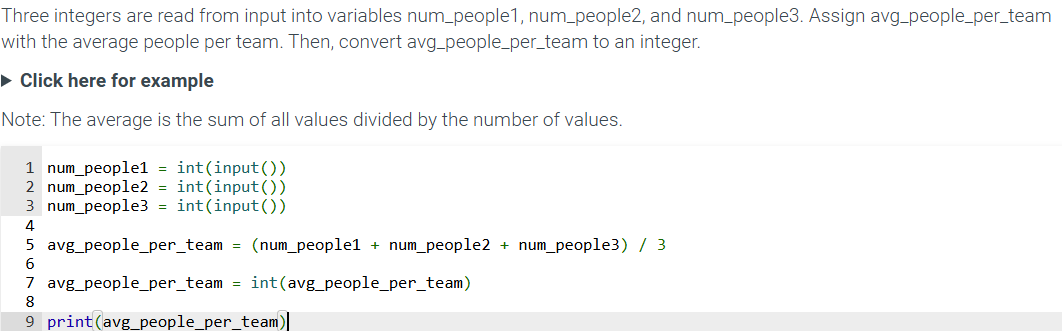




* A programmer might use a list when data has an order, such as lines of text on a page
* A programmer might use a tuple instead of a list if the constrained data should not change

3.9 Type conversions

* A **type** conversion is a conversion of one type to another. Such as an integer to a float.
* An **implicit conversion** is a type conversion automatically made by the interpreter, usually between numeric types.
  + Examples
    - 1 + 2 returns an integer type
    - 1 + 2.0 returns a float type
    - 1.0 + 2.0 returns a float type.
  + Integer-to-float conversion is straightforward: 25 becomes 25.0
  + Float-to-integer conversion just drops the fraction. 4.9 becomes 4



3.12 LABS - List basics

* Given the user inputs, complete a program that does the following tasks:
* Define a list, my\_list, containing the user inputs: my\_flower1, my\_flower2, and my\_flower3 in the same order.
* Define a list, your\_list, containing the user inputs, your\_flower1 and your\_flower2, in the same order.
* Define a list, our\_list, by concatenating my\_list and your\_list.
* Append the user input, their\_flower, to the end of our\_list.
* Replace my\_flower2 in our\_list with their\_flower.
* Remove the first occurrence of their\_flower from our\_list without using index().
* Remove the second element of our\_list.

my\_flower1 = input()

my\_flower2 = input()

my\_flower3 = input()

your\_flower1 = input()

your\_flower2 = input()

their\_flower = input()

**# 1. TODO: Define my\_list containing my\_flower1, my\_flower2, and my\_flower3**

**# in that order**

my\_list = [my\_flower1, my\_flower2, my\_flower3]

**# 2. TODO: Define your\_list containing your\_flower1 and your\_flower2**

**# in that order**

your\_list = [your\_flower1, your\_flower2]

**# 3. TODO: Define our\_list by concatenating my\_list and your\_list**

our\_list = my\_list + your\_list

print(our\_list)

**# 4. TODO: Append their\_flower to the end of our\_list**

our\_list.append(their\_flower)

print(our\_list)

**# 5. TODO: Replace my\_flower2 in our\_list with their\_flower**

our\_list[our\_list.index(my\_flower2)] = their\_flower

print(our\_list)

**# 6. TODO: Remove the first occurrence of their\_flower from**

**# our\_list without using index()**

our\_list.remove(their\_flower)

print(our\_list)

**# 7. TODO: Remove the second element of our\_list**

our\_list.pop(1) # you can also use ------------------del our\_list[1]

print(our\_list)

3.13 LAB: SET BASICS

Given the user inputs, complete a program that does the following tasks:

* Define a set, fruits, containing the user inputs: my\_fruit1, my\_fruit2, and my\_fruit3.
* Add the user inputs, your\_fruit1 and your\_fruit2, to fruits.
* Add the user input, their\_fruit, to fruits.
* Add your\_fruit1 to fruits.
* Remove my\_fruit1 from fruits.

my\_fruit1 = input()

my\_fruit2 = input()

my\_fruit3 = input()

your\_fruit1 = input()

your\_fruit2 = input()

their\_fruit = input()

**# 1. TODO: Define a set, fruits, containing my\_fruit1, my\_fruit2, and my\_fruit3**

fruits = {my\_fruit1, my\_fruit2, my\_fruit3}

print(sorted(fruits))

**# 2. TODO: Add your\_fruit1 and your\_fruit2 to fruits**

fruits.add(your\_fruit1)

fruits.add(your\_fruit2)

print(sorted(fruits))

**# 3. TODO: Add their\_fruit to fruits**

fruits.add(their\_fruit)

print(sorted(fruits))

**# 4. TODO: Add your\_fruit1 to fruits**

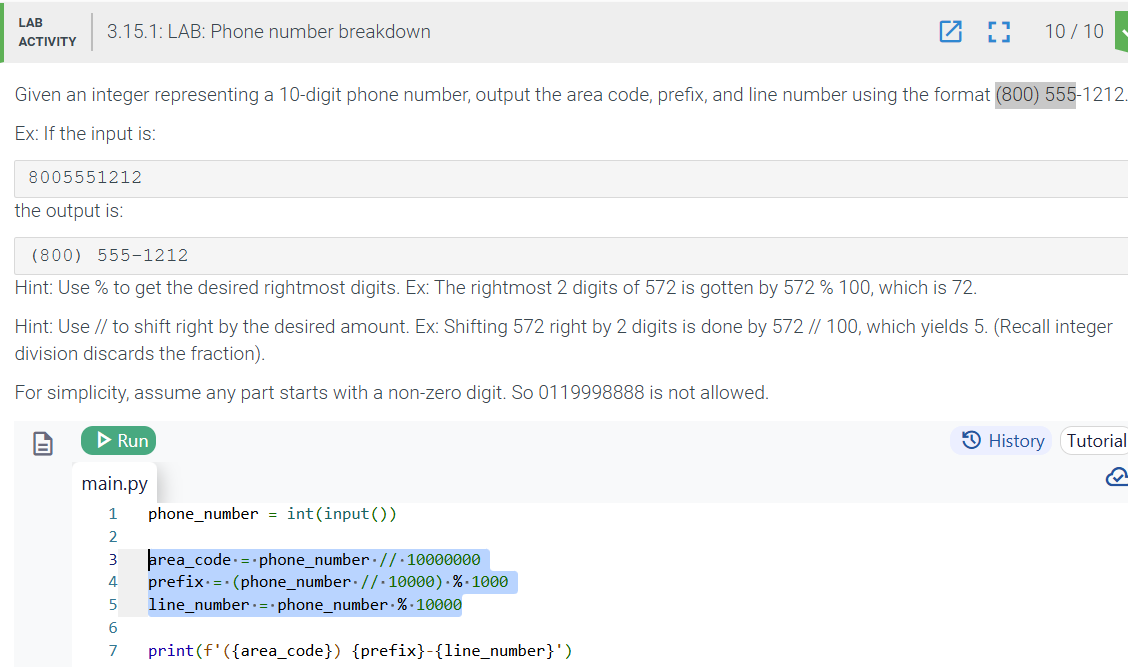
fruits.add(your\_fruit1)

print(sorted(fruits))

**# 5. TODO: Remove my\_fruit1 from fruits**

fruits.remove(my\_fruit1)

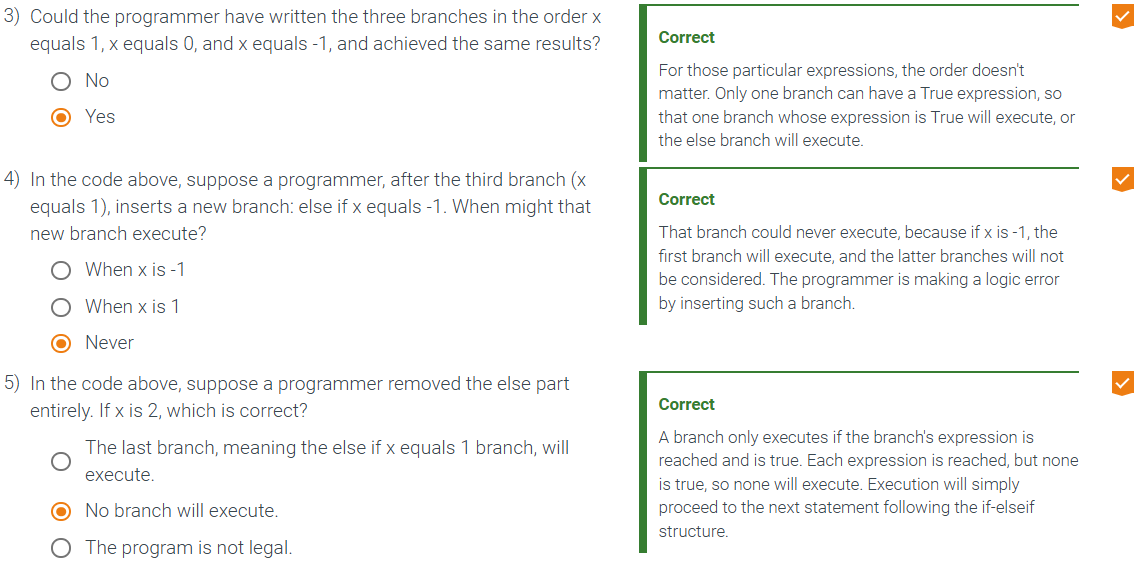
print(sorted(fruits))



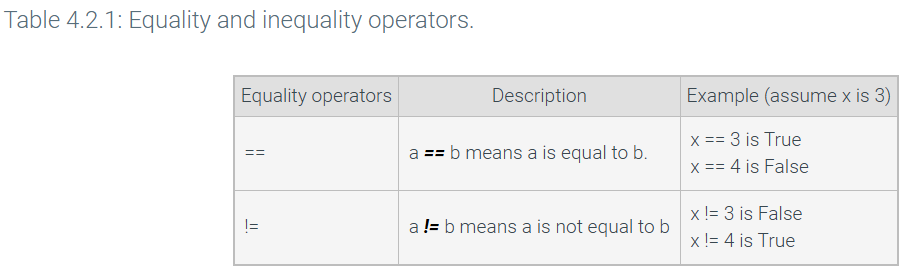
1. Branching

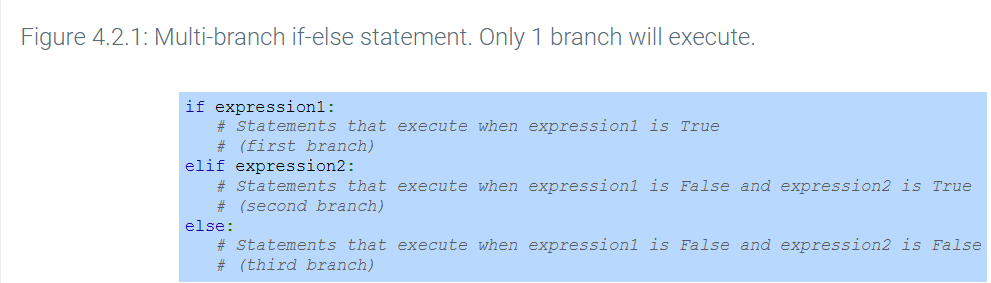
4.1 If else branches (general)

* In a program, a **branch** is a sequence of statements only executed under a certain condition.
  + Ex: Hotel may discount a price only for people over age 65
* An **if-else** branch has two branches.
  + First branch is executed if an expression is true, else, the other branch is executed.
* Only one branch can have a True expression, so that one branch whose expression is True will execute, or the else branch will execute.



4.2 Detecting equal values with branches

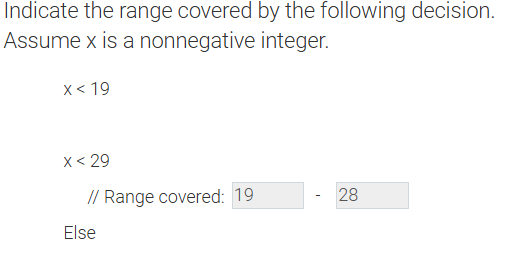
* An **if** statement executes a group of statements if an expression is true. The statements in a branch must be indented, typically four spaces.
  + The statements in a branch must be indented, typically four spaces.
* 
* An **if-else** statement executes one group of statements when an expression is true, and another group of statements when the expression is false.
* An **if-else** statement can be extended to have three or more branches.
  + Additional branches use the **elif** keyword, which means “else if”
  + Each branch expression is checked in sequence.



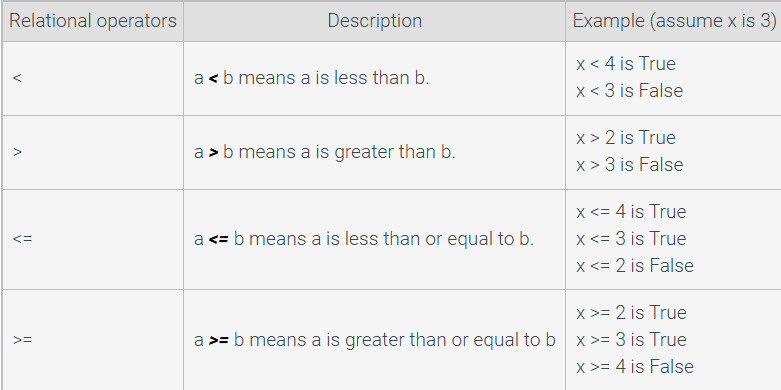


4.3 Detecting ranges with branches (general)

* The sequential nature of multi-branch if-else statements is useful to detect ranges of numbers.

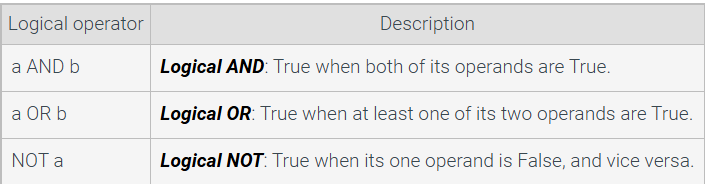


4.4 Detecting ranges with branches

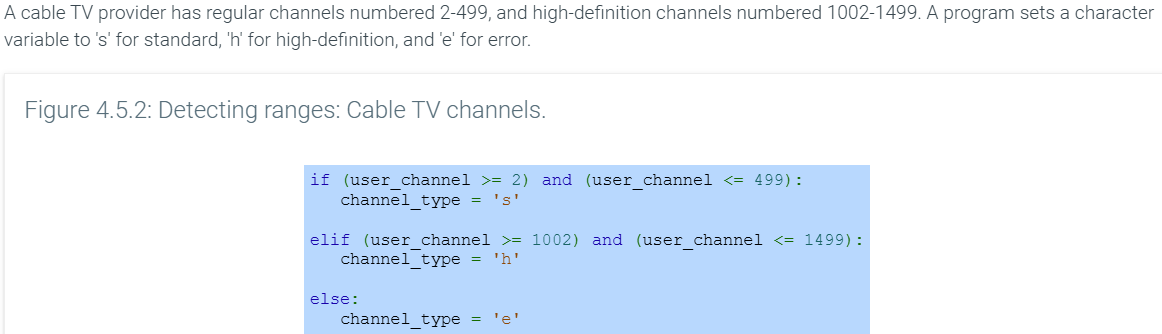
* relational operator checks how operand’s value relates to another, such as being greater than.
  + Some operators such as >= involve two characters.
  + 
* Programmers commonly use the sequential nature of the multi-branch if-else arrangement to detect ranges of numbers.
* Python supports **operator chaining** For example, a < b < c determines whether b is greater than a but less than c
  + Chaining performs comparisons left to right, evaluating a < b first. If the result is True, then b < c is evaluated next.
    - If the result of the first comparison a < b is false, then there is no need to continue evaluating the rest of the expression.

4.5 Detecting ranges using logical operators

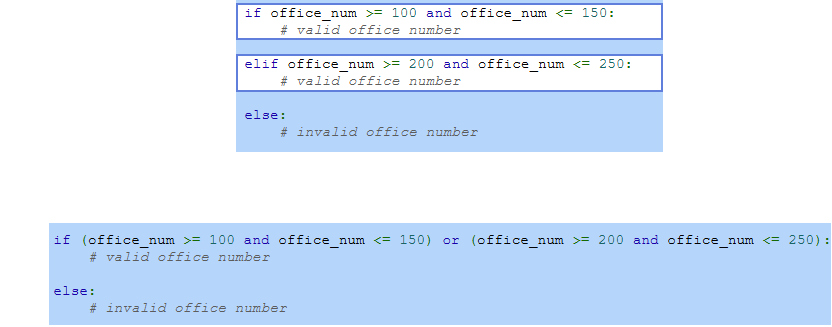
* A **logical operator** treats operands as being True or False and evaluates to True or False.
  + Logical operators include AND, OR, NOT
    - AND evaluates to true only if BOTH operands are True
    - OR evaluates to true if ANY operand is True
    - NOT evaluates to the opposite of the operand
    - Each operand is an expression.
      * If x = 7, y = 9, then (x>0) AND (y<10) is True AND true so the full expression evaluate to True



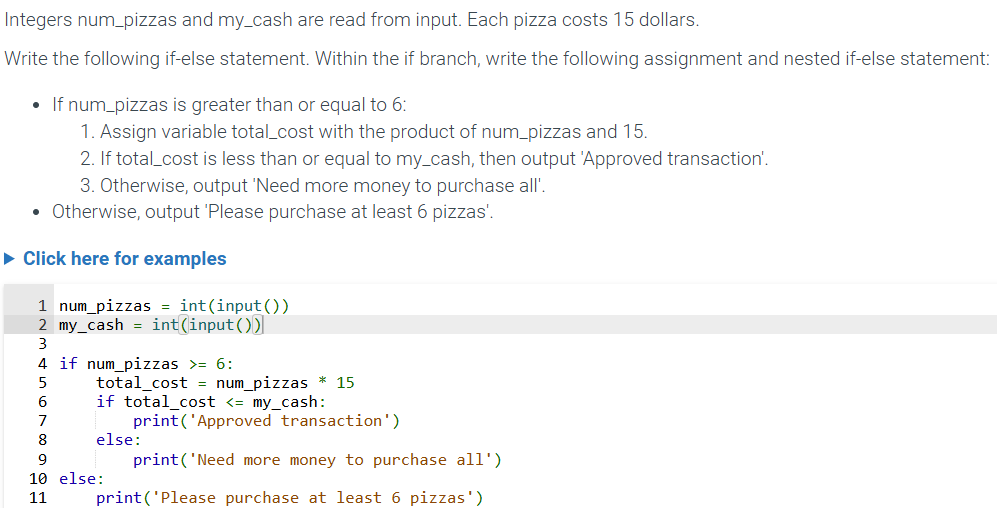
* A **boolean** refers to a value that is either True or False. Note that True and False are keywords in Python and must be capitalized.
  + A programmer can assign a boolean value by specifying True or False or by evaluating an expression that yields a Boolean.



4.6 Detecting ranges with gaps



4.7 Detecting multiple features with branches

* A branch’s statements can include any valid statements, including another if-else statement, which are known as **nested if-else** statements.
* 

4.8 Comparing data types and common errors

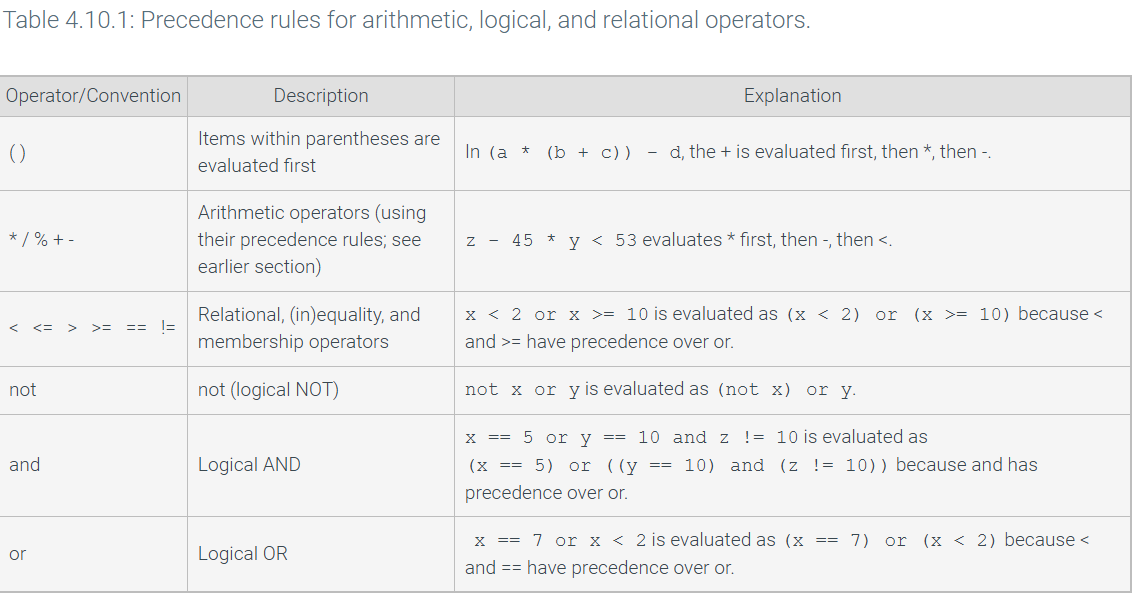
* Floating-point types should not be compared using the equality operators, due to the imprecise representation of floating-point numbers.

4.9 Membership and identity operators

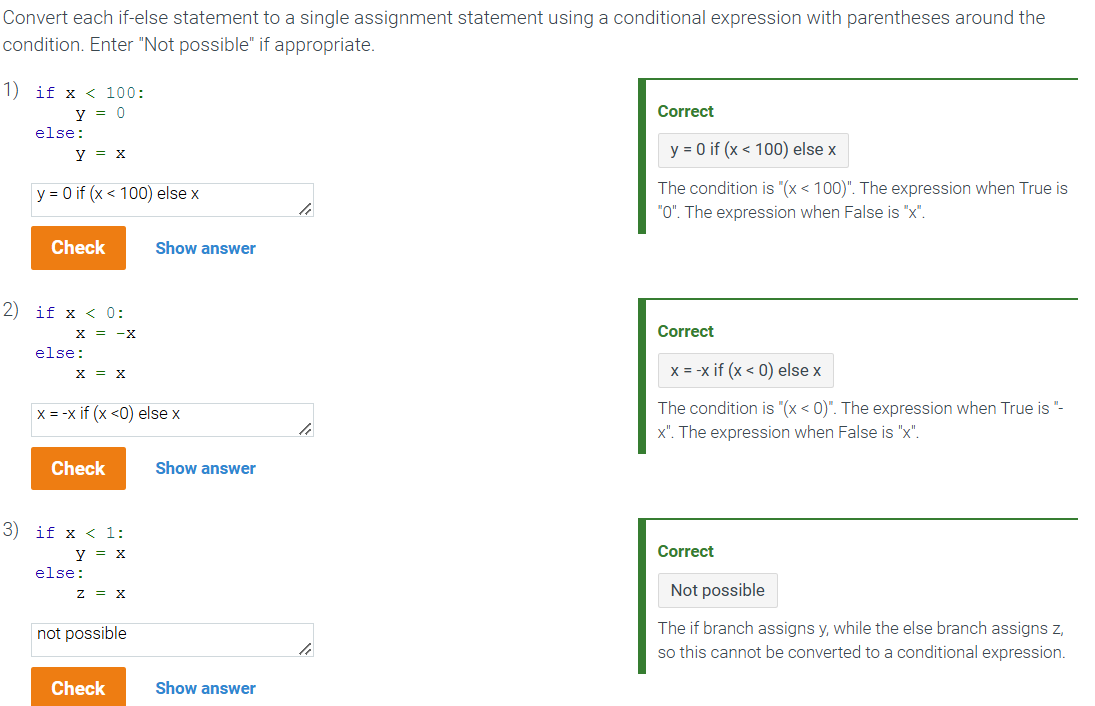
* One programming task involves determining whether a specific value can be found within a container, such as a list or dictionary
* The **in**  and **not in** operators are known as **membership operators** yield True or False if the left operand matches an element in the right operand, which is always a container.



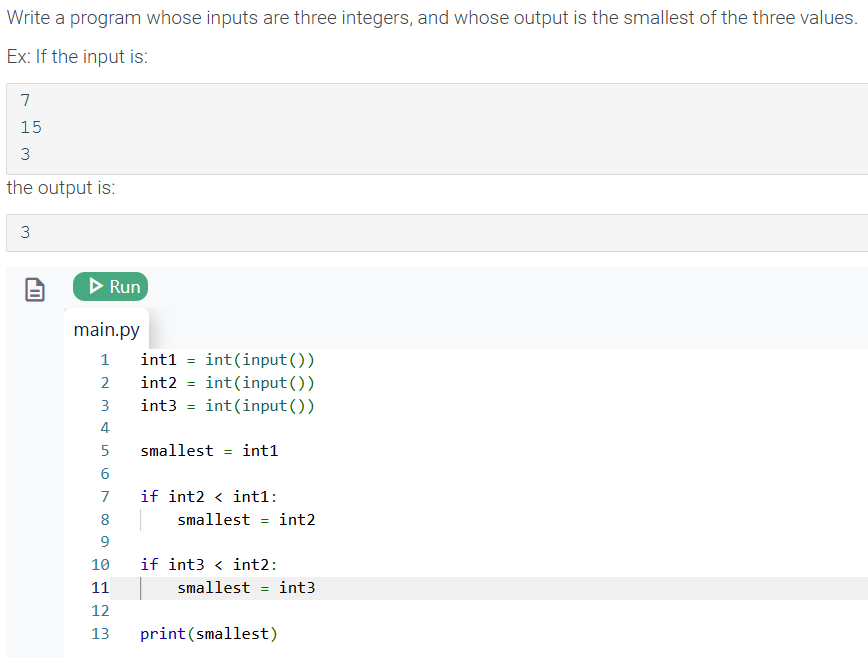
4.10 Order of evaluation

* The order in which operators are evaluated in an expression is known as precedence rules.
* 

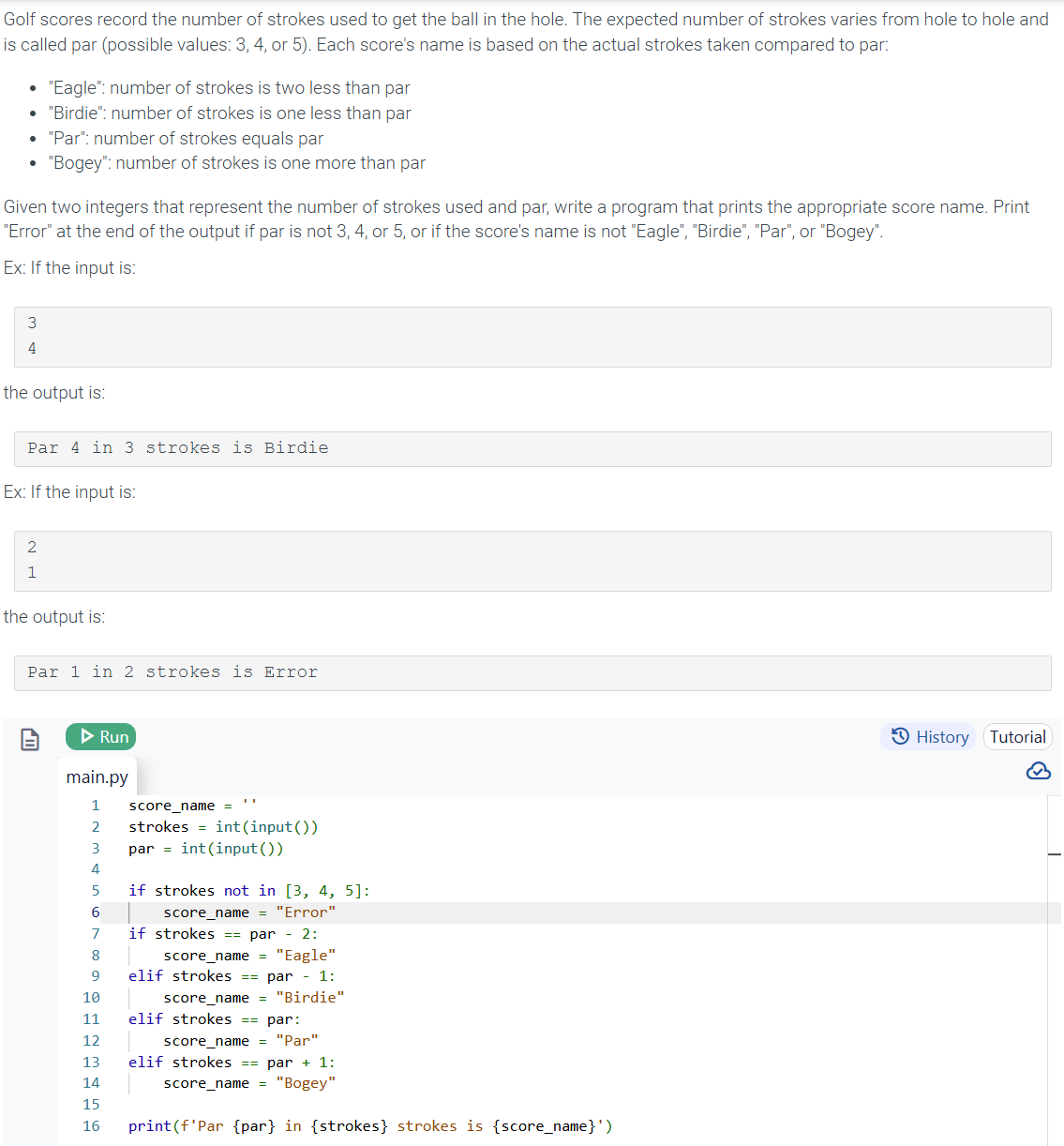
4.12 - Conditional expressions



4.14 LAB - SMALLEST NUMBER



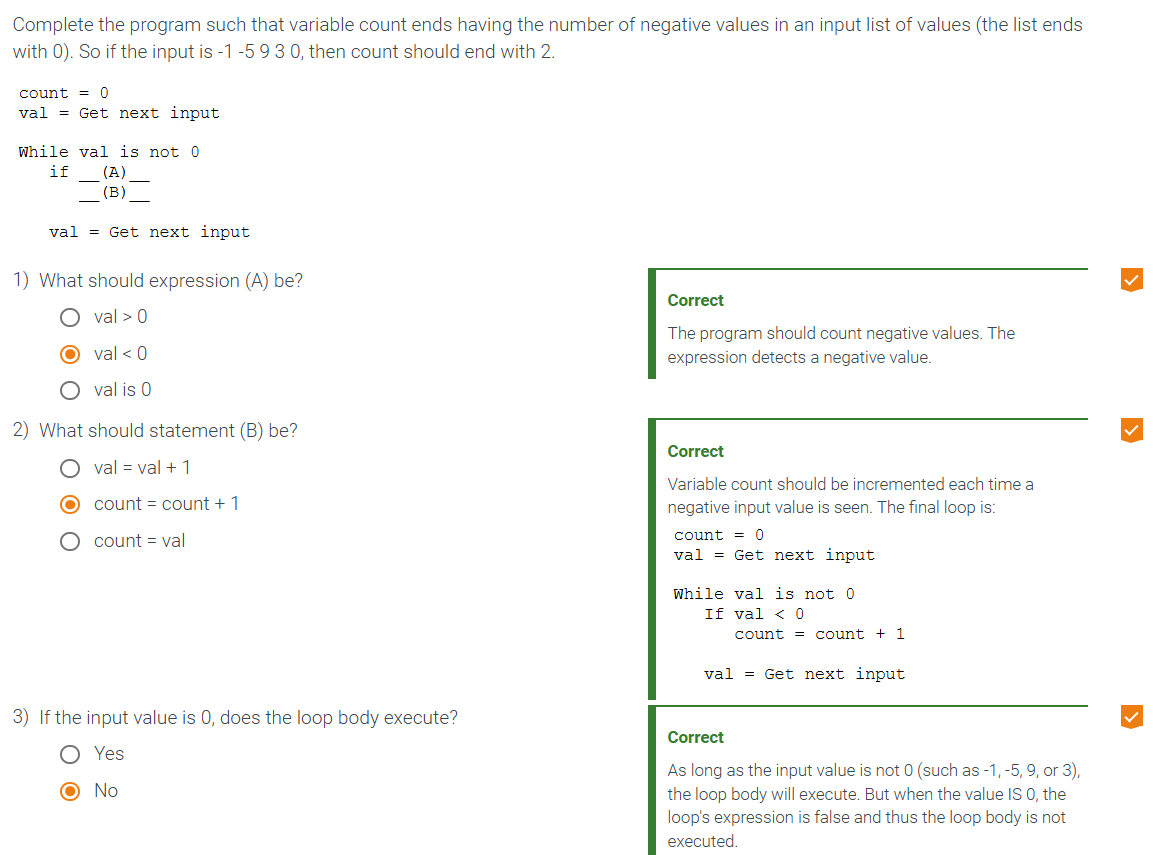
4.19 - LAB - Golf Scores

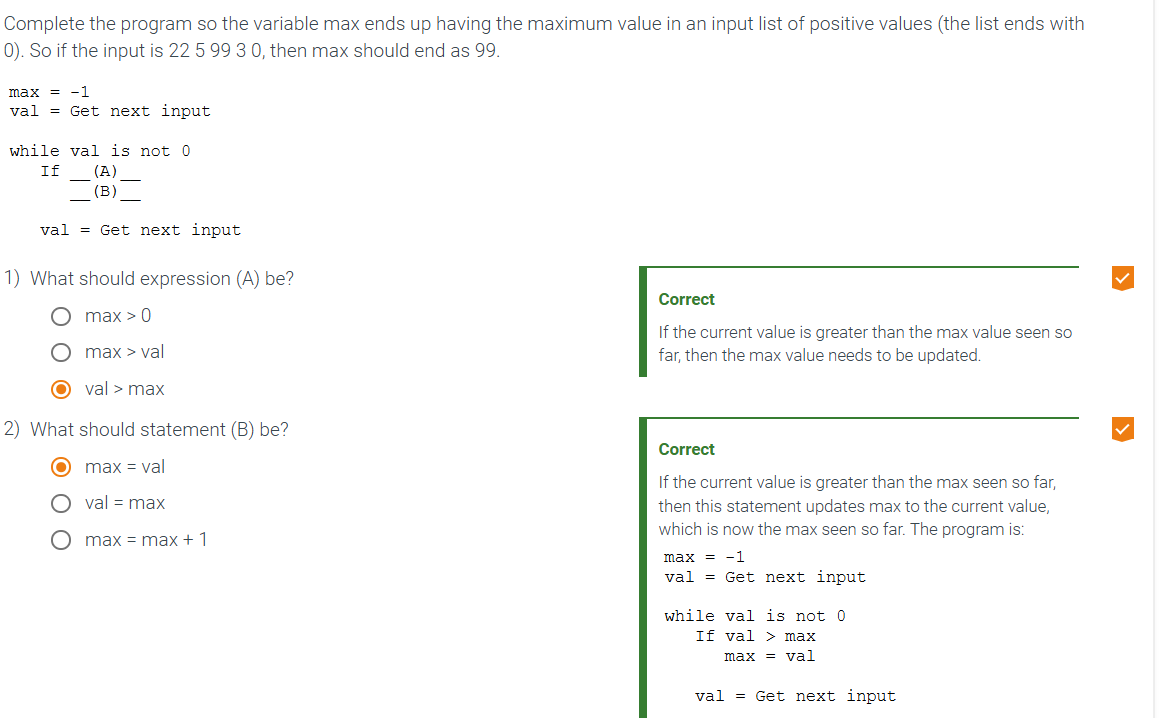


Chapter 5: Loops

5.1 Loops (general)

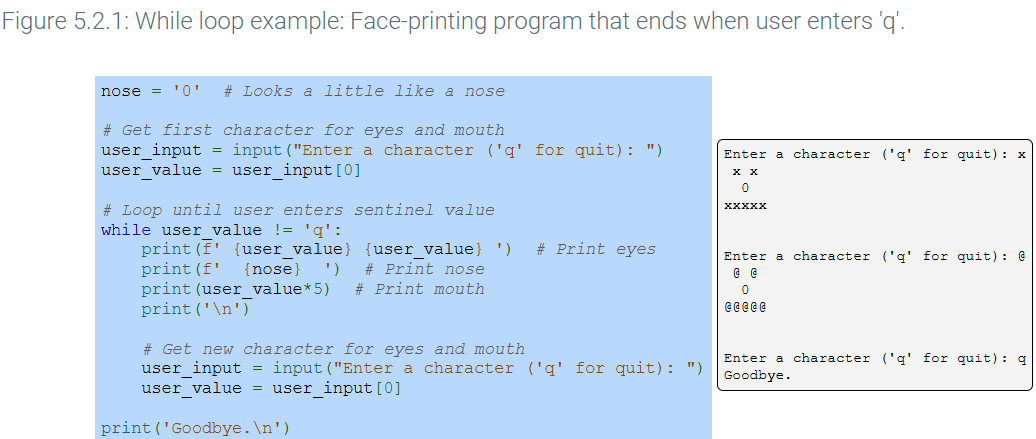
* A loop is a program construct that repeatedly executes the loop’s statements (known as the **loop body**) while the loop’s expression is true;
* When the expression is false, execution proceeds past the loop. Each time through a loop’s statement is called an **iteration**
* Programmers execute one statement at a time, Thus, using a loop to examine a list of values one value at a time and updating variables along the way





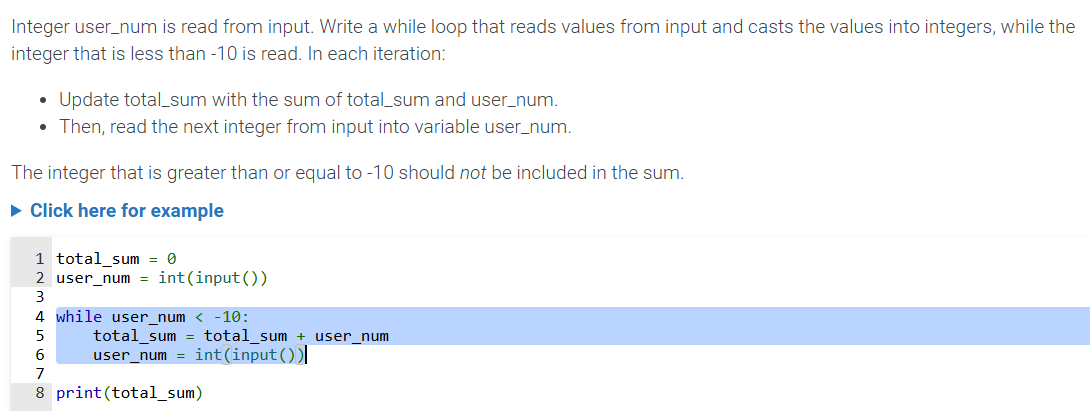
5.2 While loops

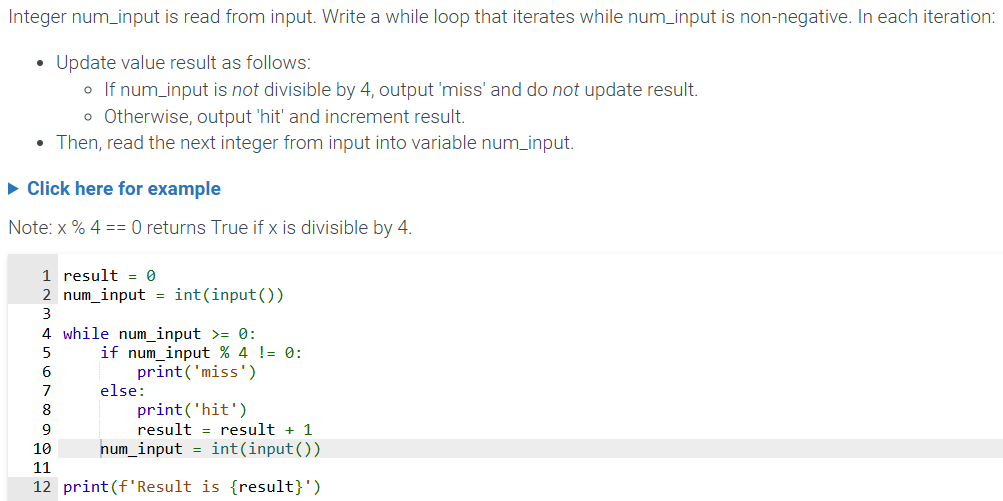
* A **while loop** is a construct that repeatedly executes an indented block of code (known as the **loop body)** as long as the loop’s expression is true. At the end of the loop body, execution goes back to the while loop statement and the loop expression is evaluated again.
* Statement is given as user\_vale != ‘q’
  + In this case, the letter ‘q’ is the **sentinel value,** a value that when evaluated by the loop expression causes the loop to terminate



* An infinite loop is a loop that will always execute because the loop’s expression is always True.

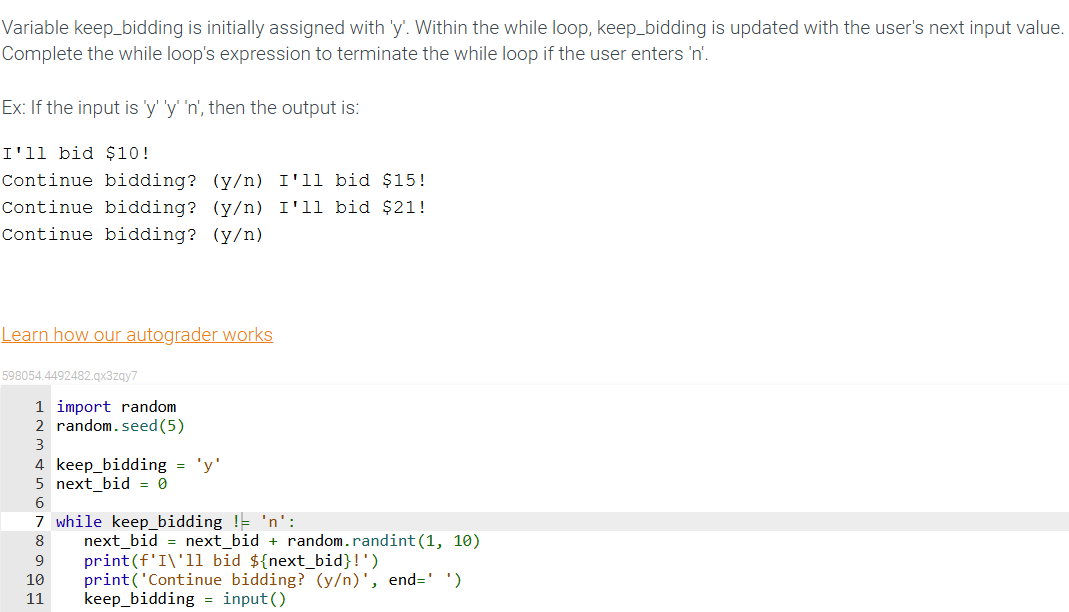


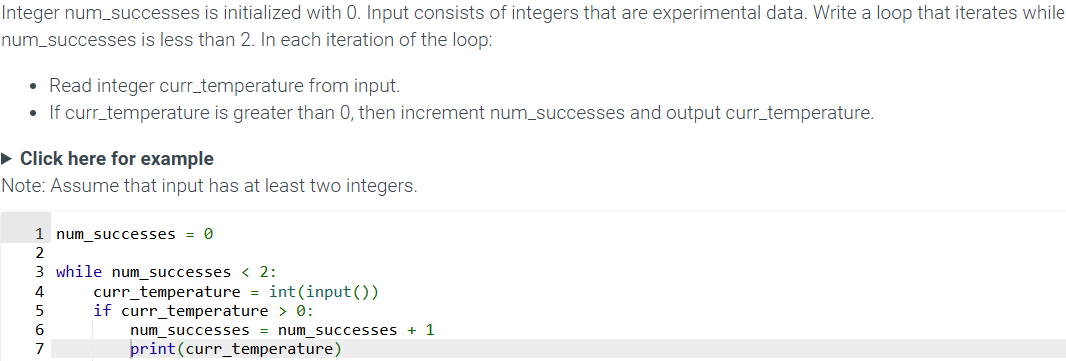


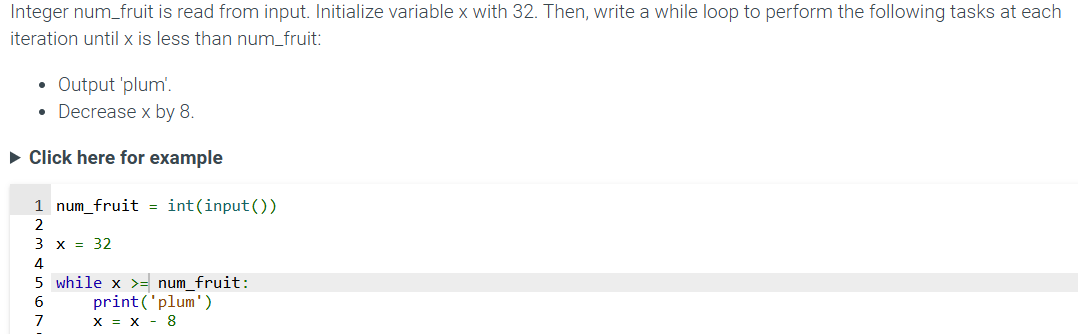


5.3 More while loop examples

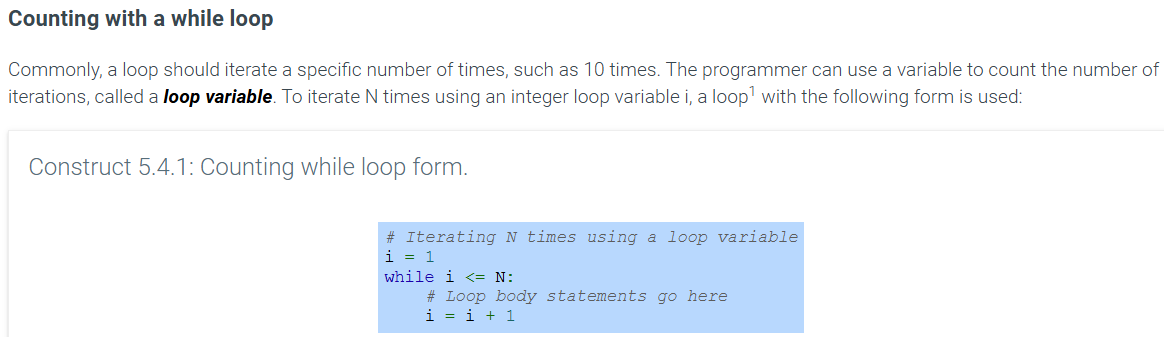
* Greatest common divisor example
* The randint() function provides a new random number each time the function is called
  + The argument to randint(), 0 and 2 provides the minimum and maximum values that the function may return.
* Loops are commonly used to process a series of input values. A sentinel value is used to terminate a loop’s processing.





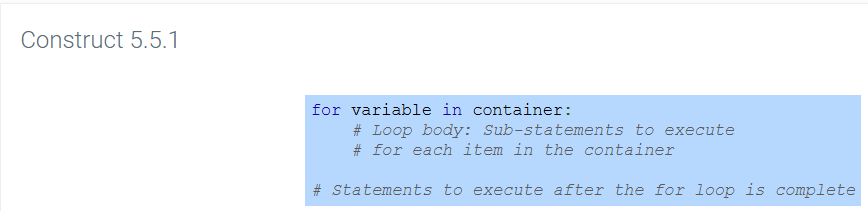


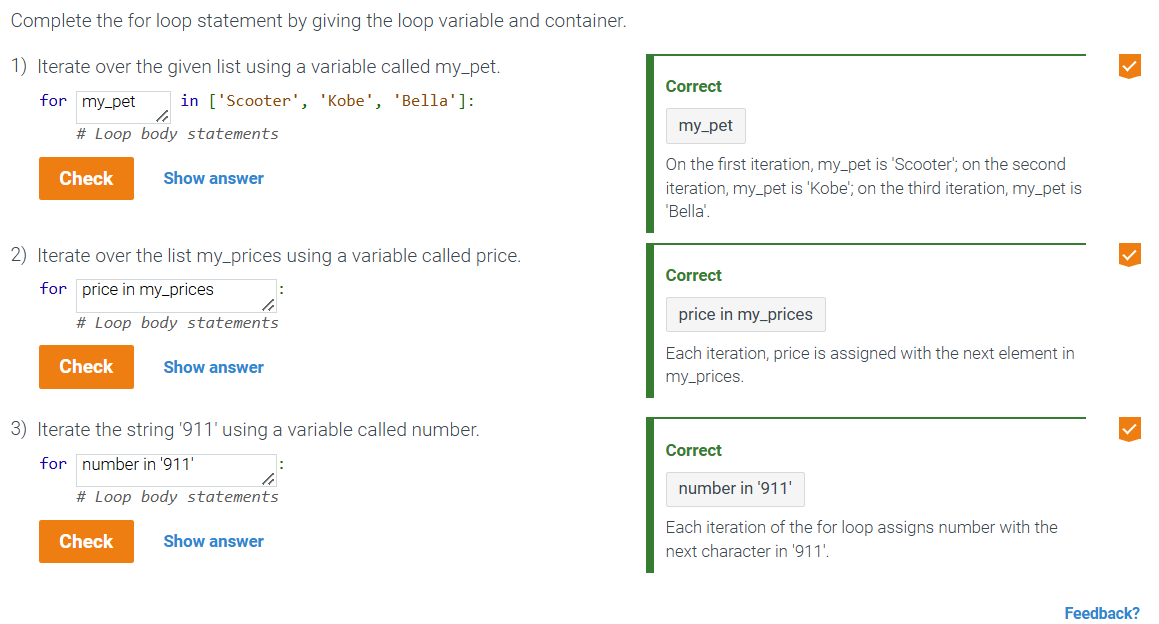
5.4 Counting

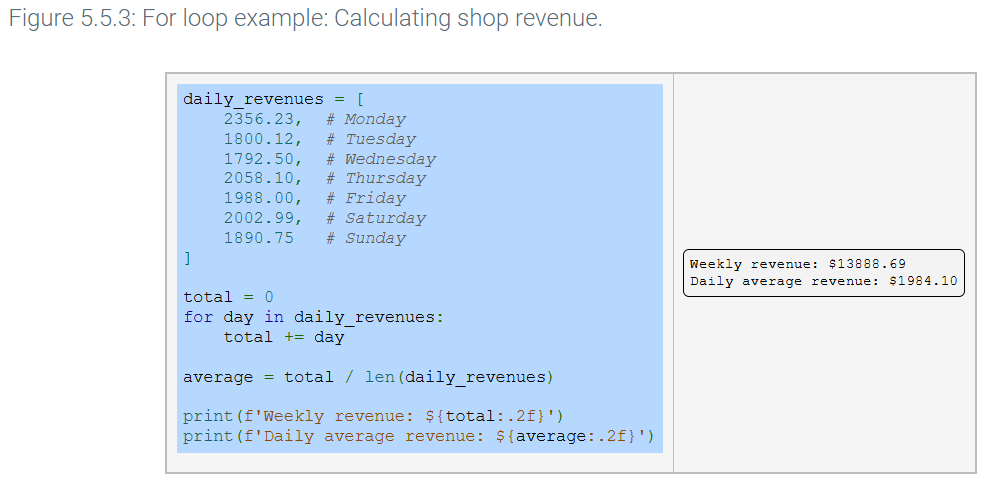
* The programmer can use a variable to count the number of iterations called a **loop variable** 
  + To iterate N times using an integer loop variable i, a loop1 with the following form i used:
* 

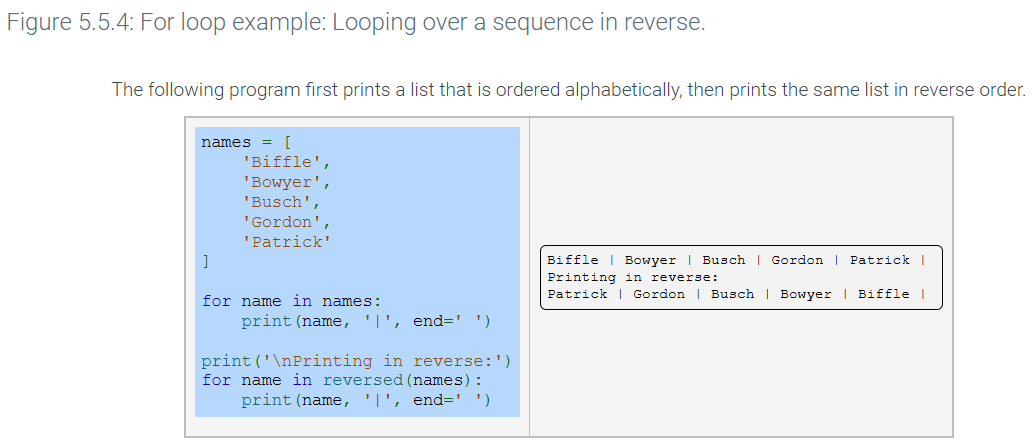
5.5 For loops

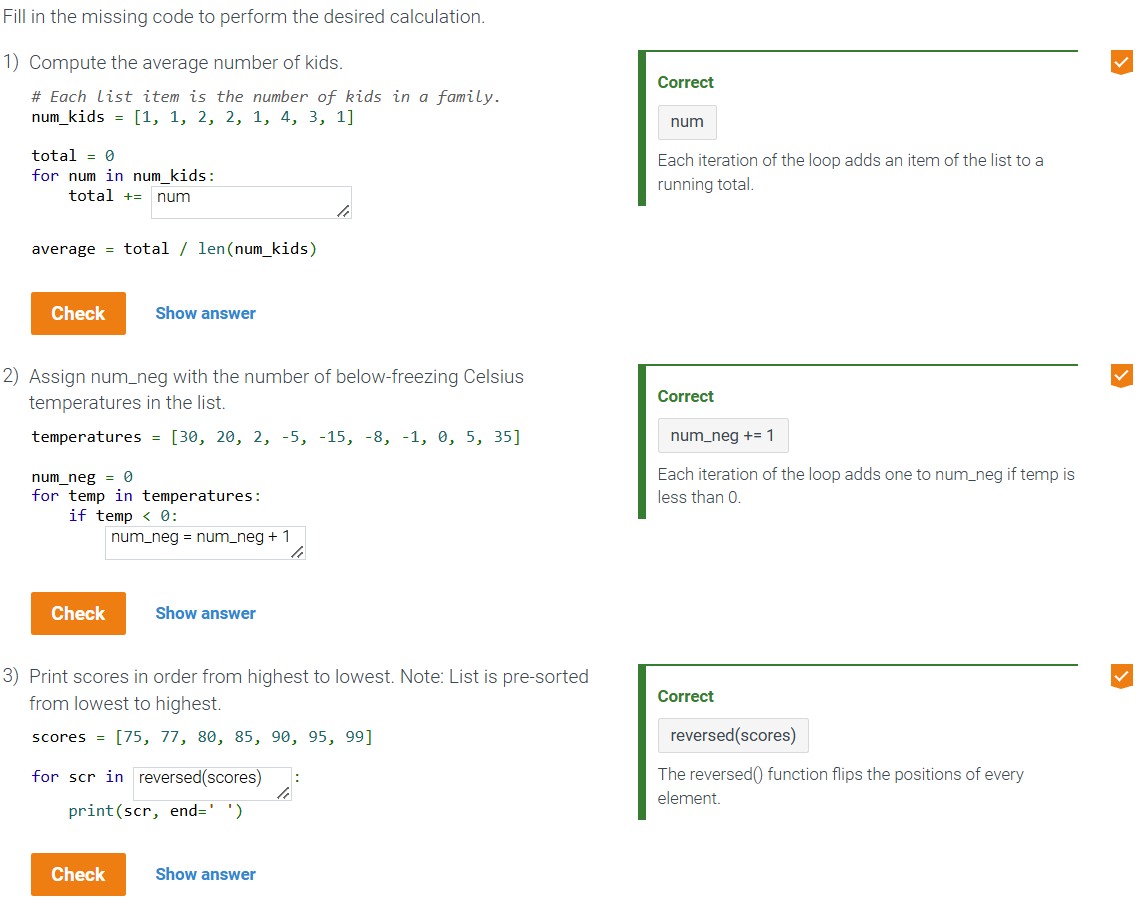
* A common programming task is to access all of the elements in a container. Ex: printing every item in a list.
  + A **for loop** statement loops over each element in a container one at a time, assigning a variable with the next element that can then be used in the loop body.
  + The container in the for loop statement is typically a list, tuple, or string.
  + Each iteration of the loop assigns the name given in the for loop statement with the next element in the container

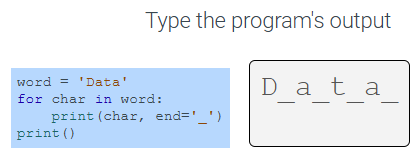






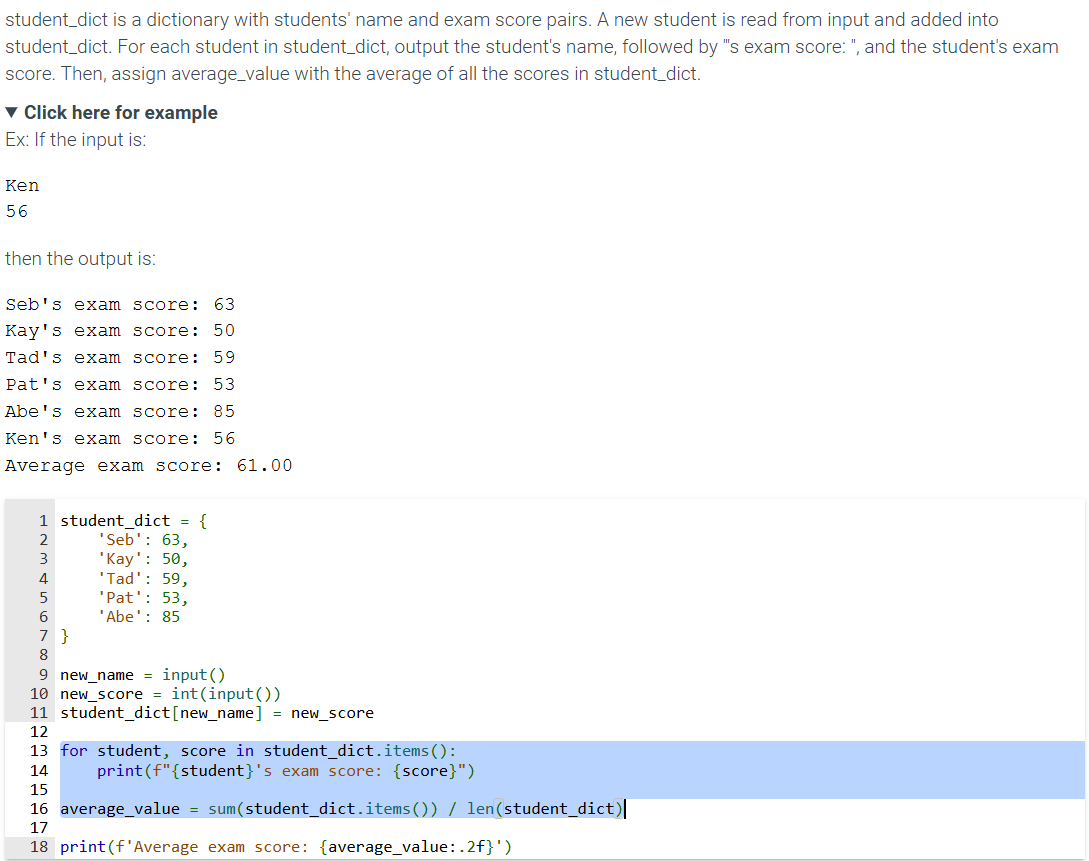






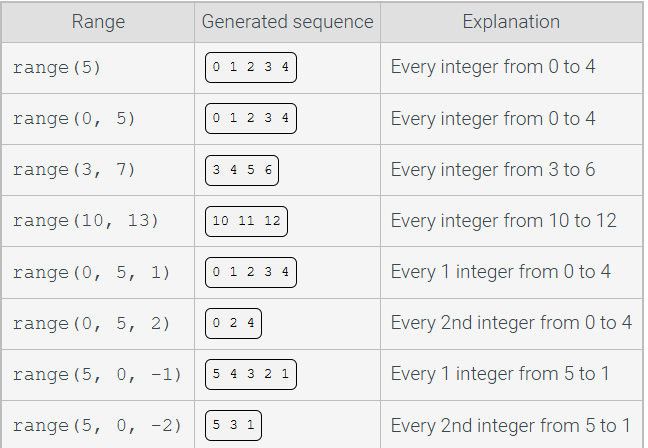


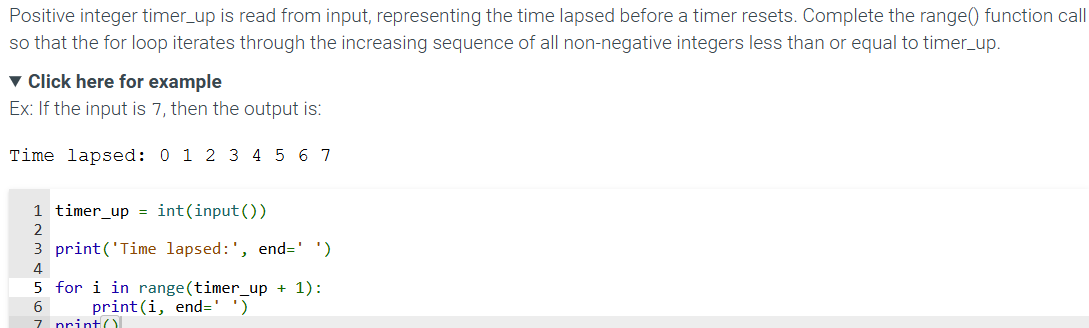


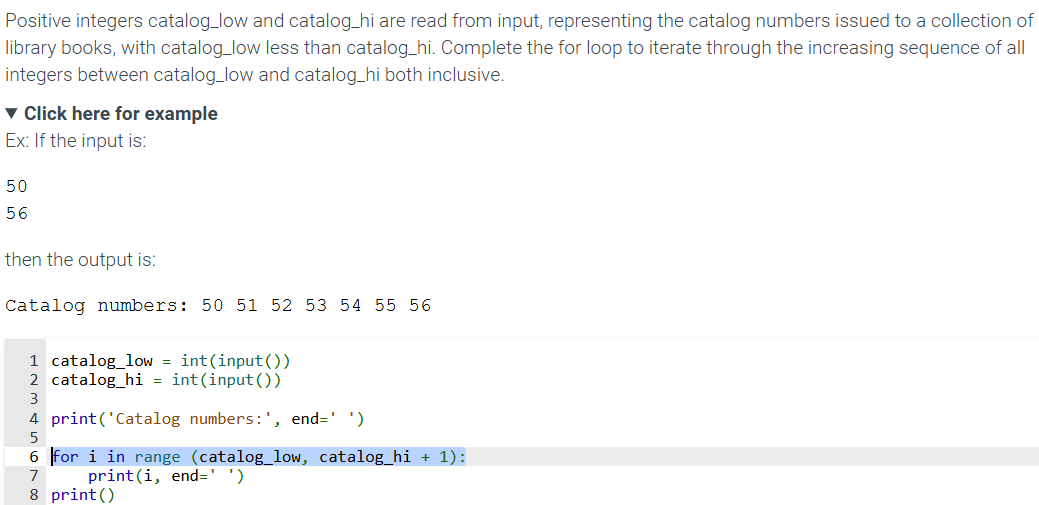


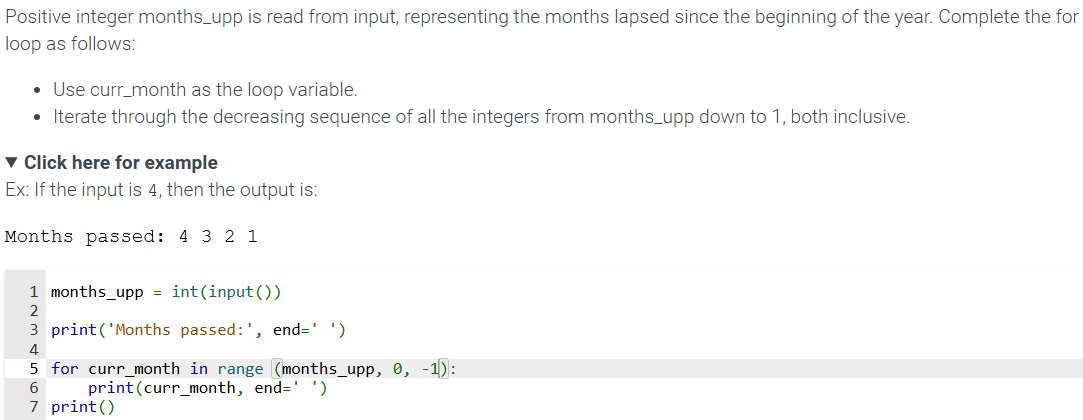
5.6 Counting using the range() function

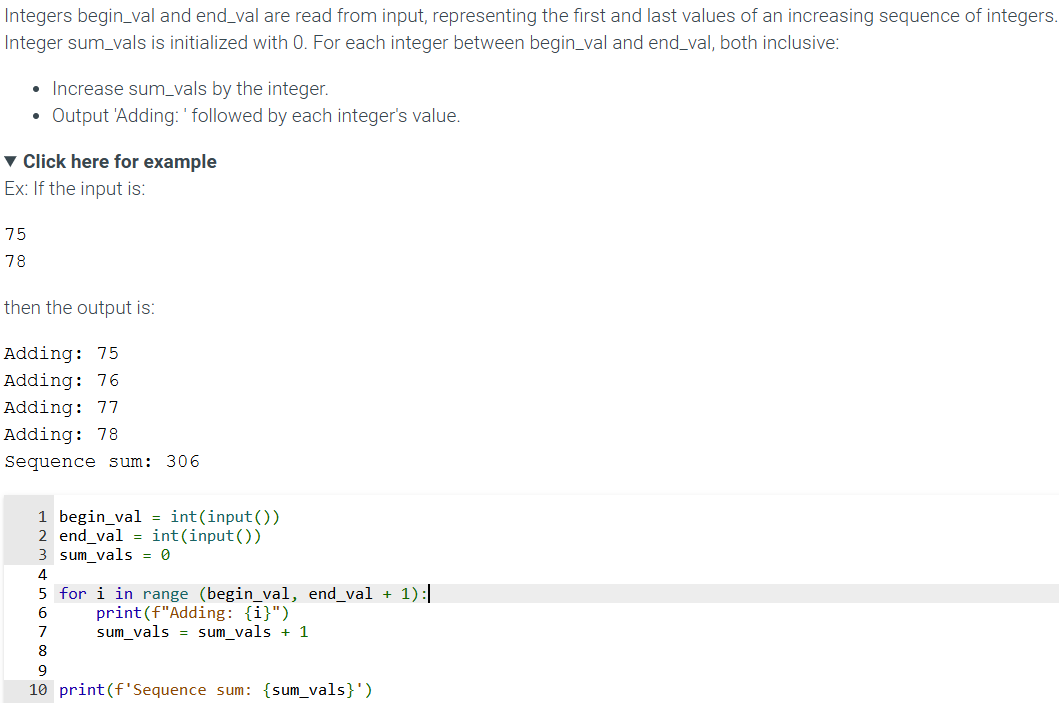
* **While loops** are used for counting a specific number of iterations
* **For loops** are used to iterate over all elements of a container.
* **range()** generates a sequence of integers between a starting integer that is included in the range, an ending integer that is not included in the range, and an integer step value.
  + The sequence is generated by starting at the start integer and incrementing by the step value until the ending integer is reached or surpassed.





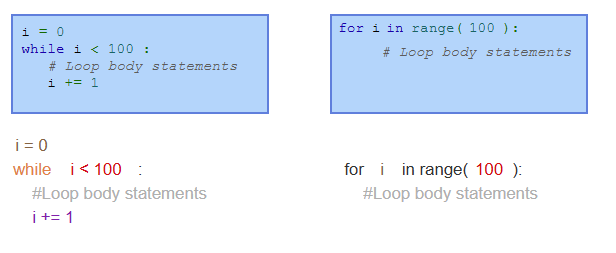




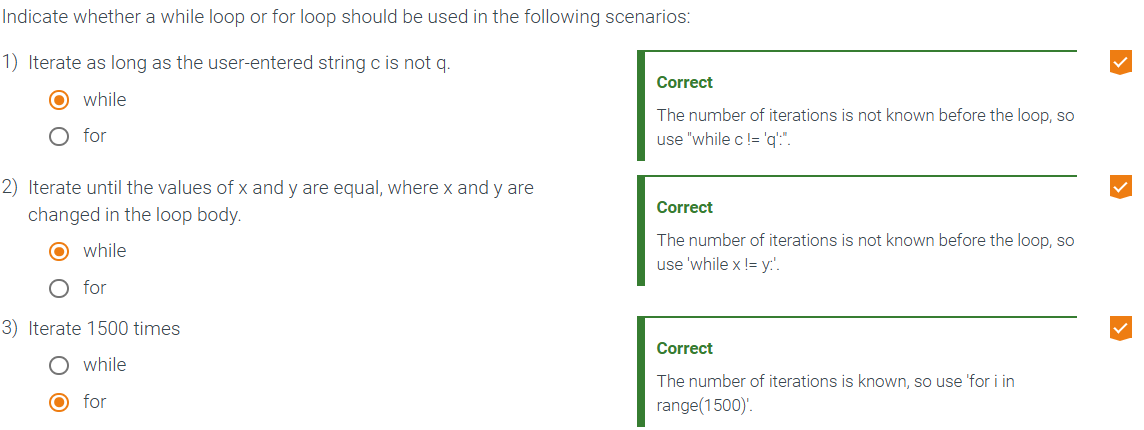


5.7 While vs. For loops

* Both while loops and for loops can be used to count a specific number of loop iterations.
* A for loop combined with range() is generally preferred over while loops, since for loops are less likely to become stuck in an infinite loop situation.

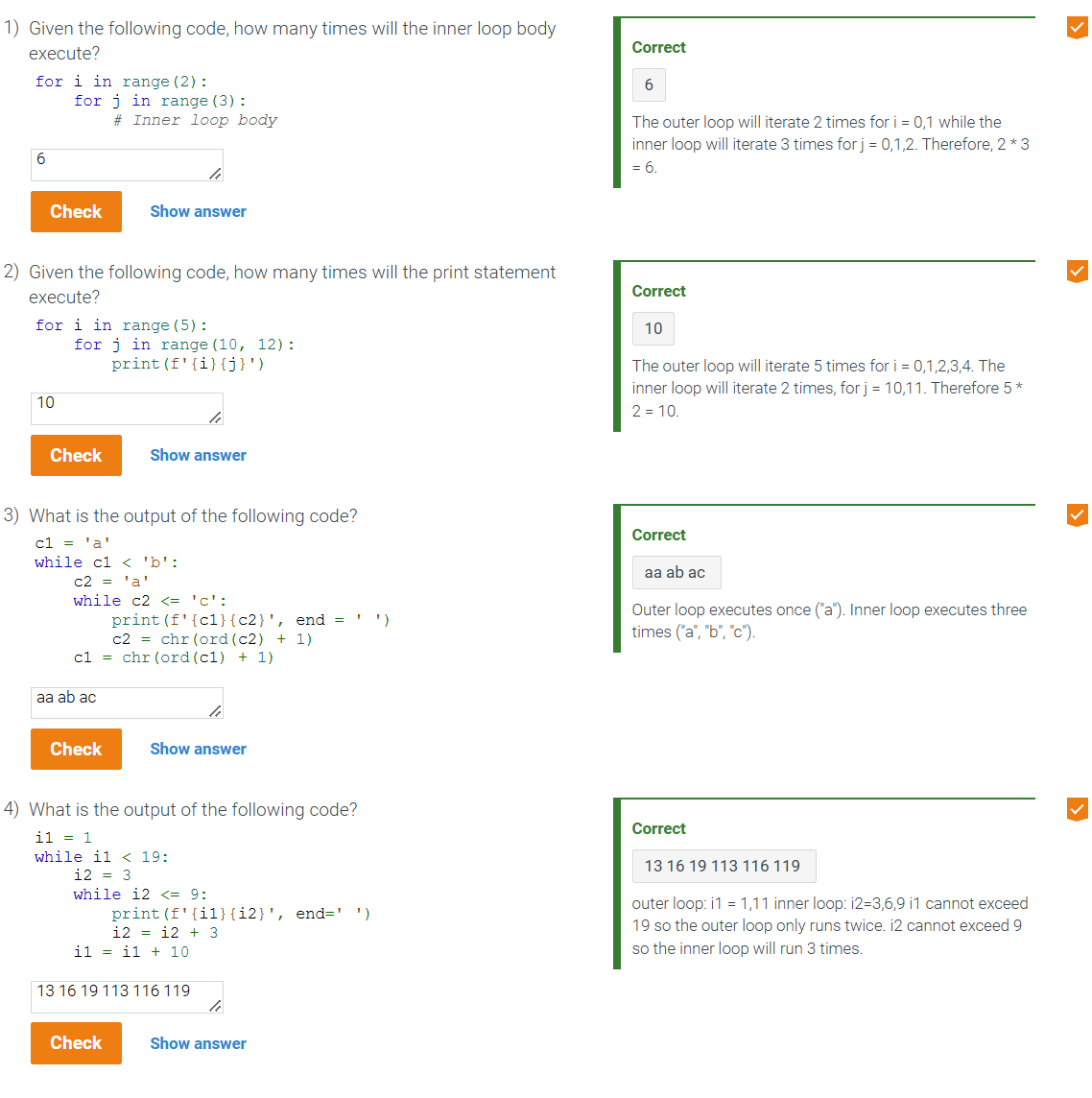


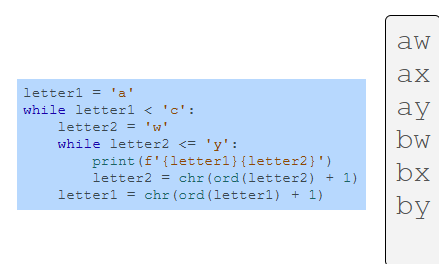
* Use a **for loop** when the number of iterations in computable before entering the loop, as when counting down from X to 0, printing a string N times, etc
* Use a **for loop** when accessing the elements of a container, as when adding 1 to every element in a list, or printing the key of every entry in a dict, etc.
* Use a **while loop** when the number of iterations in not computable before entering the loop, as when iterating until a user enters a particular character.

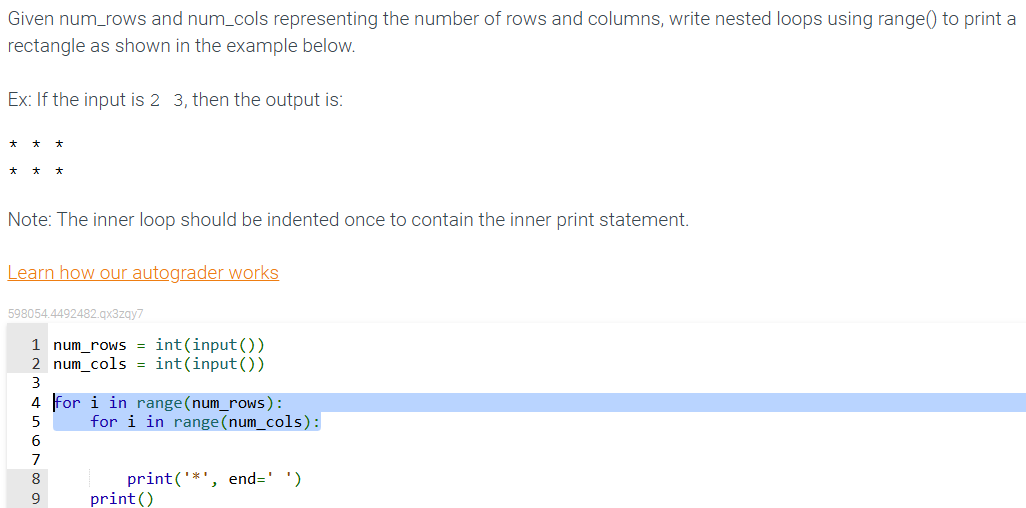


5.8 Nested loops

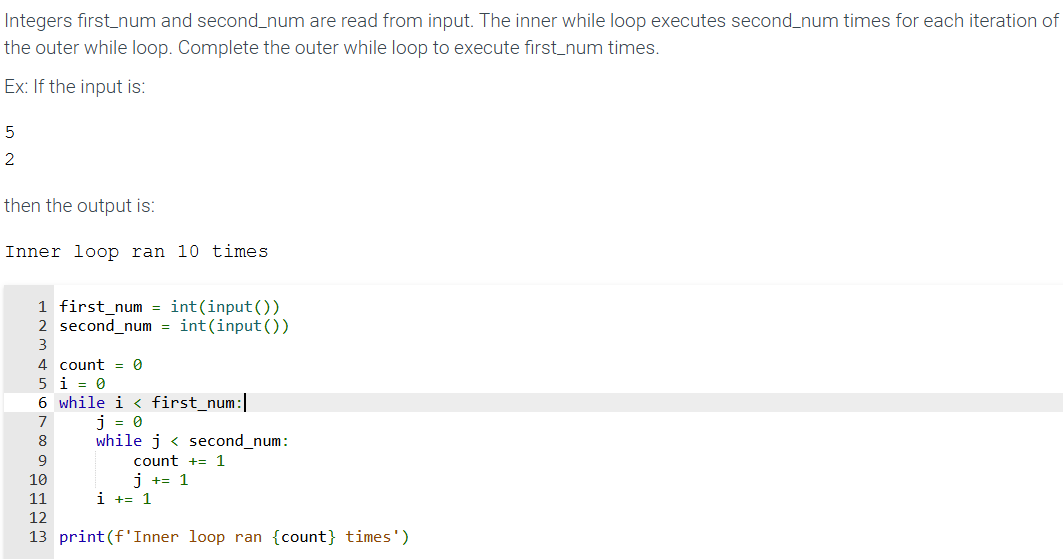
* A **nested loop** is a loop that appears as part of the body of another loop. The nested loops are commonly referred to as the **outer loop** and **inner loop**
  + Nested loops have various uses. One use is to generate all combinations of some items.

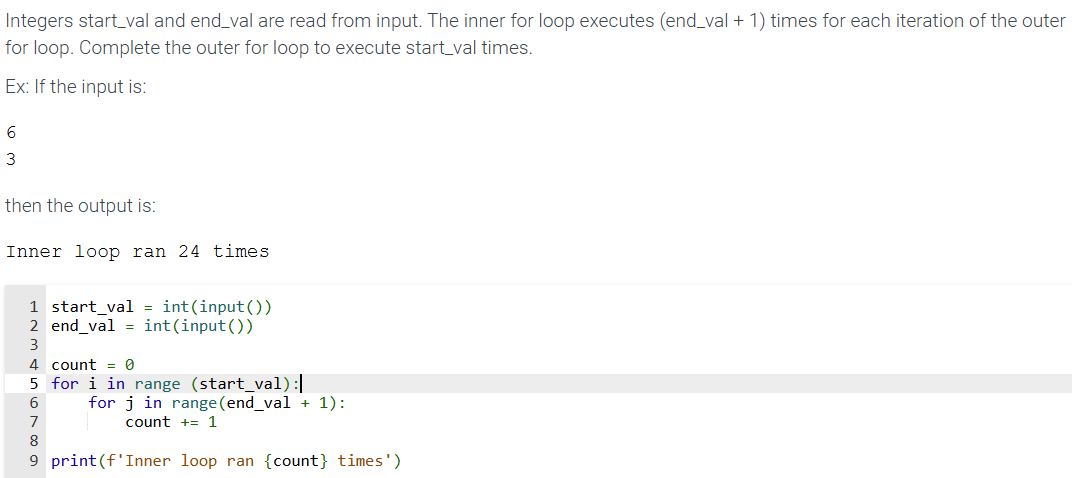


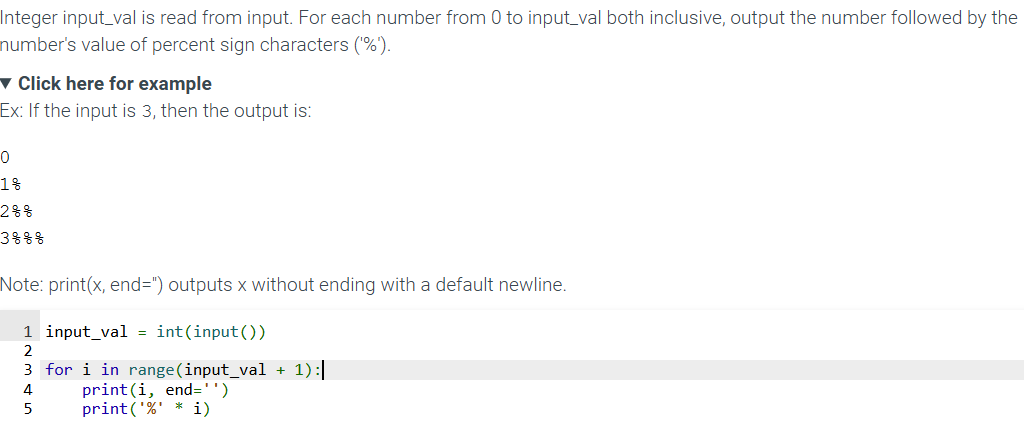








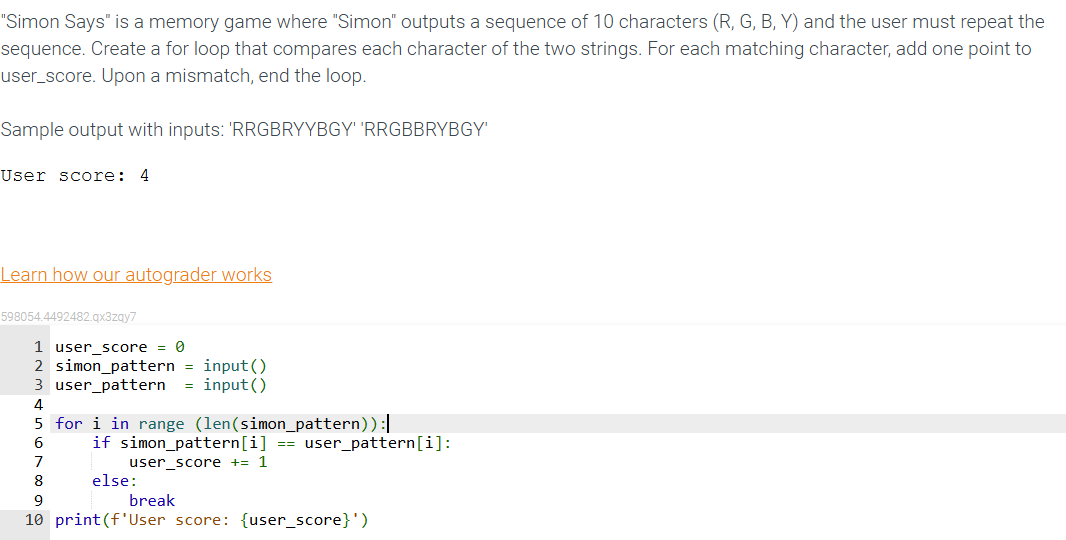




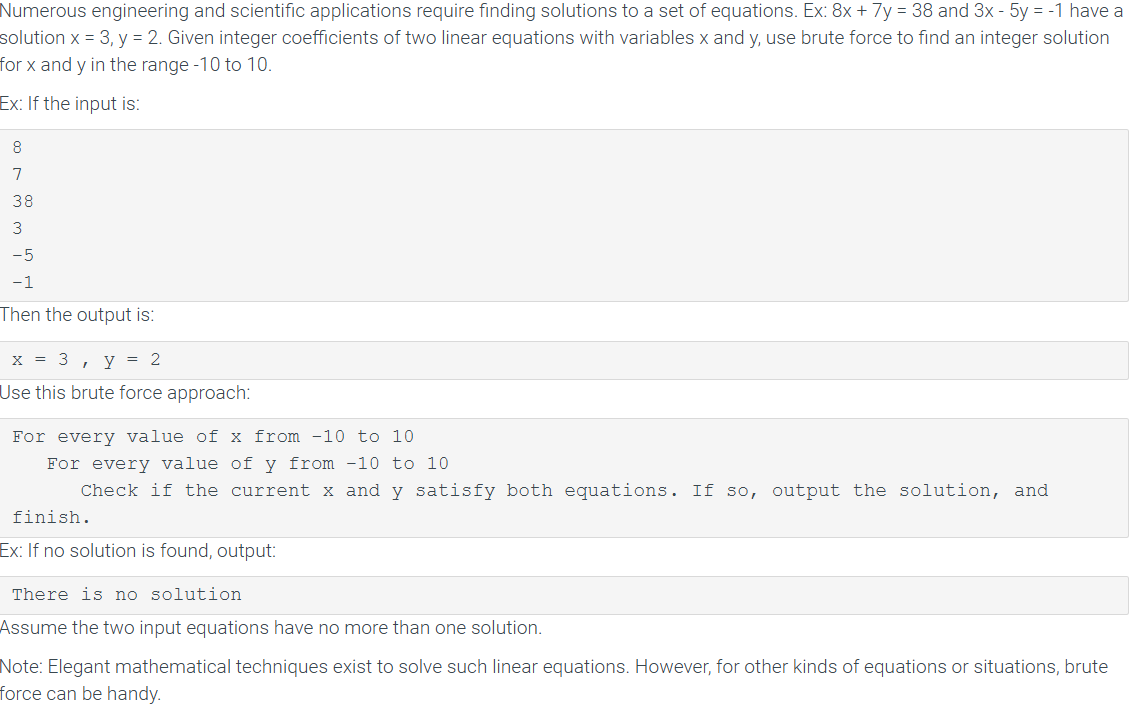


5.10 Break and continue

* A **break** statement in a loop causes the loop to exit immediately. A break statement can sometimes yield a loop that is easier to understand
* A **continue** statement in a loop causes an immediate jump to the while or for loop header statements.



5.18- LAB: Brute force equation solver



''' Read in first equation, ax + by = c '''

a = int(input())

b = int(input())

c = int(input())

''' Read in second equation, dx + ey = f '''

d = int(input())

e = int(input())

f = int(input())

solution\_found = False

for x in range(-10, 11): *# Iterate x from -10 to 10*

for y in range(-10, 11): *# Iterate y from -10 to 10*

*# Check if the current x and y satisfy both equations*

if (a \* x + b \* y == c) and (d \* x + e \* y == f):

print(f'x = {x} , y = {y}')

solution\_found = True

break *# Exit the inner loop if a solution is found*

if solution\_found:

break *# Exit the outer loop if a solution is found*

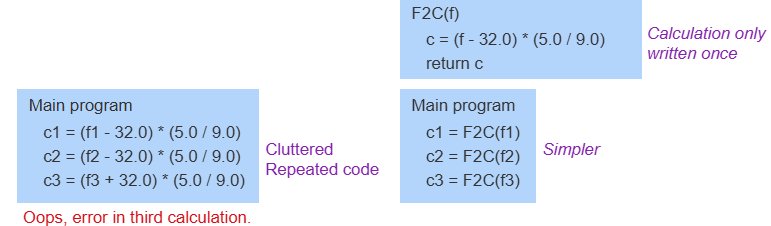
if not solution\_found:

print("There is no solution")

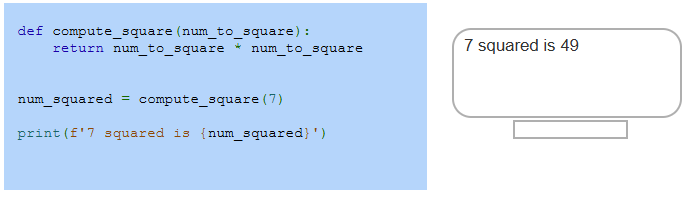
Chapter 6 - Functions

6.1 User-defined function basics

* A programmer may perform the same operation repeatedly, causing a large and confusing program due to redundancy.
* Program redundancy can be reduced by creating a grouping of predefined statements for repeated operations, known as a **function**. Even without redundancy, functions can prevent a main program from becoming large and confusing.



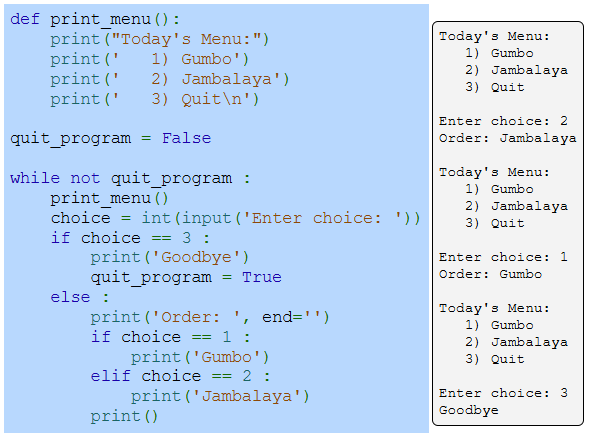
* A function is a named series of statements
  + A **function definition** consists of a functions name and a block of statements
    - Ex: def calc\_pizza\_area() : is followed by an indented block of statements
  + A **function call** is an invocation of the functions name, causing the functions statement to execute.
* Python comes with a number of built-in functions such as input(), int(),, len(), etc.
  + The **def** keyword is used to create new functions.
* A function may return one value using the **return statement.**



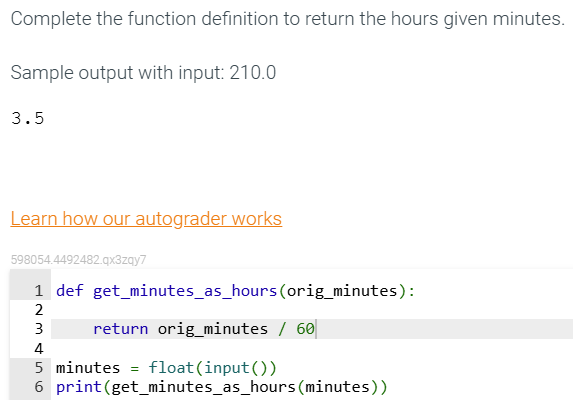
* A function may return one item, not two or more (though a **list** or **tuple** with multiple elements could be returned.
* A function with no return statement, or a return statement with no following expression returns the value as NONE.
* **None** is a special keyword that indicated no value
  + A return statement may appear at any point in a function, not just as the last statement. A function may also contain multiple return statements in different locations.
* FROM ME: RETURN STATEMENT IS THE EQUATION PART
* A programmer can influence a functions behavior via an input
  + A **parameter** is a function input specified in a function definition.
    - Ex: A pizza area function might have diameter as an input.
  + An **argument** is a value provided to a function's parameters during a function call.
    - Ex: A pizza area function might be called as calc\_pizza\_area(12.0) or as calc\_pizza\_area(16.0)
* The parameter is like a variable definition. Upon entering the function, the parameter is bound to the arguments objects provided by the call, creating a shared reference to the object.
* A **function** might have multiple **parameters** separated by commas.
* Parameters are assigned with argument values: First parameter with the first argument, second with the second.
* A functions definition with no paraamenters must still have the parentheses, as in

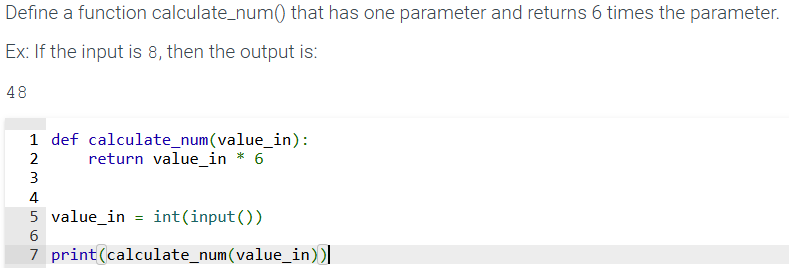
def calc\_something():

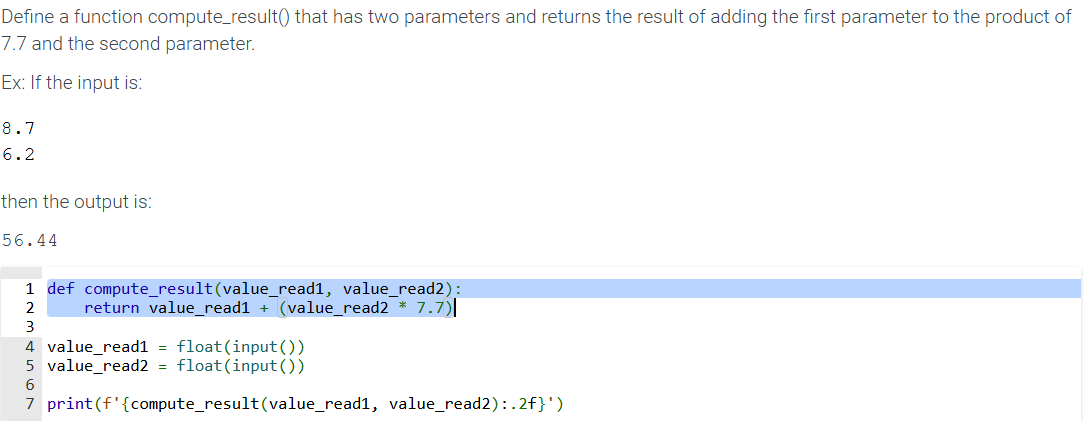






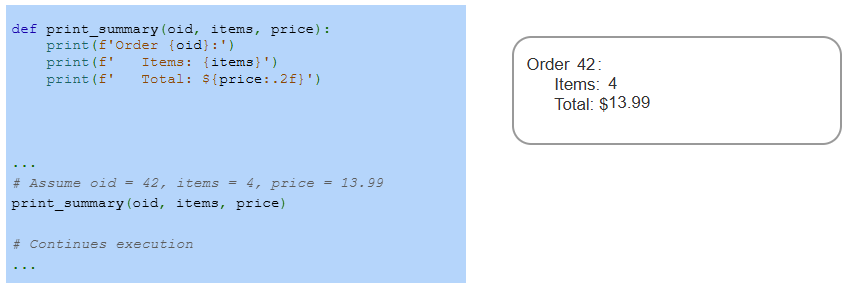


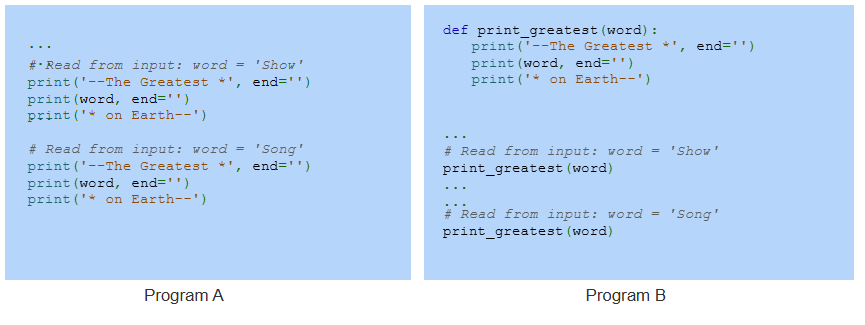


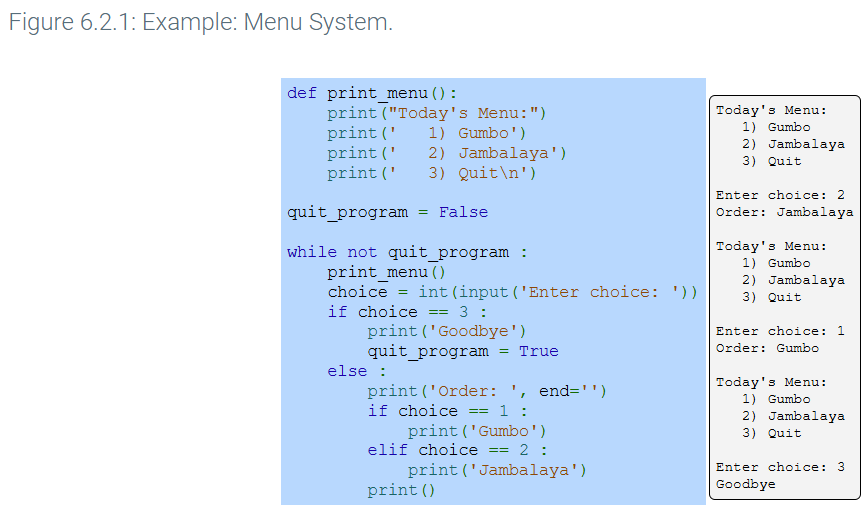


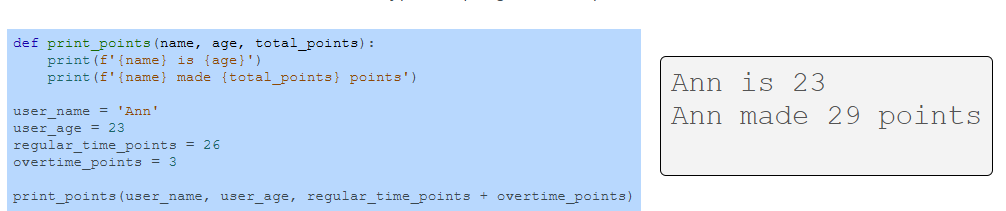
6.2 Print functions

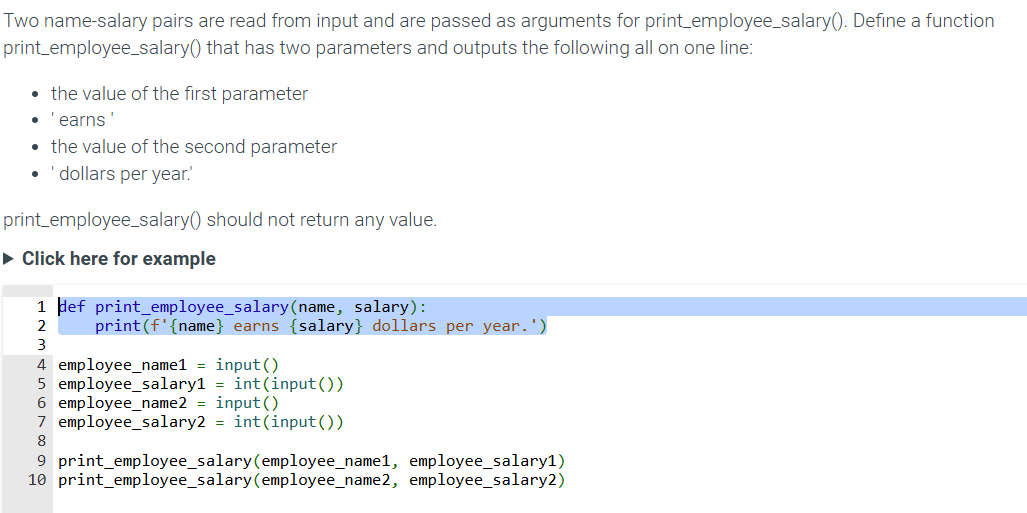
* A common operation for a function is to print text
  + A function that only prints typically does not return a value.
  + A function with no return statement is called a **void function,** and such a function returns a value None

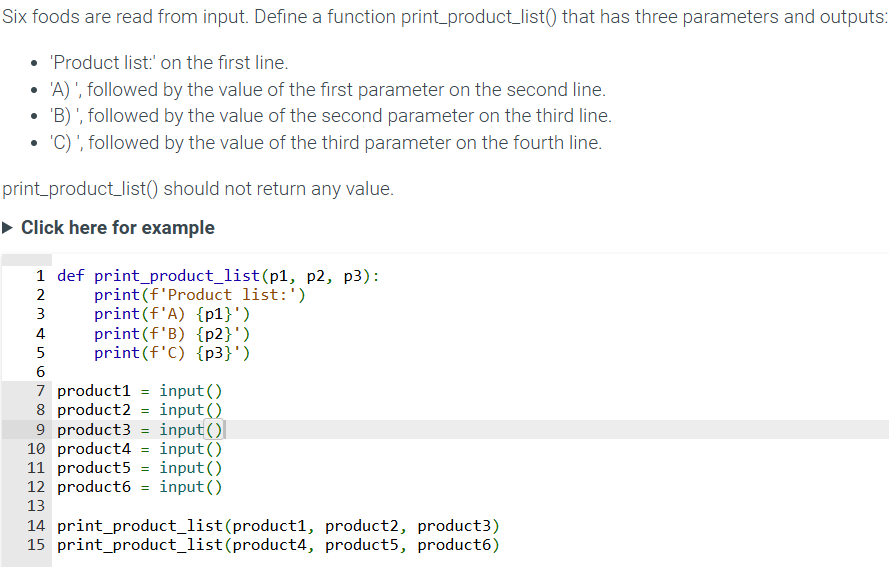


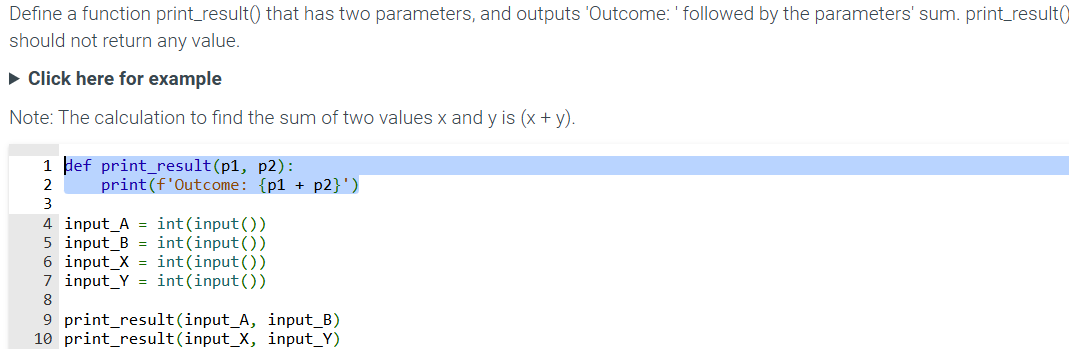


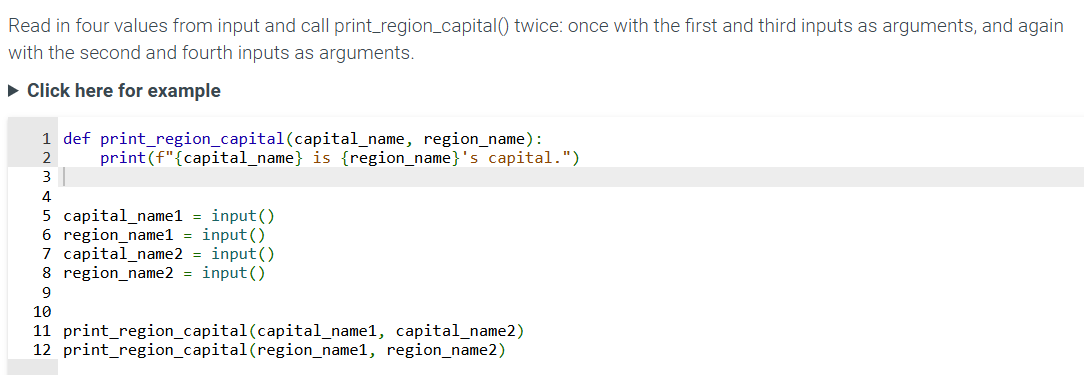








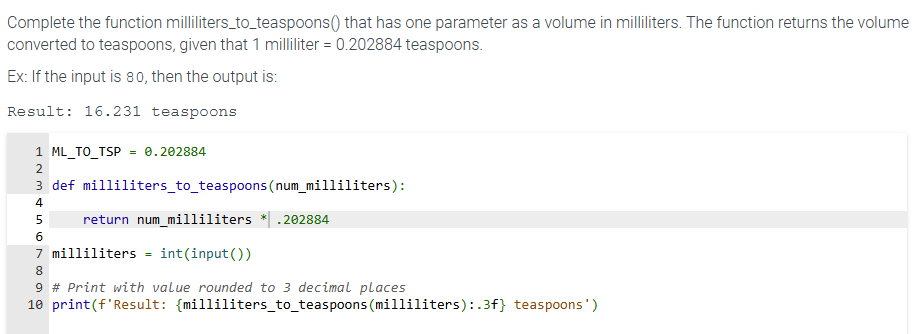




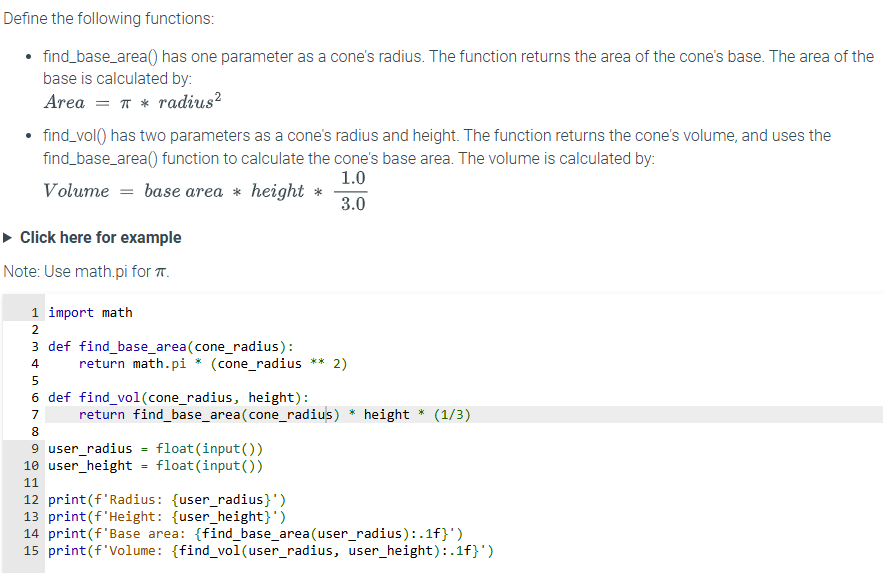
6.4 - Reasons for defining functions







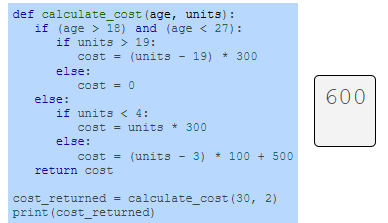




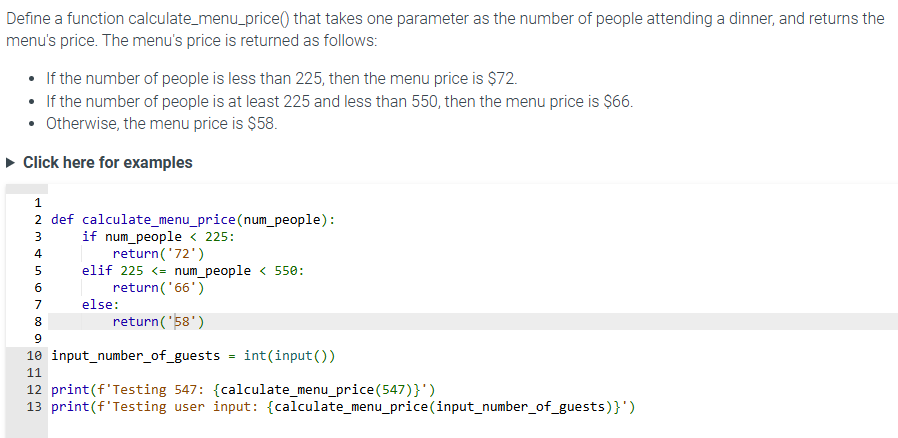
6.6 Function stubs

* **Function stubs** are function definitions whose statements haven’t been written yet.
  + Planning the route of a road trip….
* Use a **pass** keyword which performs no operations except to act as a placeholder for a required statement.

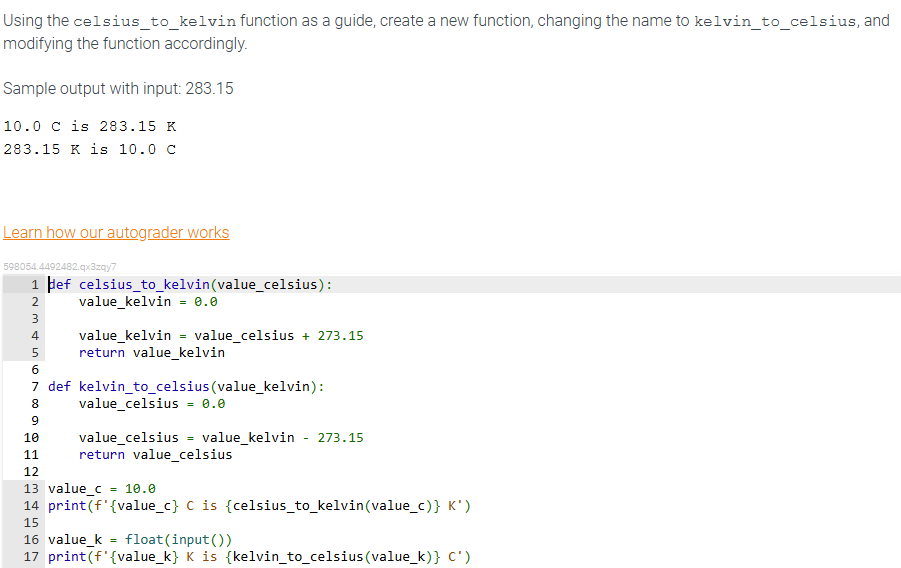
6.7 Functions with branches/loops





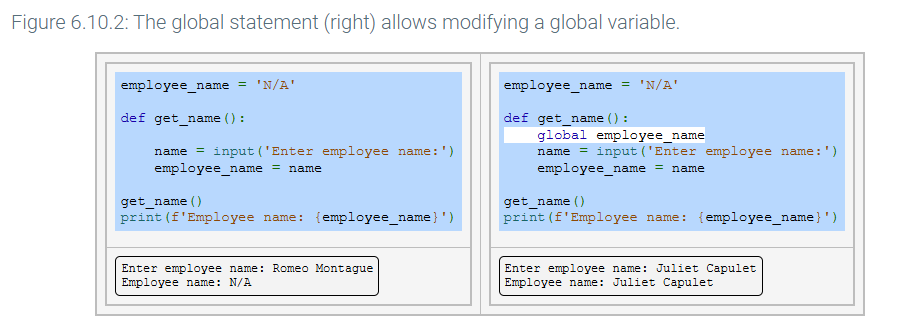






6.10 Scope of variables and functions

* A variable or function object is only visible to part of a program, known as the object's scope.
* Variables defined outside of a function are called a **global variable**.
  + A global variables scope extends from the assignment to the end of the file, and can be accessed inside of functions.
* A **global** statement must be used to change the value of a global variable inside of a function.

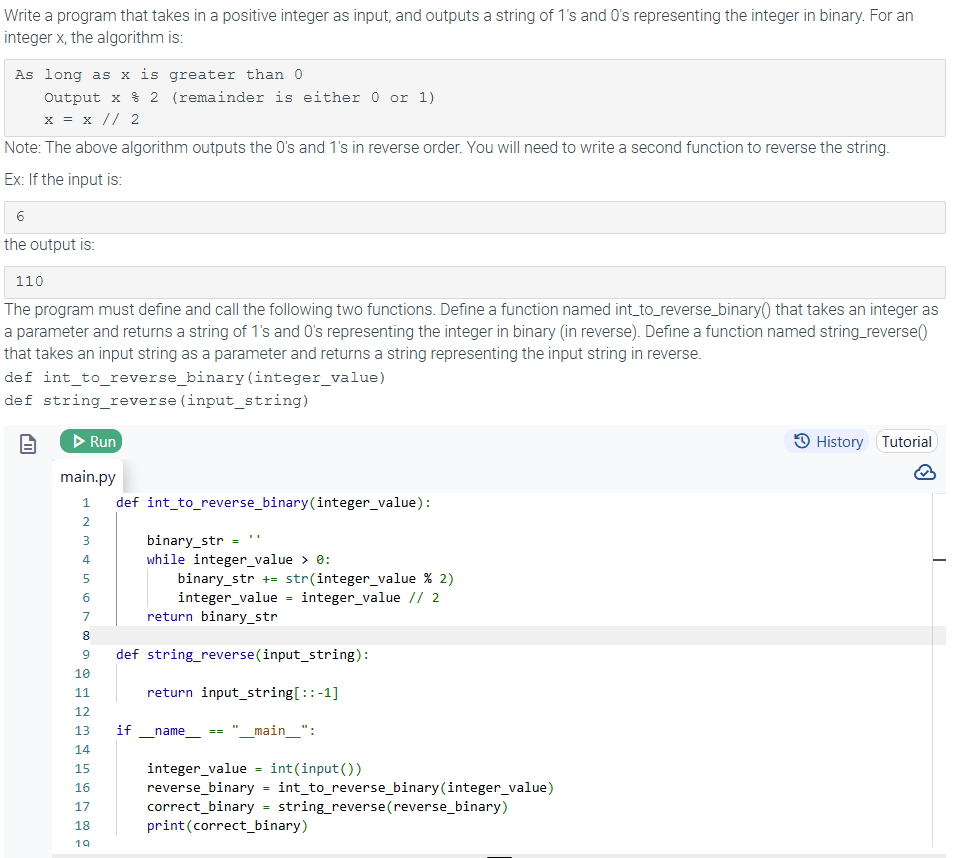


6.13 Keyword arguments and default parameter values

* Sometimes, functions have parameters that are optional. A function can have a default parameter value for one or more parameters, meaning that a function call can optionally omit an argument, and the default parameter value will be substituted for the corresponding omitted argument.



6.21 LAB: Convert to binary - functions



6.23 LAB: Fibonacci sequence



7. Classes

7.1 Classes: introduction

* Program is made up of lower-level items like variables and functions. To keep a program understandable, programmers often deal with higher-level grouping of those items, known as objects.
  + In programming, an **object** is a grouping of data (variables) and operations that can be performed on the data (functions or methods)
* **Abstraction** occurs when a user interacts with an object at a high level, allowing lower-level internal details to remain hidden. (aka **information hiding** or **encapsulation)** 
  + EX: An oven supports an abstraction of a food compartment and a knob to control the heat. An oven user does not need to interact with the internal parts of an oven.
* An **abstract data type (ADT)** is a data type whose creation and update are constrained to specific well-defined operations. A class can be used to implement an ADT.
* Python automatically creates **built-in** objects for a programmer to use and include the basic data types like integers and strings.
  + A programmer always interacts with built in objects when writing python code.
    - Ex: A string object created with mystr = ‘Hello!’ The value of the string ‘Hello!’ is one part of the object, as are functions to operate on that string like str.isdigit() and str.lower()

7.2 Classes: Grouping data

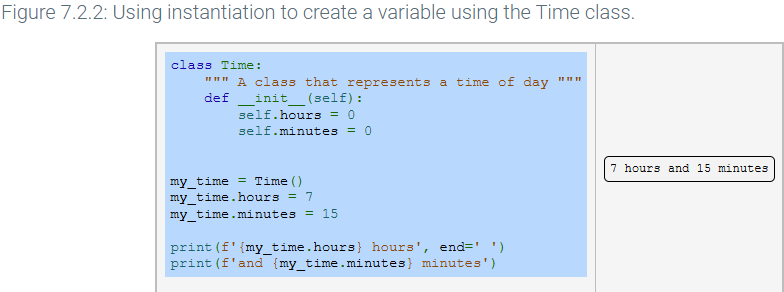
* Multiple variables are frequently closely related and should be treated as one variable with multiple parts.
  + Ex: two variables called hours and minutes might be grouped together as a single variable called time.
    - The **class** keyword can be used to create a user-defined type of object containing groups of related variables and functions.
* A **class** defines a new type that can group data and functions to form an object.
  + The object maintains a set of **attributes** that determines the data and behavior of the class.
    - EX: following code defines a new class containing two attributes, hours and minutes, whose values are initially 0:

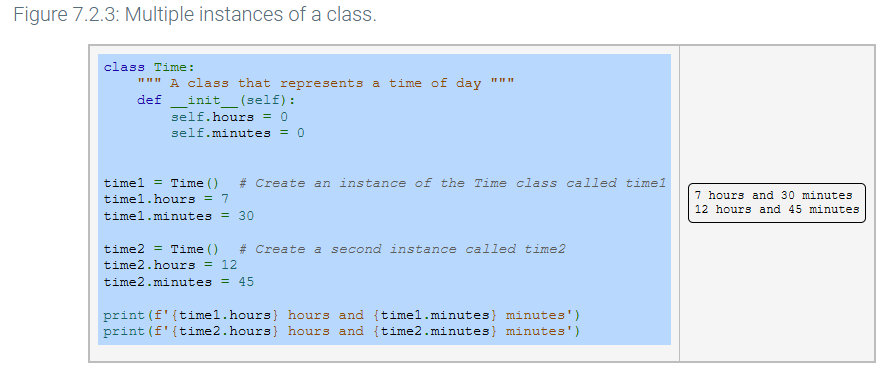


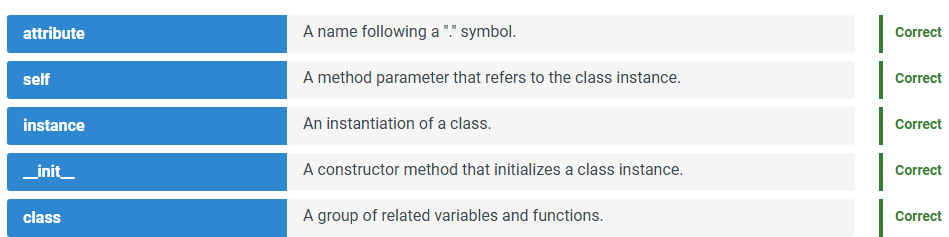
* Programmers can then use instantiation to define a new Time class variable and access the variable attributes.
  + An **instantiation** operation is performed by “calling” the class, using parentheses like a function call as in my\_time = time()
    - An instantiation operation creates an **instance**, which is an individual object of the given class.
      * An instantiation operation automatically calls the \_init\_ method defined in the class definition.
        + A **method** is a function defined within a class.

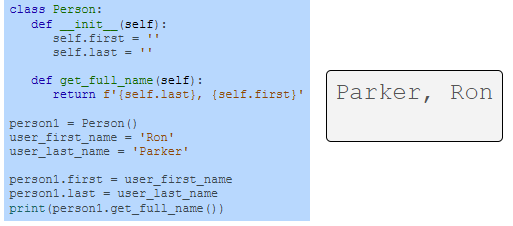
The \_init\_ method, commonly known as a constructor, is responsible for setting up the initial state of the new instance.

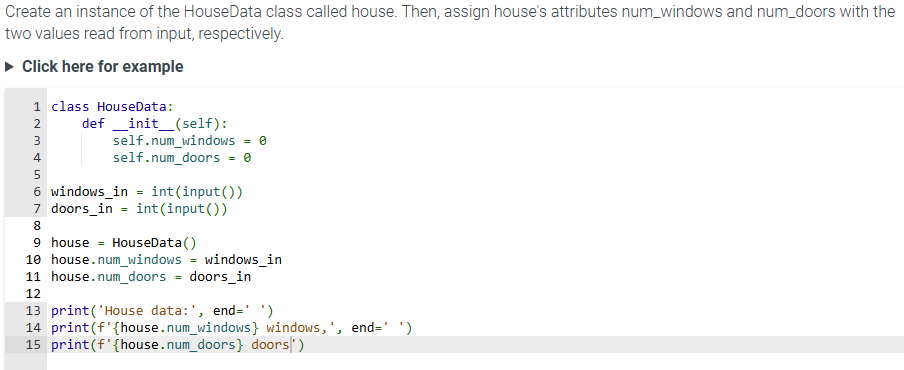
* The \_init\_ method has a single parameter “self” that automatically references the instance being created.
  + A programmer writes an expression such as self.hours = 0 within the \_init\_ method to create a new attribute, hours.

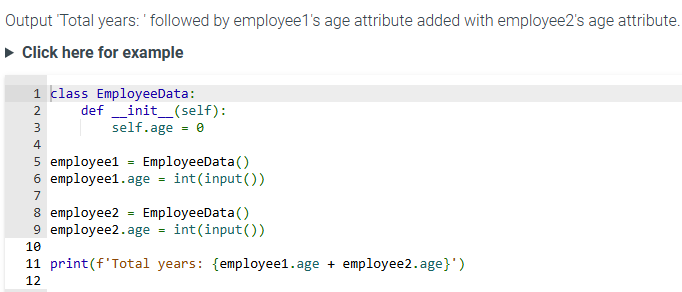


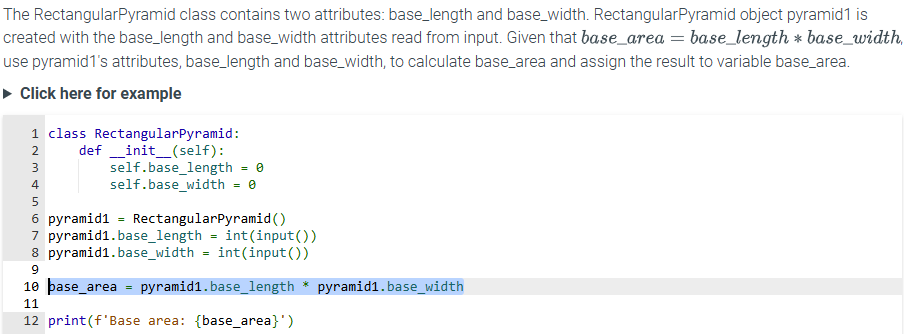
* A programmer can create multiple instances of a class in a program, with each instance having different attribute values.
* 













class BookData:

def \_\_init\_\_(self):

self.num\_chapters = 0

self.num\_pages = 0

self.genre = 'unknown'

book = BookData()

print('Book data (before):', end=' ')

print(f'{book.num\_chapters} chapters,', end=' ')

print(f'{book.num\_pages} pages,', end=' ')

print(f'{book.genre}')

book.num\_chapters = int(input())

book.num\_pages = int(input())

book.genre = input()

print('Book data (after):', end=' ')

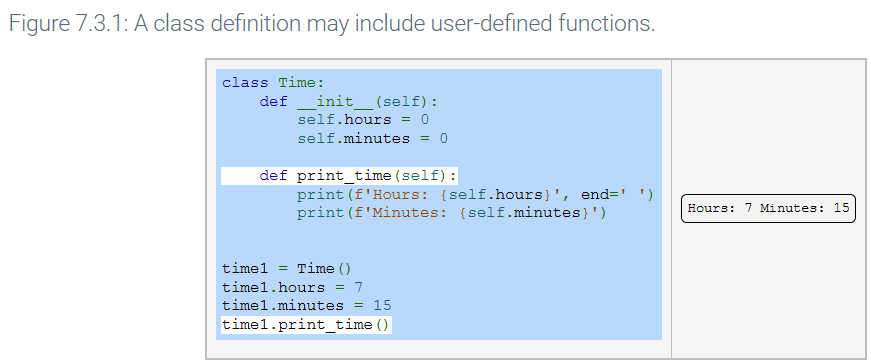
print(f'{book.num\_chapters} chapters,', end=' ')

print(f'{book.num\_pages} pages,', end=' ')

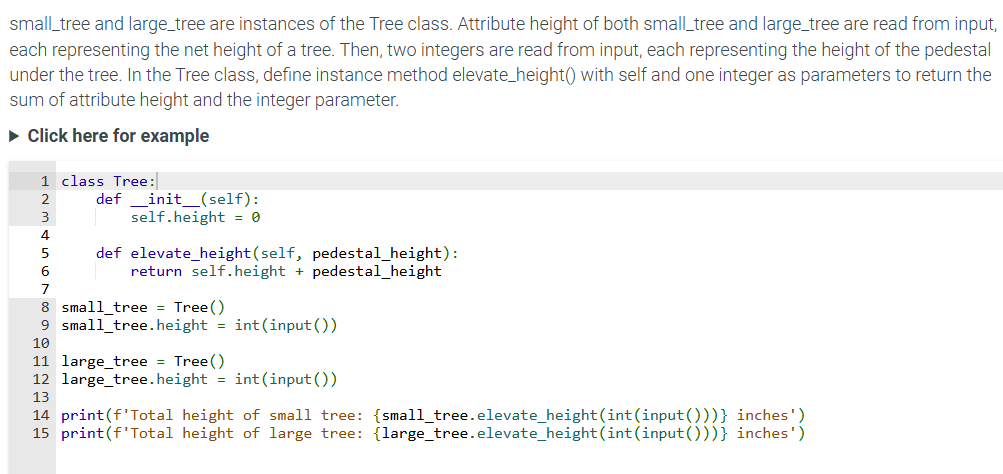
print(f'{book.genre}')

7.3 Instance methods

* A function defined within a class is known as an **instance method**.
  + An instance method can be referenced using dot notation.



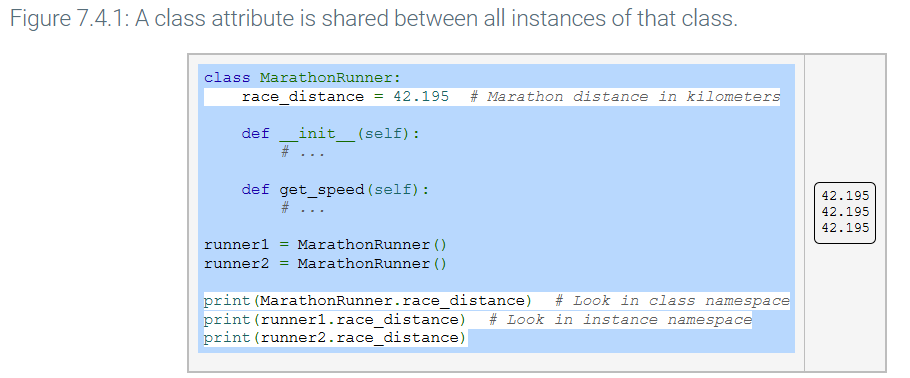
* The definition of print\_time() has a parameter “self” that provides a reference to the class instance.
  + “Self” is bound to time1 when time1.print\_time() is called.
    - A programmer does not specify an argument for “self” when calling the function;
      * The methods code can use ‘self’ to access other attributes or methods of the instance.
        + Ex: The print\_time method uses ‘self.hours’ and ‘self.minutes’ to get the value of time1 instance data attributes.

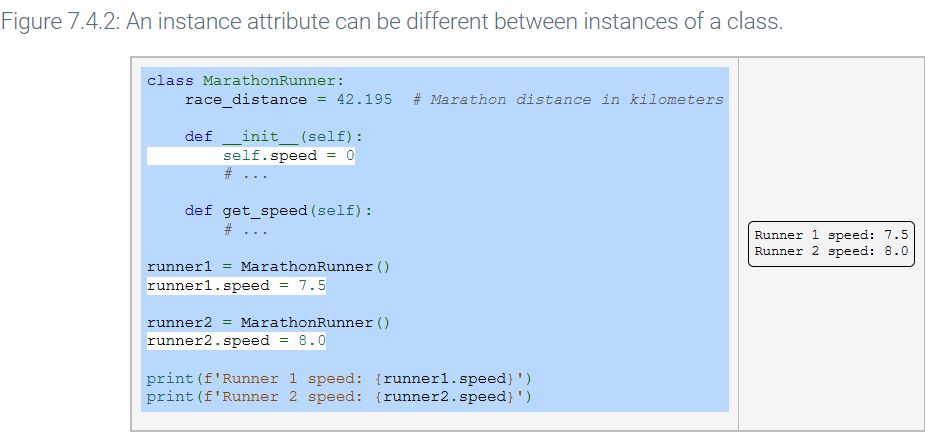


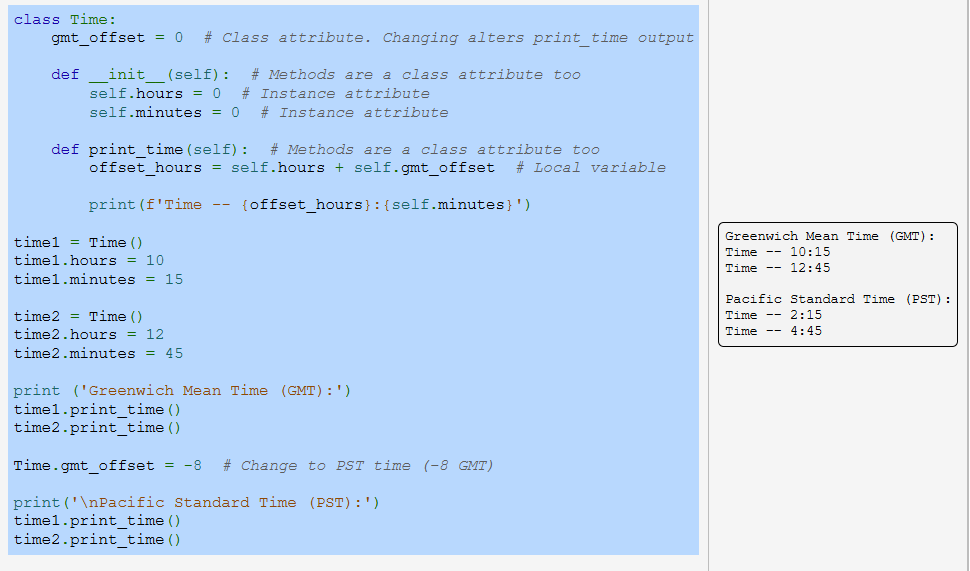


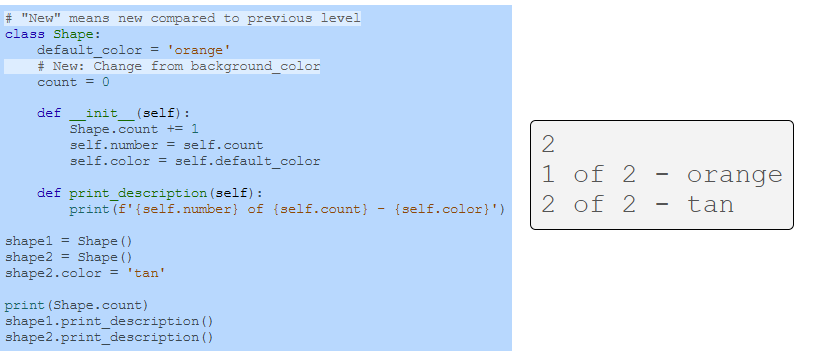
7.4 Class and instance object types

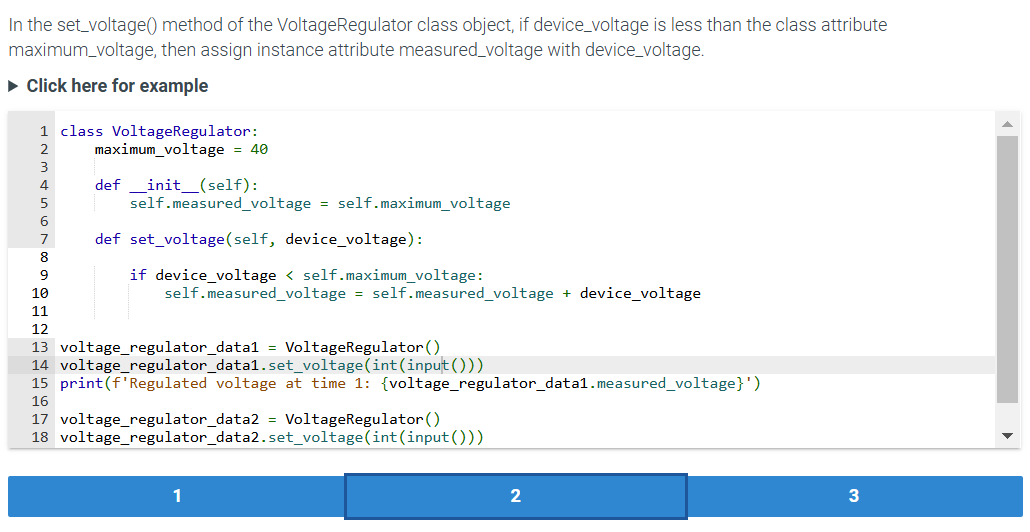
* A programmer with user-defined classes contains two additional types of objects: class objects and instance objects.
  + A **class object** acts as a factory that creates instance objects/
    - When created by the class object, an **instance object** is initialized via the \_\_init\_\_ method. The following tool demonstrates how \_\_init\_\_ method of the Time class object is used to initialize two new Time instance objects.
* A **class attribute** is shared among all instances of that class. Class attributes are defined within the scope of a class

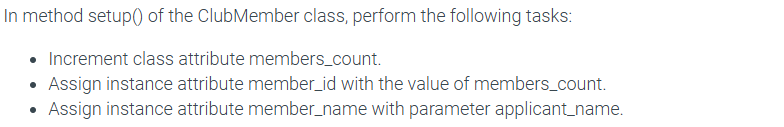


* An **instance attribute** can be unique to each instance
* 









def \_\_init\_\_(self):

self.member\_name = 'empty'

self.member\_id = 0

def setup(self, applicant\_name):

**ClubMember.members\_count = ClubMember.members\_count + 1**

**self.member\_id = ClubMember.members\_count**

**self.member\_name = applicant\_name**

def print\_value(self):

print(f"{self.member\_name}'s member ID is {self.member\_id}.")

for applicant\_name in input().split():

new\_member = ClubMember()

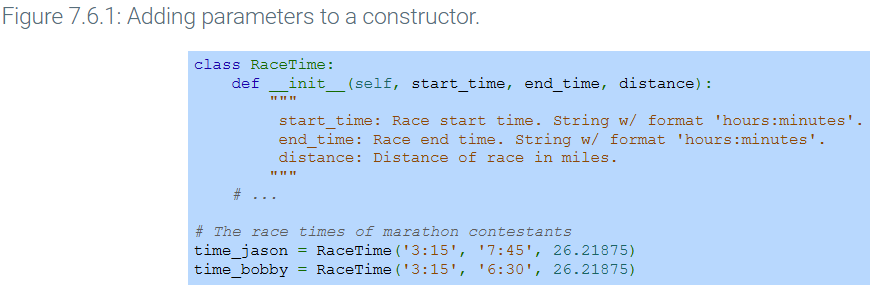
new\_member.setup(applicant\_name)

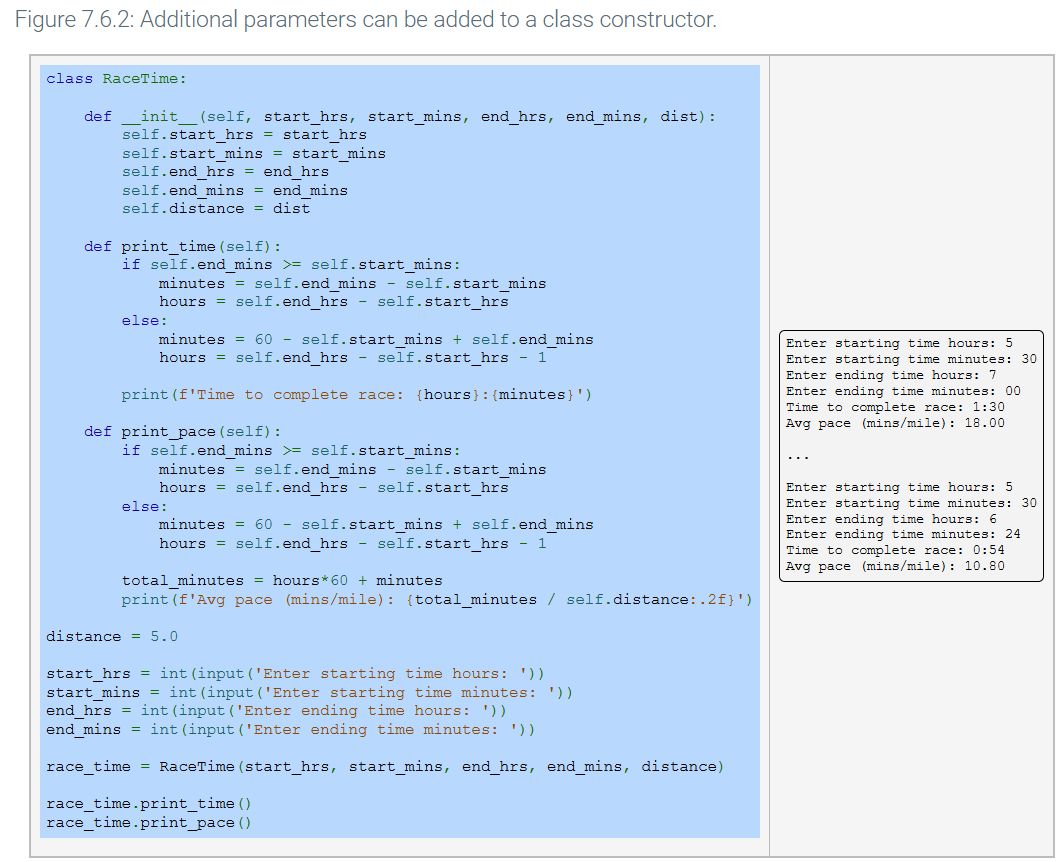
new\_member.print\_value()

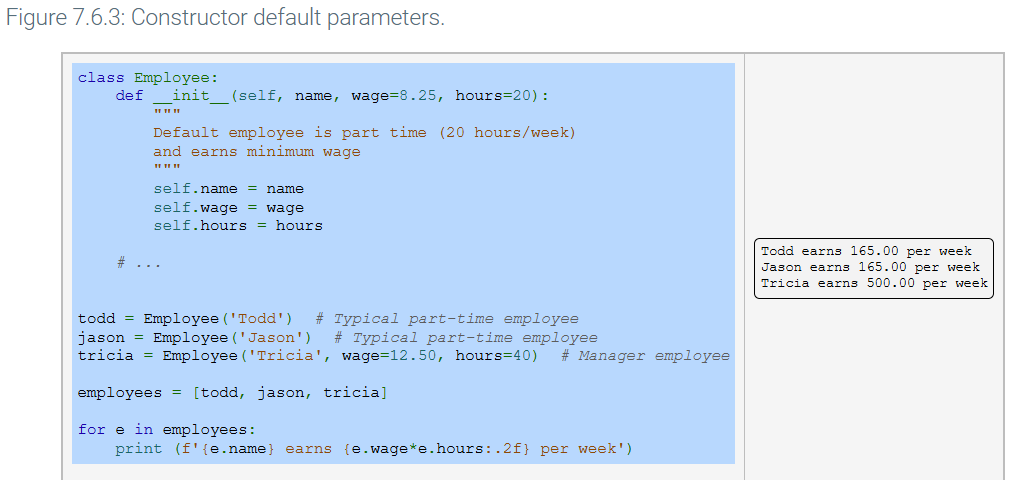
print(f'The club has {ClubMember.members\_count} members.')

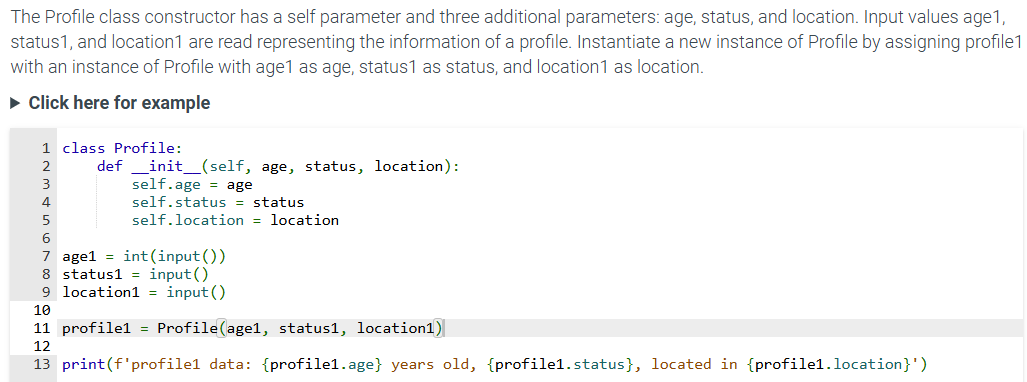
7.6 Class constructors

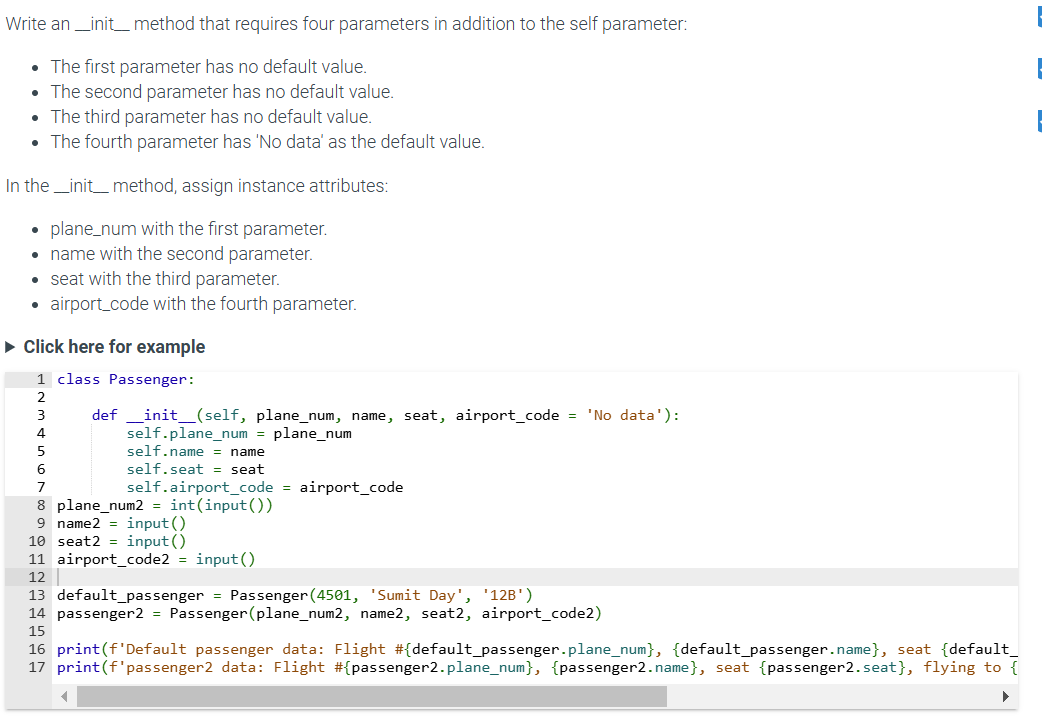
* A class instance is initialized to a specific state.
  + The \_\_init\_\_ method constructor can be customized with additional parameters, as shown below

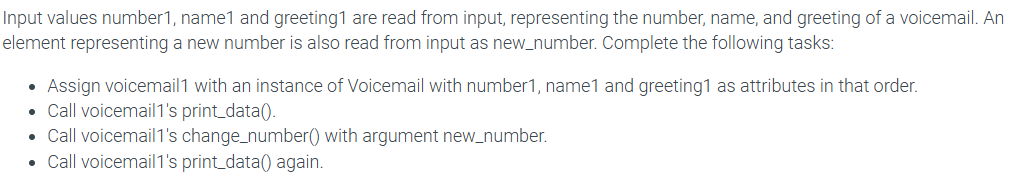












voicemail1 = Voicemail(number1, name1, greeting1)

voicemail1.print\_data()

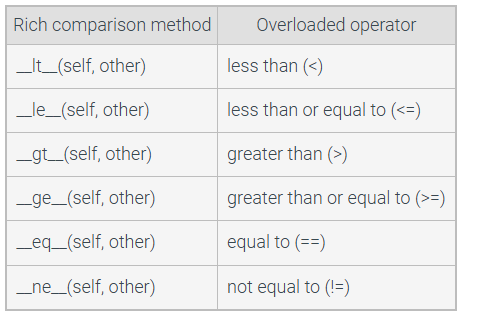
voicemail1.change\_number(new\_number)

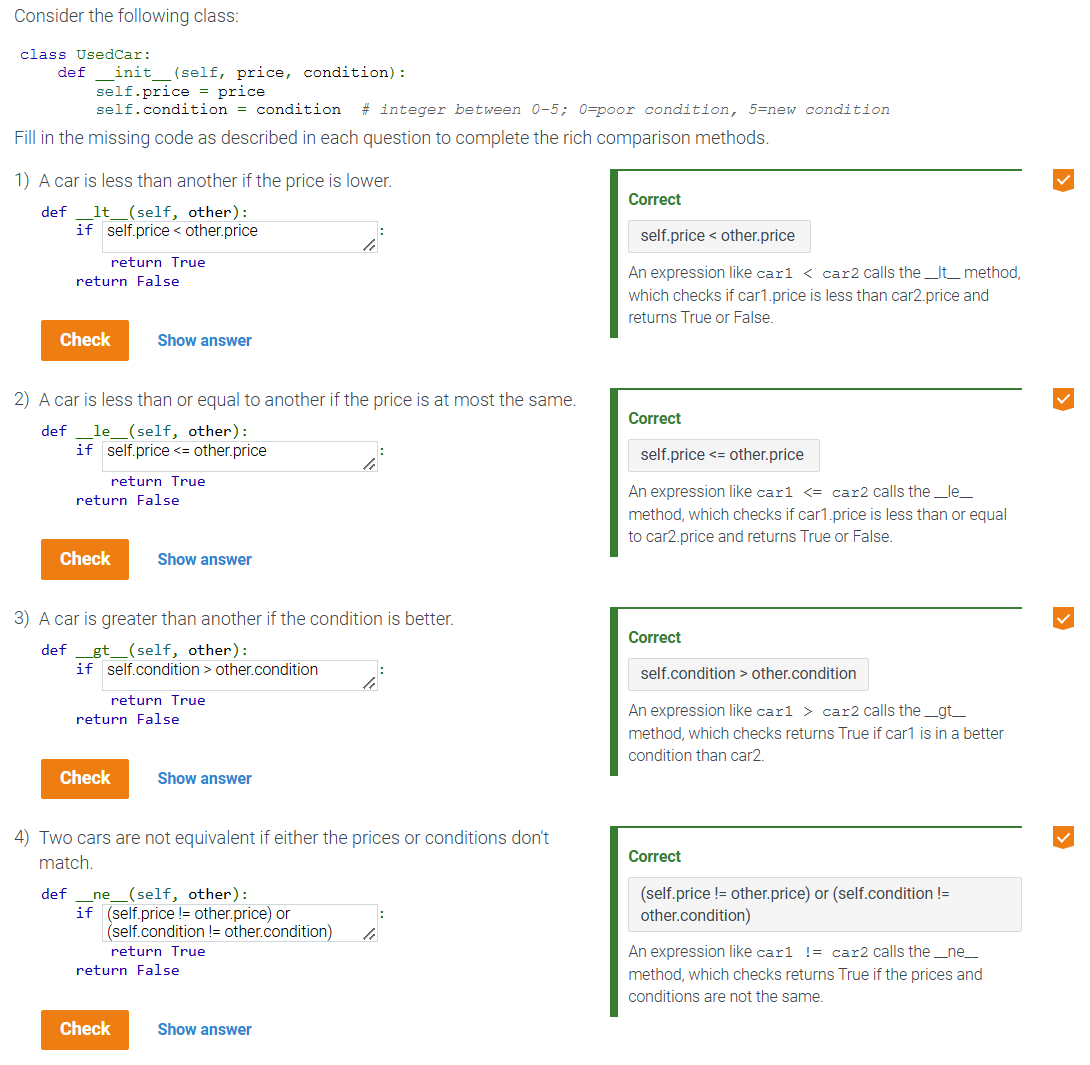
voicemail1.print\_data()

* Good practice is to prepend an underscore to methods only used internally by a class.
  + Ex: time1.\_diff\_time()

7.8

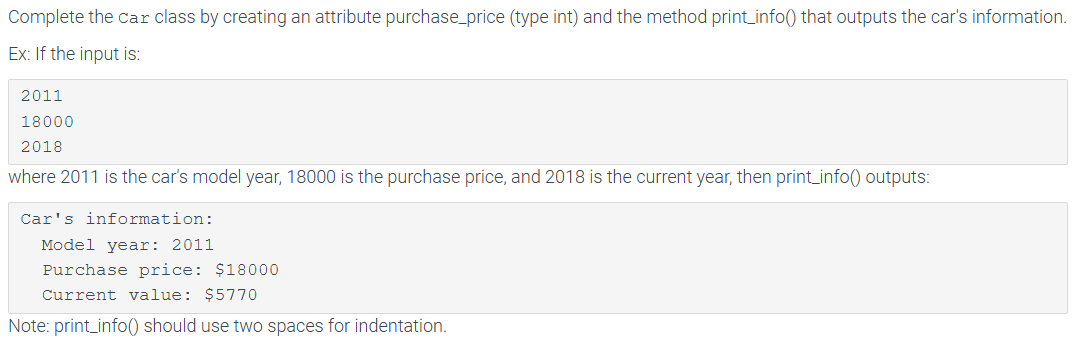
* **Class customization** is the process of defining how an instance of a class should behave for some common operations.







7.11 LAB CAR VALUE (Classes)



class Car:

def \_\_init\_\_(self):

self.model\_year = 0

self.purchase\_price = 0

self.current\_value = 0

def calc\_current\_value(self, current\_year):

depreciation\_rate = 0.15

*# Car depreciation formula*

car\_age = current\_year - self.model\_year

self.current\_value = round(self.purchase\_price \* (1 - depreciation\_rate) \*\* car\_age)

*# TODO: Define print\_info() method to output model\_year, purchase\_price, and current\_value*

def print\_info(self):

print("Car's information:")

print(f' Model year: {self.model\_year}')

print(f' Purchase price: ${self.purchase\_price}')

print(f' Current value: ${self.current\_value}')

if \_\_name\_\_ == "\_\_main\_\_":

year = int(input())

price = int(input())

current\_year = int(input())

my\_car = Car()

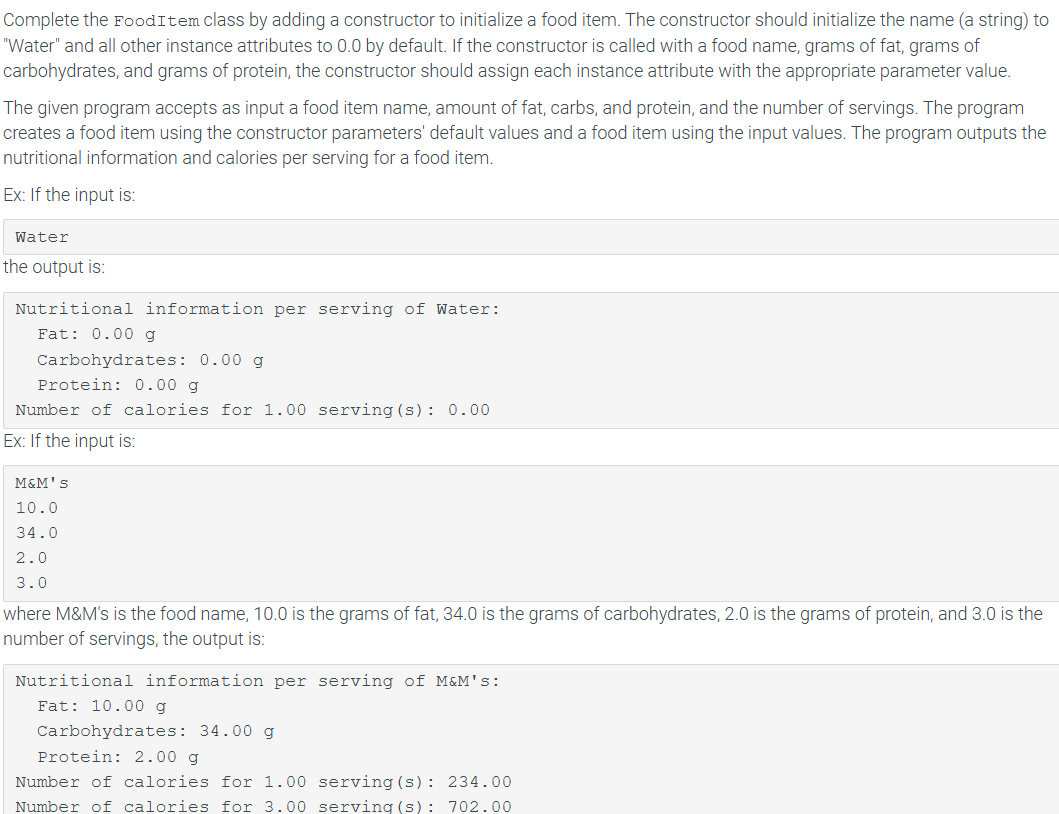
my\_car.model\_year = year

my\_car.purchase\_price = price

my\_car.calc\_current\_value(current\_year)

my\_car.print\_info()

7.12 LAB - Nutritional information (classes/constructions)



class FoodItem:

def \_\_init\_\_(self, name="Water", fat=0.0, carbs=0.0, protein=0.0):

self.name = name

self.fat = fat

self.carbs = carbs

self.protein = protein

def get\_calories(self, num\_servings):

*# Calorie formula*

calories = ((self.fat \* 9) + (self.carbs \* 4) + (self.protein \* 4)) \* num\_servings

return calories

def print\_info(self):

print(f'Nutritional information per serving of {self.name}:')

print(f' Fat: {self.fat:.2f} g')

print(f' Carbohydrates: {self.carbs:.2f} g')

print(f' Protein: {self.protein:.2f} g')

if \_\_name\_\_ == "\_\_main\_\_":

item\_name = input()

if item\_name.lower() == 'water':

food\_item = FoodItem()

food\_item.print\_info()

print(f'Number of calories for {1.0:.2f} serving(s): {food\_item.get\_calories(1.0):.2f}')

else:

amount\_fat = float(input())

amount\_carbs = float(input())

amount\_protein = float(input())

num\_servings = float(input())

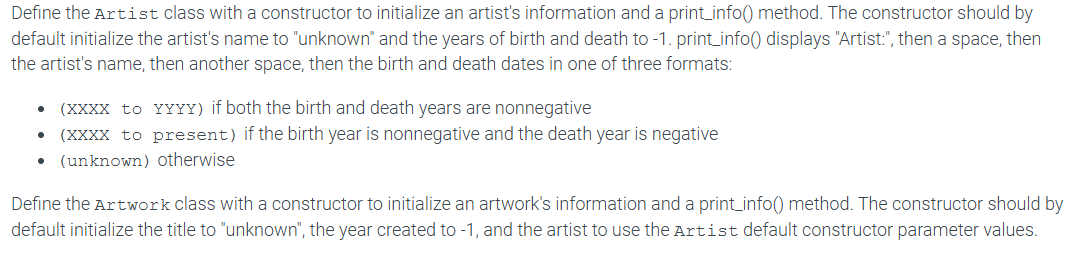
food\_item = FoodItem(item\_name, amount\_fat, amount\_carbs, amount\_protein)

food\_item.print\_info()

print(f'Number of calories for {1.0:.2f} serving(s): {food\_item.get\_calories(1.0):.2f}')

print(f'Number of calories for {num\_servings:.2f} serving(s): {food\_item.get\_calories(num\_servings):.2f}')

7.13 LAB: Artwork label (classes/construction)



class Artist:

def \_\_init\_\_(self, name="unknown", birth\_year=-1, death\_year=-1):

self.name = name

self.birth\_year = birth\_year

self.death\_year = death\_year

def print\_info(self):

if self.birth\_year >= 0 and self.death\_year >= 0:

print(f'Artist: {self.name} ({self.birth\_year} to {self.death\_year})')

elif self.birth\_year >= 0:

print(f'Artist: {self.name} ({self.birth\_year} to present)')

else:

print(f'Artist: {self.name} (unknown)')

class Artwork:

def \_\_init\_\_(self, title="unknown", year\_created=-1, artist=None):

self.title = title

self.year\_created = year\_created

self.artist = artist if artist else Artist()

def print\_info(self):

print(f'Title: {self.title}, {self.year\_created}')

if \_\_name\_\_ == "\_\_main\_\_":

user\_artist\_name = input()

user\_birth\_year = int(input())

user\_death\_year = int(input())

user\_title = input()

user\_year\_created = int(input())

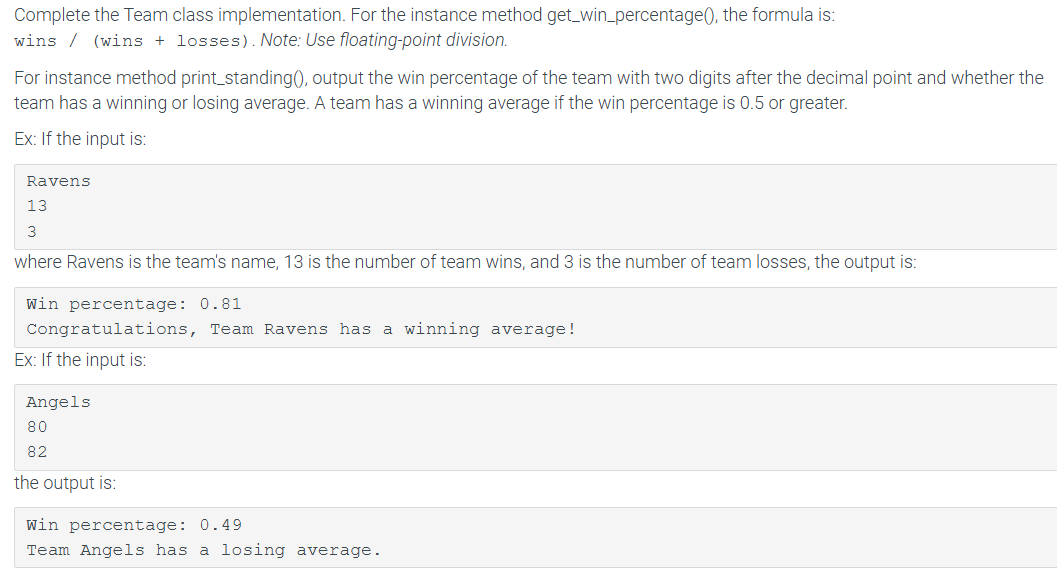
user\_artist = Artist(user\_artist\_name, user\_birth\_year, user\_death\_year)

new\_artwork = Artwork(user\_title, user\_year\_created, user\_artist)

user\_artist.print\_info() *# Print artist info*

new\_artwork.print\_info() *# Print artwork info*

7.15 LAB: Winning teams (classes)



class Team:

def \_\_init\_\_(self):

self.name = 'none'

self.wins = 0

self.losses = 0

def get\_win\_percentage(self):

return(self.wins / (self.wins + self.losses))

def print\_standing(self):

win\_percentage = self.get\_win\_percentage()

print(f"Win percentage: {win\_percentage:.2f}")

if win\_percentage >= 0.5:

print(f"Congratulations, Team {self.name} has a winning average!")

else:

print(f"Team {self.name} has a losing average.")

if \_\_name\_\_ == "\_\_main\_\_":

team = Team()

user\_name = input()

user\_wins = int(input())

user\_losses = int(input())

team.name = user\_name

team.wins = user\_wins

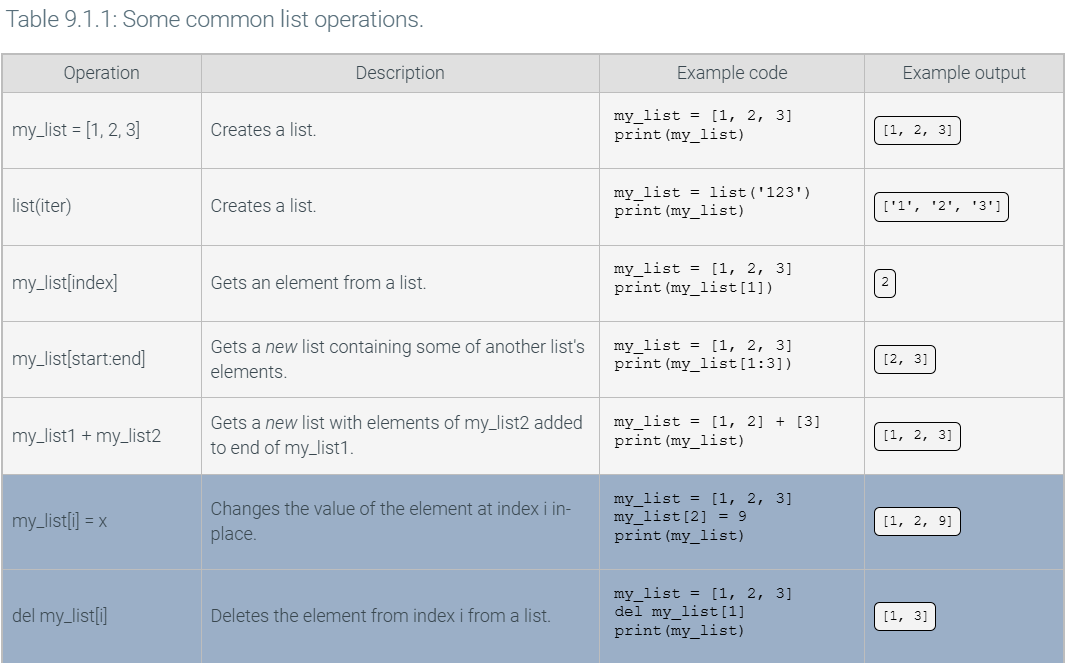
team.losses = user\_losses

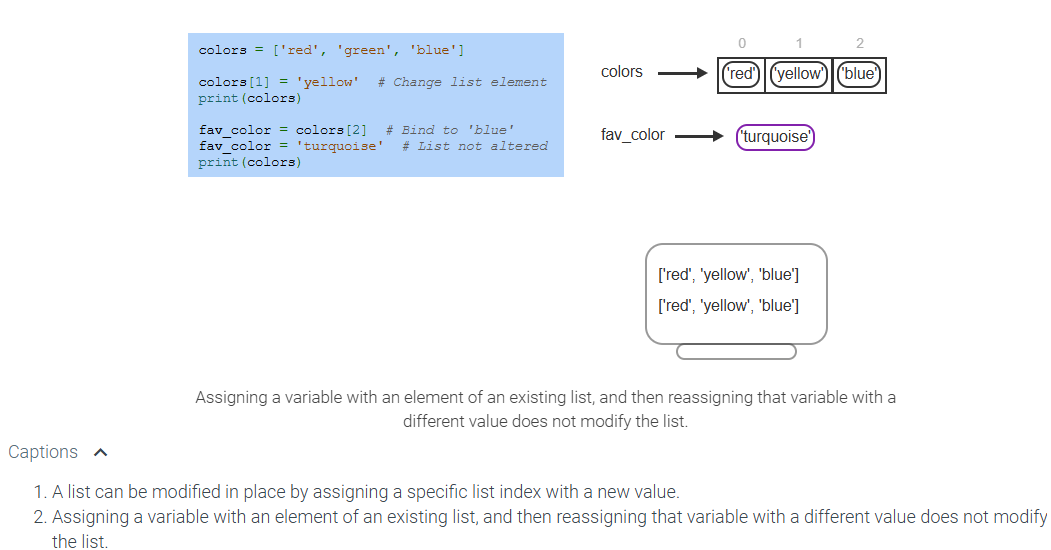
team.print\_standing()

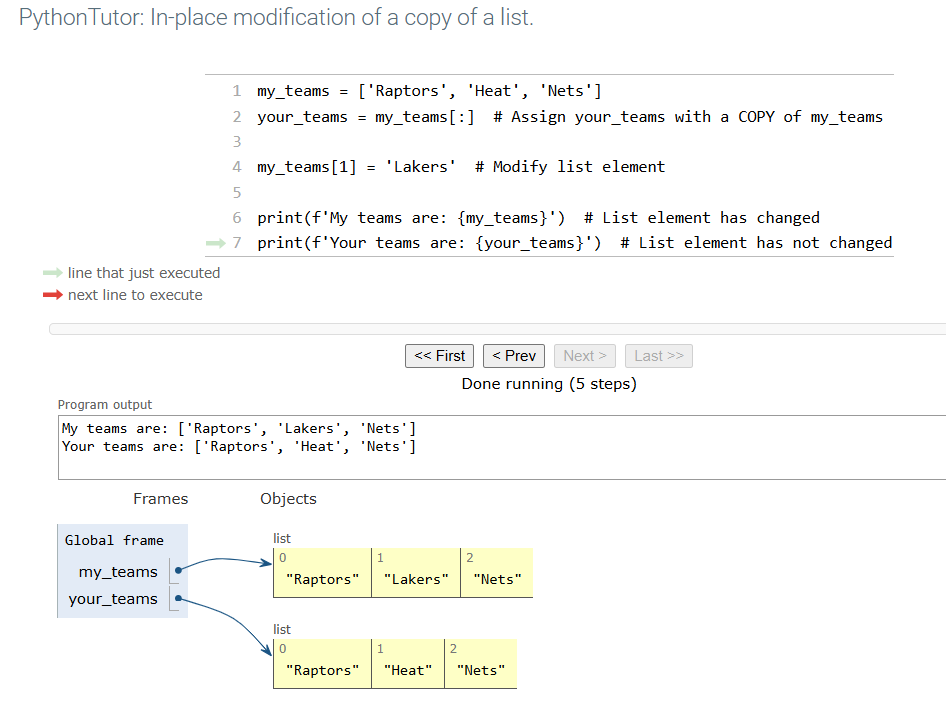
9 - Lists and Dictionaries

9.1 Lists

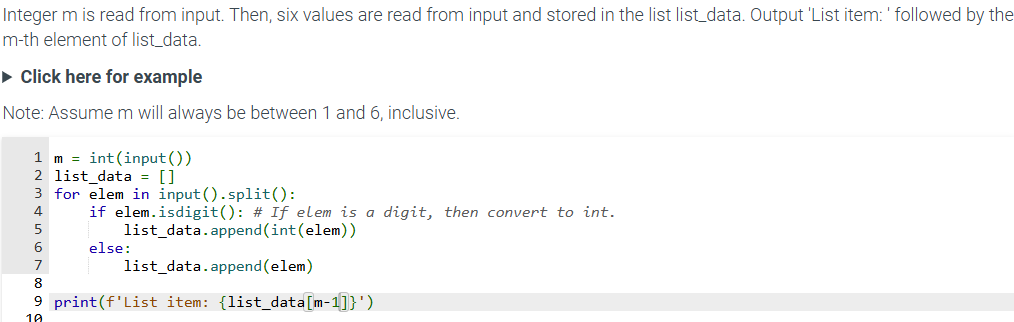
* The list object type is one of the most important and often used types in python
  + Two terms used to define a list are **container** and **mutable**
* A **Container** is an object that groups related objects together
* A **list** is a **mutable** container, meaning the size of the list can grow or shrink and elements within the list can change.
  + A **list** is also a sequence thus, the contained objects maintain a left to right positional order.
* Elements of a list can be accessed via indexing operations that specify the position of the desired element in the list.
* Each element in a list can be a different object type such as strings , integers, floats or even other lists
* Lists are created using brackets [ ]
* Lists can also be created using the built-in list() function
  + The **list()** function accepts a single iterable object argument, such as a string, list, or tuple and returns a new list object
    - Ex: list (‘abc’) creates a new list with the elements [‘a’,’b’,’c’]
* An **index** is a zero-based integer matching to a specific position in a list’s sequence of elements
* A programmer can reference a specific elements in a list by providing an index:
  + Ex: my\_list[4] uses an integer to access the element at index 4 (the 5th element) of my\_list
* An index can also be an expression, as long as that expression evaluates into an integer. Replacing the index with an integer variable, such as in my\_list [i] allows the programmer to quickly and easily look up the (i + 1)th element in a list.



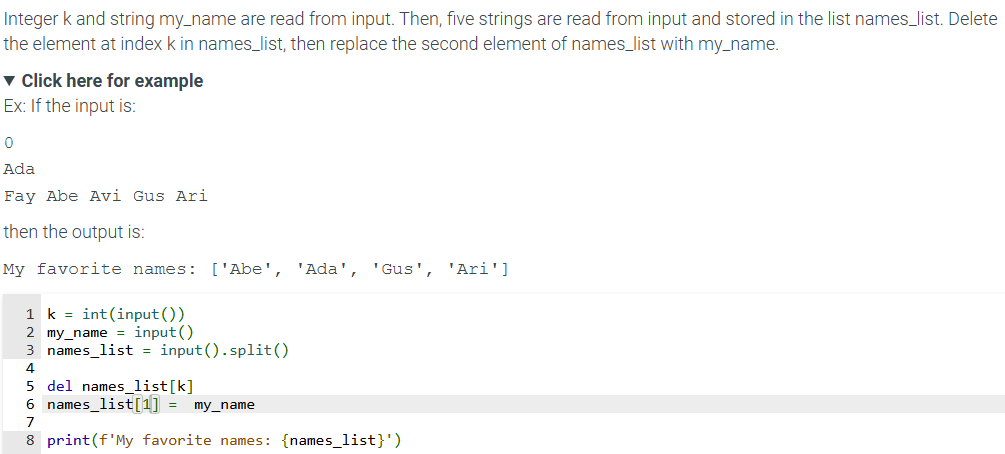




* To create a copy of a list as another list, use slice notation with no start or end indices
  + Your\_team = my\_team [:]



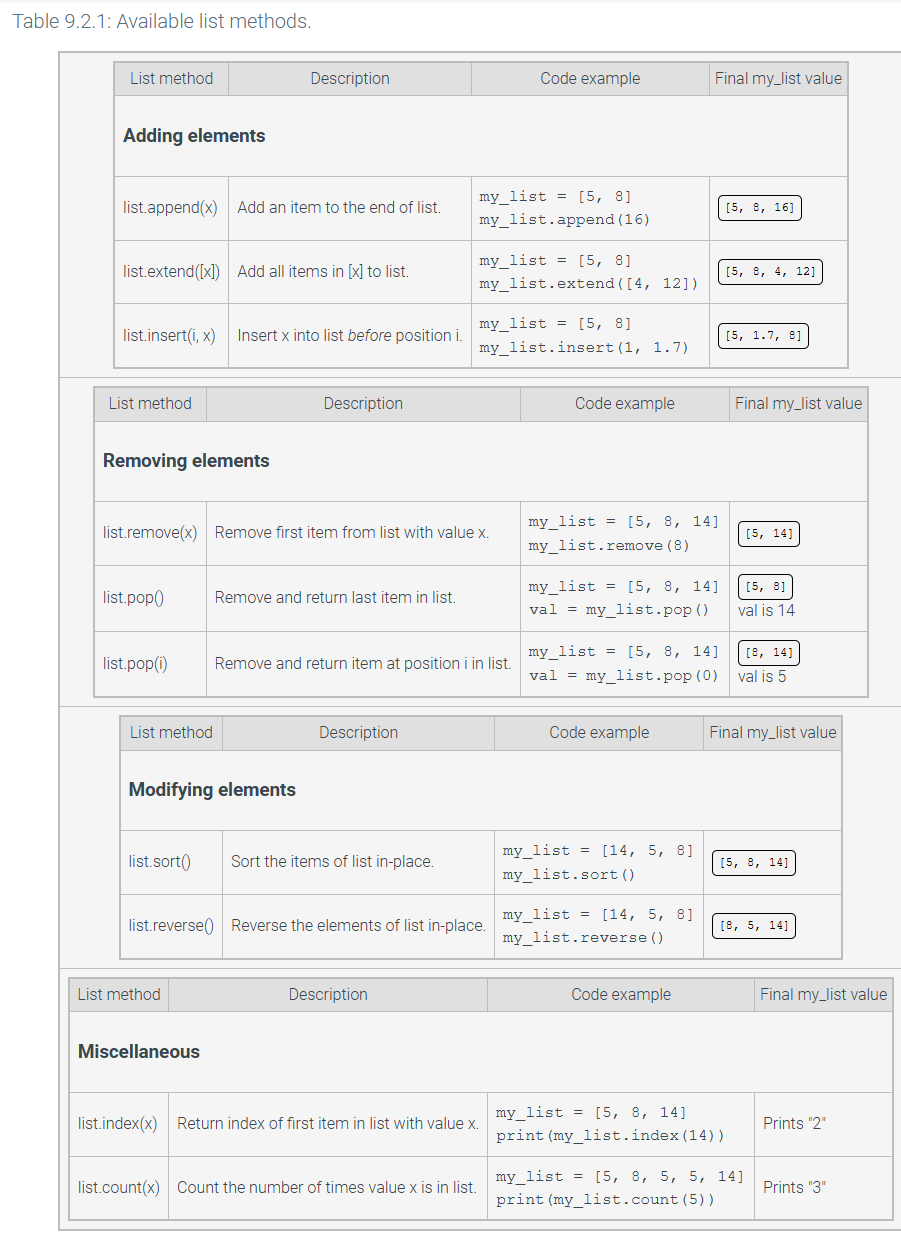






9.2 List methods

* A **list method** can perform a useful operation on a list such as adding or removing elements, sorting, reversing, etc

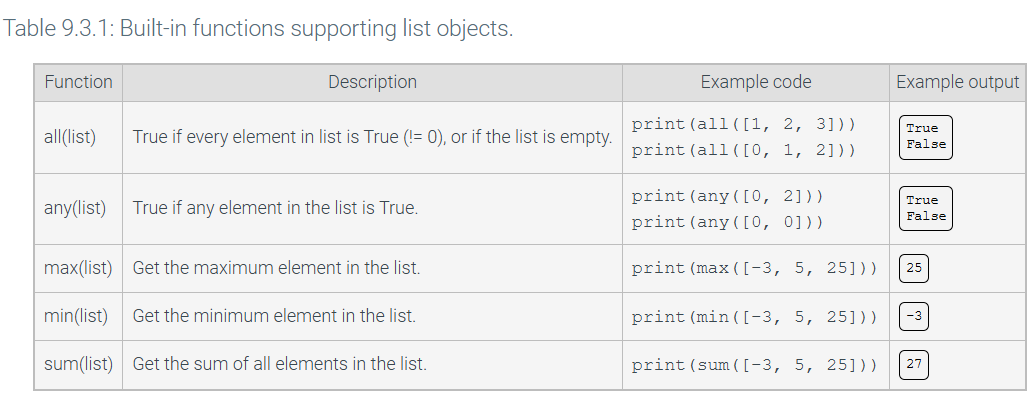


* The **list.sort()** and **list.reverse()** methods rearrange a list elements ordering, performing in place modification of the list
* The **list.index()** and **list.count()** return information about the list and do not modify the list

9.3 Iterating over a list:

* Looping through a sequence such as a list is so common that Python supports a construct called a **for loop**, specifically for iteration purposes.

## **Figure 9.3.1: Iterating through a list.**

* for my\_var in my\_list:
* *# Loop body statements go here*
* The built in **enumerate()** function iterates over a list and provides an iteration counter.
* 

PRACTICE TEST

import pandas as pd

import random as r

*# CURRENT GRADER WORKS ON q1-q6*

*# The goal is to get to 7 points in 40 minutes*

*# Good luck*

def q1(t1, t2):

*# 1 pt*

*# Task: Calculate the Euclidean distance between two 2D points.*

*# Steps:*

*# 1. Extract the x and y coordinates from each tuple `t1` and `t2`.*

x1, y1 = t1

x2, y2 = t2

*# 2. Compute the square of the difference between the x-coordinates and the y-coordinates.*

diff\_x = (x2 - x1) \*\* 2

diff\_y = (y2 - y1) \*\* 2

*# 3. Add these squared differences together.*

diff = diff\_x + diff\_y

*# 4. Take the square root of the sum to get the Euclidean distance.*

sqrt = diff \*\* .5

return sqrt

*# Example:*

*# Input: t1 = (0, 0), t2 = (3, 4)*

*# Output: 5.0 (The Euclidean distance between (0, 0) and (3, 4) is 5)*

pass

def q2(a, b, c):

*# 1 pt*

*# Task: Check if the three integers are all even and if the first integer is the smallest.*

*# Steps:*

*# 1. Check if `a` is smaller than both `b` and `c`.*

smaller = a < b and a < c

*# 2. Verify if `a`, `b`, and `c` are all divisible by 2 (even numbers).*

divisible = (a % 2 == 0) and (b % 2 == 0) and (c % 2 == 0)

*# 3. If both conditions are true, return `True`, otherwise return `False`.*

return smaller and divisible

*# Example:*

*# Input: a = 2, b = 4, c = 6*

*# Output: True (2 is the smallest, and all values are even)*

pass

def q3(list\_of\_integers):

*# 1 pt*

*# Task: Find the maximum value in a list of integers using Pandas.*

*# Steps:*

*# 1. Convert the list of integers into a Pandas `Series`.*

list\_of\_integers\_pandas = pd.Series(list\_of\_integers)

*# 2. Use the `.max()` method to return the largest integer in the series.*

largest\_integer = list\_of\_integers\_pandas.max()

return largest\_integer

*# Example:*

*# Input: list\_of\_integers = [3, 5, 7, 2, 8]*

*# Output: 8 (The largest value in the list is 8)*

pass

def q4(list\_of\_integers\_and\_strings):

*# 2 pts*

*# Task: Find the maximum integer from a mixed list of integers and strings.*

*# Steps:*

*# 1. Loop through each element in the list.*

integers\_only = []

*# 2. Check if the element is an integer (using `type(i) == int`).*

for i in list\_of\_integers\_and\_strings:

if type(i) == int:

integers\_only.append(i)

*# 3. Collect all the integers in a separate list.*

*# 4. Convert the list of integers into a Pandas `Series` and use the `.max()` method to return the largest integer.*

integer\_series = pd.Series(integers\_only)

max\_integer = integer\_series.max()

return max\_integer

*# Example:*

*# Input: list\_of\_integers\_and\_strings = [1, "apple", 7, "banana", 3]*

*# Output: 7 (The largest integer is 7)*

pass

def q5(list\_of\_integers):

*# 3 pts*

*# Input: list of integers*

*# Output: subset of list of integers that are prime numbers*

*# a prime number is an integer greater than 1 that has as its only factors 1 and itself.*

def is\_prime(n):

if n <= 1:

return False

for i in range(2, int(n\*\*.5) + 1):

if n % i == 0:

return False

return True

return [num for num in list\_of\_integers if is\_prime(num)]

*# Example:*

*# Input: list\_of\_integers = [2, 3, 4, 5, 6 , 5]*

*# Output: [2, 3, 5, 5]*

*# NOTE REPEATED NUMBERS IN INPUT ARE REPEATED IN OUTPUT*

pass

def q6(df, column):

*# 3 pts*

*# Task: Check if a specified column in a Pandas DataFrame contains any missing (`NaN`) values.*

*# Steps:*

*# 1. Use the `.isna()` method to check for missing values in the specified column.*

missing\_values = df[column].isna()

*# 2. Use `.sum()` to count how many `NaN` values exist.*

missing\_count = missing\_values.sum()

*# 3. If the count is greater than 0, return `True`; otherwise, return `False`.*

if missing\_count > 0:

return True

else:

return False

*# Example:*

*# Input: df = pd.DataFrame({'A': [1, 2, None, 4]}), column = 'A'*

*# Output: True (The column 'A' has a NaN value)*

pass

class A:

*# THE GRADER FOR THIS ISNT FINISHED*

def \_\_init\_\_(self, a=None, b=None, c=None):

*# 0 pts*

*# Task: Initialize an object with attributes `a`, `b`, and `c`.*

*# If any attribute is not provided, assign a random integer value between 1 and 10.*

*# Steps:*

*# 1. If any attribute is None, assign a random integer between 1 and 10.*

if a is not None:

self.a = a

else:

self.a = random.randint(1,10)

if b is not None:

self.b = b

else:

self.b = random.randint(1,10)

if c is not None:

self.c = c

else:

self.c = random.randint(1,10)

*# Example:*

*# Input: a = None, b = None, c = None (random values assigned)*

*# Output: a random instance with a, b, c values between 1 and 10.*

pass

def f1(self, other):

*# 1 pt*

*# Task: Compare two instances of `A` to see if the current instance's values of `a`, `b`, and `c` are all greater than the other's values.*

*# Steps:*

*# 1. Compare `self.a`, `self.b`, and `self.c` with `other.a`, `other.b`, and `other.c`.*

*# 2. Return `True` if all of `self.a > other.a`, `self.b > other.b`, and `self.c > other.c`, otherwise return `False`.*

if (self.a > other.a) and (self.b > other.b) and (self.c > other.c):

return True

else:

return False

*# Example:*

*# Input: self = A(5, 6, 7), other = A(3, 4, 6)*

*# Output: True (5 > 3, 6 > 4, and 7 > 6)*

pass

def f2(self, other):

*# 2 pts*

*# Assume self.a is the maximum of a random number and other.a is the maximum of another random number*

*# Calculate the probability that the random number from self.a is greater than the random number from other.a*

*# Task: Compute a ratio of how often the values in `self.a` are greater than the corresponding values in `other.a` over all combinations.*

pass

def f3(self, other):

*# 3 pts*

*# Task: Calculate the correlation between the attributes of `self` and `other`.*

*# Steps:*

*# 1. Create two lists, one for the attributes of `self` and one for `other`.*

attributes\_self = [self.a, self.b, self.c]

attributes\_other = [other.a, other.b, other.c]

df = pd.DataFrame({

'self': attribute\_self,

'other': attribute\_other

})

*# 3. Calculate the correlation matrix using and return the mean of the correlation values.*

correlation\_matrix = df.corr()

mean\_correlation = correlation\_matrix.mean().mean()

return mean\_correlation

*# Example:*

*# Input: self = A(1, 2, 3), other = A(4, 5, 6)*

*# Output: correlation value (ranges from -1 to 1)*

pass

def f4(self, others):

*# 2 pts*

*# Task: Create a CSV file that contains the `a`, `b`, and `c` attributes of `self` and other instances.*

*# Steps:*

*# 1. Add `self` to the list of `others`.*

*# 2. Create a list of lists with the `a`, `b`, and `c` attributes of all instances.*

*# 3. Convert this list into a DataFrame and save it as a CSV file named 'output.csv'.*

*# Example:*

*# Input: self = A(5, 6, 7), others = [A(2, 3, 4), A(8, 9, 10)]*

*# Output: A CSV file 'f4\_output.csv' with the values:*

*# 5, 6, 7*

*# 2, 3, 4*

*# 8, 9, 10*

*# ! BE SURE TO NAME THE FILE f4\_output.csv*

pass

def f5(self, other):

*# 2 pts*

*# Task: Compare `self.a`, `self.b`, and `self.c` with `other.a`, `other.b`, and `other.c` and return a vector showing the differences.*

*# Steps:*

*# 1. Initialize `x\_vector`, `y\_vector`, and `z\_vector` as 0.*

*# 2. Compare `self.a` with `other.a`, `self.b` with `other.b`, and `self.c` with `other.c`.*

*# 3. Set the vectors to 1 or -1 depending on whether `self.a` is smaller or larger than `other.a`, respectively.*

*# 4. Return the tuple `(x\_vector, y\_vector, z\_vector)` indicating the direction of difference.*

*# Example:*

*# Input: self = A(3, 6, 9), other = A(4, 3, 7)*

*# Output: (-1, 1, 1) (Because 3 < 4, 6 > 3, and 9 > 7)*

pass

if \_\_name\_\_ == "\_\_main\_\_":

*# THE SECTION BELOW IS USED TO TEST YOUR CODE IT IS NOT DIRECTLY GRADED.*

*# Testing q1 (distance between two points)*

point1 = (2, 3)

point2 = (5, 7)

print(f"q1: Distance between {point1} and {point2} is {q1(point1, point2)}")

*# Testing q2 (condition for the three values)*

a, b, c = 2, 4, 6

print(f"q2: Is ({a}, {b}, {c}) a valid even triplet? {q2(a, b, c)}")

*# Testing q3 (max value in a list)*

list\_of\_integers = [1, 3, 7, 2, 5]

print(f"q3: Maximum value in {list\_of\_integers} is {q3(list\_of\_integers)}")

*# Testing q4 (max value in list with integers and strings)*

mixed\_list = [1, 'a', 2, 'b', 3, 4, 'c']

print(f"q4: Maximum integer in {mixed\_list} is {q4(mixed\_list)}")

*# Testing q5 (values greater than 2 and divisible check)*

list\_of\_integers = [1, 2, 3, 4, 5, 6]

print(f"q5: Modified list: {q5(list\_of\_integers)}")

*# Testing q6 (check if column has missing values)*

data = {'A': [1, 2, 3], 'B': [None, 5, 6]}

df = pd.DataFrame(data)

print(f"q6: Does column 'B' have missing values? {q6(df, 'B')}")

*# Testing class A*

obj1 = A(5, 3, 8)

obj2 = A(6, 7, 9)

obj3 = A()

*# f1: Test if obj1 is "greater" than obj2*

print(f"f1: Does obj1 > obj2? {obj1.f1(obj2)}")

*# f2: Ratio of comparisons for obj1 vs obj2*

print(f"f2: Ratio of comparisons for obj1 vs obj2: {obj1.f2(obj2)}")

*# f3: Correlation between obj1 and obj2 attributes*

print(f"f3: Correlation between obj1 and obj2: {obj1.f3(obj2)}")

*# f4: Save object list data to CSV (creates 'output.csv')*

obj1.f4([obj2, obj3])

*# f5: Vector comparison between obj1 and obj2*

print(f"f5: Vector comparison between obj1 and obj2: {obj1.f5(obj2)}")

import pandas as pd

import random as r

*# CURRENT GRADER WORKS ON q1-q6*

*# The goal is to get to 7 points in 40 minutes*

*# Good luck*

def q1(t1, t2):

*# 1 pt*

*# Task: Calculate the Euclidean distance between two 2D points.*

*# Steps:*

*# 1. Extract the x and y coordinates from each tuple `t1` and `t2`.*

x1, y1 = t1

x2 , y2 = t2

*# 2. Compute the square of the difference between the x-coordinates and the y-coordinates.*

x\_diff = (x2 - x1) \*\* 2

y\_diff = (y2 - y1) \*\* 2

*# 3. Add these squared differences together.*

diff\_sum = x\_diff + y\_diff

*# 4. Take the square root of the sum to get the Euclidean distance.*

sqrt = diff\_sum \*\* .5

return sqrt

*# Example:*

*# Input: t1 = (0, 0), t2 = (3, 4)*

*# Output: 5.0 (The Euclidean distance between (0, 0) and (3, 4) is 5)*

pass

def q2(a, b, c):

*# 1 pt*

*# Task: Check if the three integers are all even and if the first integer is the smallest.*

*# Steps:*

*# 1. Check if `a` is smaller than both `b` and `c`.*

smaller = (a < b) and (a < c)

*# 2. Verify if `a`, `b`, and `c` are all divisible by 2 (even numbers).*

even\_numbers = (a % 2 == 0) and (b % 2 == 0) and (c % 2 == 0)

*# 3. If both conditions are true, return `True`, otherwise return `False`.*

return smaller and even\_numbers

*# Example:*

*# Input: a = 2, b = 4, c = 6*

*# Output: True (2 is the smallest, and all values are even)*

pass

def q3(list\_of\_integers):

*# 1 pt*

*# Task: Find the maximum value in a list of integers using Pandas.*

*# Steps:*

*# 1. Convert the list of integers into a Pandas `Series`.*

pandas\_series = pd.Series(list\_of\_integers)

*# 2. Use the `.max()` method to return the largest integer in the series.*

largest\_integer = pandas\_series.max()

return largest\_integer

*# Example:*

*# Input: list\_of\_integers = [3, 5, 7, 2, 8]*

*# Output: 8 (The largest value in the list is 8)*

pass

def q4(list\_of\_integers\_and\_strings):

*# 2 pts*

*# Task: Find the maximum integer from a mixed list of integers and strings.*

*# Steps:*

*# 1. Loop through each element in the list.*

my\_list = []

*# 2. Check if the element is an integer (using `type(i) == int`).*

for i in list\_of\_integers\_and\_strings:

if type(i) == int:

my\_list.append(i)

*# 3. Collect all the integers in a separate list.*

*# 4. Convert the list of integers into a Pandas `Series` and use the `.max()` method to return the largest integer.*

pand\_series = pd.Series(my\_list)

max\_num = pand\_series.max()

return max\_num

*# Example:*

*# Input: list\_of\_integers\_and\_strings = [1, "apple", 7, "banana", 3]*

*# Output: 7 (The largest integer is 7)*

pass

def q5(list\_of\_integers):

*# 3 pts*

*# Input: list of integers*

*# Output: subset of list of integers that are prime numbers*

*# a prime number is an integer greater than 1 that has as its only factors 1 and itself.*

def is\_prime(num):

if num <= 1:

return False

for i in range(2, int(num\*\*0.5) + 1):

if num % i == 0:

return False

return True

primes = []

for num in list\_of\_integers:

if is\_prime(num):

primes.append(num)

return primes

*# Example:*

*# Input: list\_of\_integers = [2, 3, 4, 5, 6 , 5]*

*# Output: [2, 3, 5, 5]*

*# NOTE REPEATED NUMBERS IN INPUT ARE REPEATED IN OUTPUT*

pass

def q6(df, column):

*# 3 pts*

*# Task: Check if a specified column in a Pandas DataFrame contains any missing (`NaN`) values.*

*# Steps:*

*# 1. Use the `.isna()` method to check for missing values in the specified column.*

missing\_values = df[column].isna()

*# 2. Use `.sum()` to count how many `NaN` values exist.*

sum\_missing\_values = missing\_values.sum()

*# 3. If the count is greater than 0, return `True`; otherwise, return `False`.*

if sum\_missing\_values > 0:

return True

else:

return False

*# Example:*

*# Input: df = pd.DataFrame({'A': [1, 2, None, 4]}), column = 'A'*

*# Output: True (The column 'A' has a NaN value)*

pass

class A:

*# THE GRADER FOR THIS ISNT FINISHED*

def \_\_init\_\_(self, a=None, b=None, c=None):

*# 0 pts*

*# Task: Initialize an object with attributes `a`, `b`, and `c`.*

*# If any attribute is not provided, assign a random integer value between 1 and 10.*

*# Steps:*

*# 1. If any attribute is None, assign a random integer between 1 and 10.*

if a is None:

self.a = r.randint(1,10)

else:

self.a = a

if b is None:

self.b = r.randint(1,10)

else:

self.b = b

if c is None:

self.c = r.randint(1,10)

else:

self.c = c

*# Example:*

*# Input: a = None, b = None, c = None (random values assigned)*

*# Output: a random instance with a, b, c values between 1 and 10.*

pass

def f1(self, other):

*# 1 pt*

*# Task: Compare two instances of `A` to see if the current instance's values of `a`, `b`, and `c` are all greater than the other's values.*

*# Steps:*

*# 1. Compare `self.a`, `self.b`, and `self.c` with `other.a`, `other.b`, and `other.c`.*

if (self.a > other.a) and (self.b > other.b) and (self.c > other.c):

return True

else:

return False

*# 2. Return `True` if all of `self.a > other.a`, `self.b > other.b`, and `self.c > other.c`, otherwise return `False`.*

*# Example:*

*# Input: self = A(5, 6, 7), other = A(3, 4, 6)*

*# Output: True (5 > 3, 6 > 4, and 7 > 6)*

pass

def f2(self, other):

*# 2 pts*

*# Assume self.a is the maximum of a random number and other.a is the maximum of another random number*

*# Calculate the probability that the random number from self.a is greater than the random number from other.a*

*# Task: Compute a ratio of how often the values in `self.a` are greater than the corresponding values in `other.a` over all combinations.*

self\_values = self.a if isinstance(self.a, (list, tuple)) else [self.a]

other\_values = other.a if isinstance(other.a, (list, tuple)) else [other.a]

total\_comparisons = 0

self\_greater\_count = 0

for self\_val in self\_values:

for other\_val in other\_values:

total\_comparisons += 1

if self\_val > other\_val:

self\_greater\_count += 1

probability = self\_greater\_count / total\_comparisons if total\_comparisons > 0 else 0

return probability

pass

def f3(self, other):

*# 3 pts*

*# Task: Calculate the correlation between the attributes of `self` and `other`.*

*# Steps:*

*# 1. Create two lists, one for the attributes of `self` and one for `other`.*

list\_attributes\_self = [self.a, self.b, self.c]

list\_attributes\_other = [other.a, other.b, other.c]

*# 2. Create a DataFrame from these lists.*

df = pd.DataFrame()

df['self'] = list\_attributes\_self

df['other'] = list\_attributes\_other

*# 3. Calculate the correlation matrix using and return the mean of the correlation values.*

correlation\_matrix = df.corr()

mean\_correlation = correlation\_matrix.values.mean()

return mean\_correlation

*# Example:*

*# Input: self = A(1, 2, 3), other = A(4, 5, 6)*

*# Output: correlation value (ranges from -1 to 1)*

pass

def f4(self, others):

*# 2 pts*

*# Task: Create a CSV file that contains the `a`, `b`, and `c` attributes of `self` and other instances.*

*# Steps:*

*# 1. Add `self` to the list of `others`.*

all\_instances = [self] + others

*# 2. Create a list of lists with the `a`, `b`, and `c` attributes of all instances.*

all\_instances\_list = []

for i in all\_instances:

attributes = [i.a, i.b, i.c]

all\_instances\_list.append(attributes)

*# 3. Convert this list into a DataFrame and save it as a CSV file named 'output.csv'.*

df = pd.DataFrame(all\_instances\_list, columns=['a', 'b', 'c'])

df.to\_csv('f4\_output.csv', index=False)

*# Example:*

*# Input: self = A(5, 6, 7), others = [A(2, 3, 4), A(8, 9, 10)]*

*# Output: A CSV file 'f4\_output.csv' with the values:*

*# 5, 6, 7*

*# 2, 3, 4*

*# 8, 9, 10*

*# ! BE SURE TO NAME THE FILE f4\_output.csv*

pass

def f5(self, other):

*# 2 pts*

*# Task: Compare `self.a`, `self.b`, and `self.c` with `other.a`, `other.b`, and `other.c` and return a vector showing the differences.*

*# Steps:*

*# 1. Initialize `x\_vector`, `y\_vector`, and `z\_vector` as 0.*

x\_vector = 0

y\_vector = 0

z\_vector = 0

*# 2. Compare `self.a` with `other.a`, `self.b` with `other.b`, and `self.c` with `other.c`.*

if self.a > other.a:

x\_vector = 1

elif self.a < other.a:

x\_vector = -1

if self.b > other.b:

y\_vector = 1

elif self.b < other.b:

y\_vector = -1

if self.c > other.c:

z\_vector = 1

elif self.c < other.c:

z\_vector = -1

*# 3. Set the vectors to 1 or -1 depending on whether `self.a` is smaller or larger than `other.a`, respectively.*

*# 4. Return the tuple `(x\_vector, y\_vector, z\_vector)` indicating the direction of difference.*

return (x\_vector, y\_vector, z\_vector)

*# Example:*

*# Input: self = A(3, 6, 9), other = A(4, 3, 7)*

*# Output: (-1, 1, 1) (Because 3 < 4, 6 > 3, and 9 > 7)*

pass

if \_\_name\_\_ == "\_\_main\_\_":

*# THE SECTION BELOW IS USED TO TEST YOUR CODE IT IS NOT DIRECTLY GRADED.*

*# Testing q1 (distance between two points)*

point1 = (2, 3)

point2 = (5, 7)

print(f"q1: Distance between {point1} and {point2} is {q1(point1, point2)}")

*# Testing q2 (condition for the three values)*

a, b, c = 2, 4, 6

print(f"q2: Is ({a}, {b}, {c}) a valid even triplet? {q2(a, b, c)}")

*# Testing q3 (max value in a list)*

list\_of\_integers = [1, 3, 7, 2, 5]

print(f"q3: Maximum value in {list\_of\_integers} is {q3(list\_of\_integers)}")

*# Testing q4 (max value in list with integers and strings)*

mixed\_list = [1, 'a', 2, 'b', 3, 4, 'c']

print(f"q4: Maximum integer in {mixed\_list} is {q4(mixed\_list)}")

*# Testing q5 (values greater than 2 and divisible check)*

list\_of\_integers = [1, 2, 3, 4, 5, 6]

print(f"q5: Modified list: {q5(list\_of\_integers)}")

*# Testing q6 (check if column has missing values)*

data = {'A': [1, 2, 3], 'B': [None, 5, 6]}

df = pd.DataFrame(data)

print(f"q6: Does column 'B' have missing values? {q6(df, 'B')}")

*# Testing class A*

obj1 = A(5, 3, 8)

obj2 = A(6, 7, 9)

obj3 = A()

*# f1: Test if obj1 is "greater" than obj2*

print(f"f1: Does obj1 > obj2? {obj1.f1(obj2)}")

*# f2: Ratio of comparisons for obj1 vs obj2*

print(f"f2: Ratio of comparisons for obj1 vs obj2: {obj1.f2(obj2)}")

*# f3: Correlation between obj1 and obj2 attributes*

print(f"f3: Correlation between obj1 and obj2: {obj1.f3(obj2)}")

*# f4: Save object list data to CSV (creates 'output.csv')*

obj1.f4([obj2, obj3])

*# f5: Vector comparison between obj1 and obj2*

print(f"f5: Vector comparison between obj1 and obj2: {obj1.f5(obj2)}")

def q1(t1, t2, t3):

# 1 pt

# Task: Calculate the area of a triangle given its three vertices in a 2D space.

# Steps:

# 1. Extract the x and y coordinates from each vertex (t1, t2, t3).

x1, y1 = t1

x2, y2 = t2

x3, y3 = t3

# 2. Use the formula to calculate the area of a triangle:

# Area = 0.5 \* abs(x1\*(y2-y3) + x2\*(y3-y1) + x3\*(y1-y2)).

area = 0.5 \* abs(x1 \* (y2 - y3) + x2 \* (y3 - y1) + x3 \* (y1 - y2))

# 3. Return the calculated area.

return area

# Example:

# Input: t1 = (0, 0), t2 = (4, 0), t3 = (0, 3)

# Output: 6.0 (The area of the triangle is 6)

def q2(s):

# 1 pt

# Task: Check if a given string is a palindrome (reads the same forwards and backwards).

# Steps:

# 1. Remove all non-alphanumeric characters and convert the string to lowercase.

cleaned = ''

for char in s:

if char.isalnum():

cleaned += char.lower()

# 2. Compare the cleaned string to its reverse to check if it is a palindrome.

is\_palindrome = cleaned == cleaned[::-1]

# 3. Return True if it is a palindrome, otherwise return False.

return is\_palindrome

# Example:

# Input: s = "A man, a plan, a canal, Panama"

# Output: True (The string is a palindrome)

def q3(s):

# 1 pt

# Task: Count the number of vowels in a given string.

# Steps:

# 1. Define the vowels (a, e, i, o, u).

vowels = 'aeiou'

count = 0

# 2. Loop through each character in the string and check if it is a vowel.

for char in s.lower():

if char in vowels:

count += 1

# 3. Return the total count of vowels.

return count

# Example:

# Input: s = "hello world"

# Output: 3 (The vowels are 'e', 'o', and 'o')

def q4(list1, list2):

# 2 pts

# Task: Find the common elements between two lists.

# Steps:

# 1. Create an empty list to store the common elements.

common = []

# 2. Loop through each element in the first list.

for item in list1:

# 3. If the item is also in the second list, add it to the common list.

if item in list2:

common.append(item)

# 4. Return the list of common elements.

return list(set(common))

# Example:

# Input: list1 = [1, 2, 3, 4], list2 = [3, 4, 5, 6]

# Output: [3, 4] (The common elements between the lists)

def q5(words, n):

# 2 pts

# Task: Filter a list of words to include only those with length greater than `n`.

# Steps:

# 1. Create an empty list to store the filtered words.

filtered = []

# 2. Loop through each word in the list.

for word in words:

# 3. Check if the word's length is greater than `n`.

if len(word) > n:

filtered.append(word)

# 4. Return the list of filtered words.

return filtered

# Example:

# Input: words = ["apple", "banana", "cherry", "date"], n = 5

# Output: ["banana", "cherry"] (Words with length greater than 5)

def q6(data):

# 3 pts

# Task: Create a Pandas DataFrame from a list of dictionaries.

# Steps:

# 1. Import the `pandas` library.

import pandas as pd

# 2. Use `pd.DataFrame()` to create a DataFrame from the list of dictionaries.

df = pd.DataFrame(data)

# 3. Return the resulting DataFrame.

return df

# Example:

# Input: data = [{"Name": "Alice", "Age": 25}, {"Name": "Bob", "Age": 30}]

# Output: A DataFrame with columns 'Name' and 'Age' and corresponding rows.

def q7(df):

# 3 pts

# Task: Check if any column in a Pandas DataFrame contains missing (`NaN`) values.

# Steps:

# 1. Create an empty dictionary to store the results.

result = {}

# 2. Loop through each column in the DataFrame.

for column in df.columns:

# 3. Use the `.isna()` method to check for missing values.

result[column] = df[column].isna().any()

# 4. Return the dictionary with column names as keys and `True` or `False` as values.

return result

# Example:

# Input: df = pd.DataFrame({"A": [1, None, 3], "B": [4, 5, 6]})

# Output: {"A": True, "B": False} (Column A has missing values, B does not)

def q8(n):

# 2 pts

# Task: Generate the first `n` numbers in the Fibonacci sequence.

# Steps:

# 1. Create a list with the first two numbers of the sequence (0 and 1).

fib = [0, 1]

# 2. Use a loop to calculate the remaining numbers.

for \_ in range(2, n):

next\_number = fib[-1] + fib[-2]

fib.append(next\_number)

# 3. Return the first `n` numbers from the list.

return fib[:n]

# Example:

# Input: n = 6

# Output: [0, 1, 1, 2, 3, 5] (The first 6 Fibonacci numbers)

def q9(numbers):

# 2 pts

# Task: Find the second largest number in a list of integers.

# Steps:

# 1. Remove duplicate numbers by converting the list to a set.

unique\_numbers = list(set(numbers))

# 2. Sort the unique numbers in descending order.

unique\_numbers.sort(reverse=True)

# 3. Return the second largest number.

if len(unique\_numbers) > 1:

return unique\_numbers[1]

else:

return None

# Example:

# Input: numbers = [4, 2, 6, 8, 6]

# Output: 6 (The second largest unique number)

def q10(str1, str2):

# 3 pts

# Task: Check if two strings are anagrams of each other (contain the same characters in any order).

# Steps:

# 1. Remove all non-alphanumeric characters and convert both strings to lowercase.

clean\_str1 = ''.join(c.lower() for c in str1 if c.isalnum())

clean\_str2 = ''.join(c.lower() for c in str2 if c.isalnum())

# 2. Sort the characters in both strings.

sorted\_str1 = sorted(clean\_str1)

sorted\_str2 = sorted(clean\_str2)

# 3. Compare the sorted strings and return `True` if they are equal, otherwise `False`.

return sorted\_str1 == sorted\_str2

# Example:

# Input: str1 = "listen", str2 = "silent"

# Output: True (The strings are anagrams)

def q1(t1, t2, t3):

# 1 pt

# Task: Calculate the area of a triangle given its three vertices in a 2D space.

# Steps:

# 1. Extract the x and y coordinates from each vertex (t1, t2, t3).

x1, y1 = t1

x2, y2 = t2

x3, y3 = t3

# 2. Use the formula to calculate the area of a triangle:

# Area = 0.5 \* abs(x1\*(y2-y3) + x2\*(y3-y1) + x3\*(y1-y2)).

area = 0.5 \* abs(x1 \* (y2 - y3) + x2 \* (y3 - y1) + x3 \* (y1 - y2))

# 3. Return the calculated area.

return area

# Example:

# Input: t1 = (0, 0), t2 = (4, 0), t3 = (0, 3)

# Output: 6.0 (The area of the triangle is 6)

def q2(s):

# 1 pt

# Task: Check if a given string is a palindrome (reads the same forwards and backwards).

# Steps:

# 1. Remove all non-alphanumeric characters and convert the string to lowercase.

cleaned = ''

for char in s:

if char.isalnum():

cleaned += char.lower()

# 2. Compare the cleaned string to its reverse to check if it is a palindrome.

is\_palindrome = cleaned == cleaned[::-1]

# 3. Return True if it is a palindrome, otherwise return False.

return is\_palindrome

# Example:

# Input: s = "A man, a plan, a canal, Panama"

# Output: True (The string is a palindrome)

def q3(s):

# 1 pt

# Task: Count the number of vowels in a given string.

# Steps:

# 1. Define the vowels (a, e, i, o, u).

vowels = 'aeiou'

count = 0

# 2. Loop through each character in the string and check if it is a vowel.

for char in s.lower():

if char in vowels:

count += 1

# 3. Return the total count of vowels.

return count

# Example:

# Input: s = "hello world"

# Output: 3 (The vowels are 'e', 'o', and 'o')

def q4(list1, list2):

# 2 pts

# Task: Find the common elements between two lists.

# Steps:

# 1. Create an empty list to store the common elements.

common = []

# 2. Loop through each element in the first list.

for item in list1:

# 3. If the item is also in the second list, add it to the common list.

if item in list2:

common.append(item)

# 4. Return the list of common elements.

return list(set(common))

# Example:

# Input: list1 = [1, 2, 3, 4], list2 = [3, 4, 5, 6]

# Output: [3, 4] (The common elements between the lists)

def q5(words, n):

# 2 pts

# Task: Filter a list of words to include only those with length greater than `n`.

# Steps:

# 1. Create an empty list to store the filtered words.

filtered = []

# 2. Loop through each word in the list.

for word in words:

# 3. Check if the word's length is greater than `n`.

if len(word) > n:

filtered.append(word)

# 4. Return the list of filtered words.

return filtered

# Example:

# Input: words = ["apple", "banana", "cherry", "date"], n = 5

# Output: ["banana", "cherry"] (Words with length greater than 5)

def q6(data):

# 3 pts

# Task: Create a Pandas DataFrame from a list of dictionaries.

# Steps:

# 1. Import the `pandas` library.

import pandas as pd

# 2. Use `pd.DataFrame()` to create a DataFrame from the list of dictionaries.

df = pd.DataFrame(data)

# 3. Return the resulting DataFrame.

return df

# Example:

# Input: data = [{"Name": "Alice", "Age": 25}, {"Name": "Bob", "Age": 30}]

# Output: A DataFrame with columns 'Name' and 'Age' and corresponding rows.

def q7(df):

# 3 pts

# Task: Check if any column in a Pandas DataFrame contains missing (`NaN`) values.

# Steps:

# 1. Create an empty dictionary to store the results.

result = {}

# 2. Loop through each column in the DataFrame.

for column in df.columns:

# 3. Use the `.isna()` method to check for missing values.

result[column] = df[column].isna().any()

# 4. Return the dictionary with column names as keys and `True` or `False` as values.

return result

# Example:

# Input: df = pd.DataFrame({"A": [1, None, 3], "B": [4, 5, 6]})

# Output: {"A": True, "B": False} (Column A has missing values, B does not)

def q8(n):

# 2 pts

# Task: Generate the first `n` numbers in the Fibonacci sequence.

# Steps:

# 1. Create a list with the first two numbers of the sequence (0 and 1).

fib = [0, 1]

# 2. Use a loop to calculate the remaining numbers.

for \_ in range(2, n):

next\_number = fib[-1] + fib[-2]

fib.append(next\_number)

# 3. Return the first `n` numbers from the list.

return fib[:n]

# Example:

# Input: n = 6

# Output: [0, 1, 1, 2, 3, 5] (The first 6 Fibonacci numbers)

def q9(numbers):

# 2 pts

# Task: Find the second largest number in a list of integers.

# Steps:

# 1. Remove duplicate numbers by converting the list to a set.

unique\_numbers = list(set(numbers))

# 2. Sort the unique numbers in descending order.

unique\_numbers.sort(reverse=True)

# 3. Return the second largest number.

if len(unique\_numbers) > 1:

return unique\_numbers[1]

else:

return None

# Example:

# Input: numbers = [4, 2, 6, 8, 6]

# Output: 6 (The second largest unique number)

def q10(str1, str2):

# 3 pts

# Task: Check if two strings are anagrams of each other (contain the same characters in any order).

# Steps:

# 1. Remove all non-alphanumeric characters and convert both strings to lowercase.

clean\_str1 = ''.join(c.lower() for c in str1 if c.isalnum())

clean\_str2 = ''.join(c.lower() for c in str2 if c.isalnum())

# 2. Sort the characters in both strings.

sorted\_str1 = sorted(clean\_str1)

sorted\_str2 = sorted(clean\_str2)

# 3. Compare the sorted strings and return `True` if they are equal, otherwise `False`.

return sorted\_str1 == sorted\_str2

# Example:

# Input: str1 = "listen", str2 = "silent"

# Output: True (The strings are anagrams)

import pandas as pd

import math

# CURRENT GRADER WORKS ON q1-q9

# The goal is to get to 7 points in 40 minutes

# Good luck

def q1(r):

# 2 pts

# Task: Calculate the area of a circle given its radius.

# Steps:

# 1. Import `pi` from the math module.

# 2. Use the formula: area = pi \* r^2.

# 3. Return the area.

area = math.pi \* (r \*\* 2)

return round(area, 2)

# Example:

# Input: r = 3

# Output: 28.27 (approximately)

pass

def q2(a, b):

# 2 pts

# Task: Safely divide two numbers and handle division by zero.

# Steps:

# 1. Use a try-except block to handle ZeroDivisionError.

# 2. If `b` is zero, return "Division by zero is not allowed."

# 3. Otherwise, return the result of a / b.

try:

result = a / b

except ZeroDivisionError:

return "Division by zero is not allowed"

return result

# Example:

# Input: a = 10, b = 0

# Output: "Division by zero is not allowed"

pass

def q3(r):

# 2 pts

# Task: Calculate the volume of a sphere given its radius.

# Steps:

# 1. Use the formula: V = (4/3) \* pi \* r^3.

# 2. Return the volume rounded to two decimal places.

volume = (4 / 3) \* math.pi \* (r \*\* 3)

return round(volume, 2)

# Example:

# Input: r = 2

# Output: 33.51

pass

def q4(a, b, c):

# 1 pt

# Task: Check if all three numbers are greater than 10.

# Steps:

# 1. Use an if statement to check if a > 10, b > 10, and c > 10.

# 2. Return True if all conditions are met, otherwise return False.

return a > 10 and b > 10 and c > 10

# Example:

# Input: a = 15, b = 20, c = 8

# Output: False

pass

def q5(list\_of\_integers):

# 2 pts

# Task: Filter out even numbers from a list.

# Steps:

# 1. Use a list comprehension to iterate through the list.

# 2. Return a new list with only even numbers.

return [num for num in list\_of\_integers if num % 2 == 0]

# Example:

# Input: [1, 2, 3, 4, 5]

# Output: [2, 4]

pass

def q6(input\_string):

# 1 pt

# Task: Remove leading and trailing whitespace from a string.

# Steps:

# 1. Use the `.strip()` method.

# 2. Return the cleaned string.

return input\_string.strip()

# Example:

# Input: " hello world "

# Output: "hello world"

pass

def q7(df, col1, col2):

# 3 pts

# Task: Check if two columns in a DataFrame are equal.

# Steps:

# 1. Use `.eq()` to compare the two columns.

# 2. Return True if they are equal, otherwise return False.

return df[col1].eq(df[col2]).all()

# Example:

# Input: df = pd.DataFrame({'A': [1, 2], 'B': [1, 2]}), col1 = 'A', col2 = 'B'

# Output: True

pass

def q8(list1, list2):

# 3 pts

# Task: Check if two lists have the same length.

# Steps:

# 1. Use len() to compare the lengths of the lists.

# 2. Return True if they are equal, otherwise return False.

return len(list1) == len(list2)

# Example:

# Input: list1 = [1, 2, 3], list2 = [4, 5]

# Output: False

pass

def q9(r, h):

# 2 pts

# Task: Calculate the volume of a cylinder given its radius and height.

# Steps:

# 1. Use the formula: V = pi \* r^2 \* h.

# 2. Return the volume rounded to two decimal places.

volume = math.pi \* (r \*\* 2) \* h

return round(volume, 2)

# Example:

# Input: r = 2, h = 5

# Output: 62.83

pass

import pandas as pd

# CURRENT GRADER WORKS ON q1-q9

# The goal is to get to 7 points in 40 minutes

# Good luck

def q1(point\_list):

# 2 pts

# Task: Calculate the Euclidean distance between multiple 2D points.

# Steps:

# 1. Accept a list of tuples, each representing a 2D point (x, y).

# 2. Calculate the Euclidean distance between consecutive points.

# 3. Return a list of distances rounded to two decimal places.

distances = []

for i in range(len(point\_list) - 1):

x1, y1 = point\_list[i]

x2, y2 = point\_list[i + 1]

distance = ((x2 - x1) \*\* 2 + (y2 - y1) \*\* 2) \*\* 0.5

distances.append(round(distance, 2))

return distances

# Example:

# Input: [(0, 0), (3, 4), (6, 8)]

# Output: [5.0, 5.0]

pass

def q2(a, b, operation):

# 2 pts

# Task: Perform arithmetic operations with error handling.

# Steps:

# 1. Accept two numbers and a string specifying an operation ('add', 'subtract', 'multiply', 'divide').

# 2. Use if-elif conditions to perform the appropriate operation.

# 3. Handle ZeroDivisionError when the operation is 'divide' and b is 0.

# 4. Return the result of the operation or an appropriate error message.

try:

if operation == 'add':

return a + b

elif operation == 'subtract':

return a - b

elif operation == 'multiply':

return a \* b

elif operation == 'divide':

return a / b

else:

return "Invalid operation"

except ZeroDivisionError:

return "Division by zero is not allowed"

# Example:

# Input: a = 10, b = 0, operation = 'divide'

# Output: "Division by zero is not allowed"

pass

def q3(data\_list):

# 2 pts

# Task: Find the mode of a list of integers.

# Steps:

# 1. Use a dictionary to count the occurrences of each number in the list.

# 2. Find the number(s) with the highest frequency.

# 3. Return the mode as a list (to handle ties).

counts = {}

for num in data\_list:

counts[num] = counts.get(num, 0) + 1

max\_count = max(counts.values())

modes = [key for key, value in counts.items() if value == max\_count]

return modes

# Example:

# Input: [1, 2, 2, 3, 3, 4]

# Output: [2, 3]

pass

def q4(df, col1, col2):

# 3 pts

# Task: Check if two columns in a DataFrame are inversely correlated.

# Steps:

# 1. Calculate the correlation coefficient between the two columns using `.corr()`.

# 2. Check if the coefficient is less than -0.5.

# 3. Return True if they are inversely correlated, otherwise return False.

correlation = df[col1].corr(df[col2])

return correlation < -0.5

# Example:

# Input: df = pd.DataFrame({'A': [1, 2, 3], 'B': [3, 2, 1]}), col1 = 'A', col2 = 'B'

# Output: True

pass

def q5(words\_list):

# 3 pts

# Task: Find the longest word in a list of strings.

# Steps:

# 1. Iterate through the list and find the word with the maximum length.

# 2. Handle the case where there are ties by returning all longest words as a list.

max\_length = max(len(word) for word in words\_list)

longest\_words = [word for word in words\_list if len(word) == max\_length]

return longest\_words

# Example:

# Input: ["cat", "elephant", "dog", "dinosaur"]

# Output: ["elephant", "dinosaur"]

pass

def q6(df, column):

# 3 pts

# Task: Replace missing values in a DataFrame column with the column mean.

# Steps:

# 1. Calculate the mean of the column (ignoring NaN values).

# 2. Replace all NaN values in the column with the calculated mean.

# 3. Return the modified DataFrame.

mean\_value = df[column].mean()

df[column].fillna(mean\_value, inplace=True)

return df

# Example:

# Input: df = pd.DataFrame({'A': [1, None, 3]}), column = 'A'

# Output: pd.DataFrame({'A': [1, 2.0, 3]})

pass

def q7(list1, list2):

# 2 pts

# Task: Find the intersection of two lists (common elements).

# Steps:

# 1. Use set operations to find elements present in both lists.

# 2. Return the intersection as a list.

return list(set(list1) & set(list2))

# Example:

# Input: list1 = [1, 2, 3], list2 = [2, 3, 4]

# Output: [2, 3]

pass

def q8(r, h, n):

# 3 pts

# Task: Calculate the volume of `n` cylinders and return the total volume.

# Steps:

# 1. Use the formula: V = 3.14159 \* r^2 \* h for each cylinder.

# 2. Multiply the result by `n` to get the total volume.

# 3. Return the total volume rounded to two decimal places.

pi = 3.14159

volume = pi \* (r \*\* 2) \* h \* n

return round(volume, 2)

# Example:

# Input: r = 2, h = 5, n = 3

# Output: 188.5

pass

def q9(sentence):

# 3 pts

# Task: Count the number of vowels in a string (case insensitive).

# Steps:

# 1. Define a set of vowels ('a', 'e', 'i', 'o', 'u').

# 2. Iterate through the string and count the vowels.

# 3. Return the count.

vowels = set('aeiouAEIOU')

count = sum(1 for char in sentence if char in vowels)

return count

# Example:

# Input: "Hello World"

# Output: 3

pass

44.9 Rock, paper scissors

import random as r

# Constants for game choices, results, and states

ROCK = 1

PAPER = 2

SCISSORS = 3

INVALID = 99

CONTINUE = 1

QUIT = 2

PLAYER\_WINS = 4

PLAYER\_LOSES = 5

TIE = 6

def computer\_choose():

"""

Randomly select a move for the computer.

"""

return r.choice([ROCK, PAPER, SCISSORS])

def player\_choose(automated, choice=None):

"""

Allow the player to choose a move or use automated mode for testing.

"""

if automated:

if choice is None:

return computer\_choose()

return choice

else:

try:

player\_input = int(input("Choose your move (1 for ROCK, 2 for PAPER, 3 for SCISSORS): "))

if player\_input in [ROCK, PAPER, SCISSORS]:

return player\_input

else:

return INVALID

except ValueError:

return INVALID

def determine\_winner(computer\_choice, player\_choice):

"""

Determine the winner based on the rules of Rock-Paper-Scissors.

"""

choices\_map = {ROCK: "ROCK", PAPER: "PAPER", SCISSORS: "SCISSORS"}

print(f"Player selected {choices\_map.get(player\_choice, 'INVALID')} and computer selected {choices\_map.get(computer\_choice, 'INVALID')}.")

if player\_choice == computer\_choice:

return TIE

elif (player\_choice == ROCK and computer\_choice == SCISSORS) or \

(player\_choice == PAPER and computer\_choice == ROCK) or \

(player\_choice == SCISSORS and computer\_choice == PAPER):

return PLAYER\_WINS

else:

return PLAYER\_LOSES

def check\_for\_another\_game(automated, answer=1):

"""

Ask if the player wants to continue playing or quit.

"""

if automated:

return answer

else:

try:

player\_input = int(input("Continue (1), or Quit (2)? "))

if player\_input in [CONTINUE, QUIT]:

return player\_input

else:

return INVALID

except ValueError:

return INVALID

def play\_game(automated, max\_wins=3):

"""

Main function to run the game loop.

"""

games = 0

player\_wins = 0

computer\_wins = 0

while player\_wins < max\_wins and computer\_wins < max\_wins:

computer\_choice = computer\_choose()

player\_choice = player\_choose(automated)

if player\_choice == INVALID:

print("Invalid choice. Please try again.")

continue

result = determine\_winner(computer\_choice, player\_choice)

games += 1

if result == PLAYER\_WINS:

print("Player wins this round!")

player\_wins += 1

elif result == PLAYER\_LOSES:

print("Computer wins this round!")

computer\_wins += 1

else:

print("It's a tie!")

print(f"Score: Player {player\_wins}, Computer {computer\_wins}, Games Played: {games}")

if player\_wins < max\_wins and computer\_wins < max\_wins:

continue\_game = check\_for\_another\_game(automated)

if continue\_game == QUIT:

print("Game ended by player.")

break

elif continue\_game == INVALID:

print("Invalid input. Ending the game.")

break

print("Game Over!")

if player\_wins > computer\_wins:

print("Player is the overall winner!")

elif computer\_wins > player\_wins:

print("Computer is the overall winner!")

else:

print("The game ends in a tie!")

if \_\_name\_\_ == "\_\_main\_\_":

"""

Start the game in manual mode if executed directly.

"""

play\_game(automated=not EXAMPLE)

import pandas as pd

import random as r

# The goal is to get to 7 points in 40 minutes

# Good luck

def q1(t1, t2):

# 1 pt

# Task: Calculate the Manhattan distance between two 2D points.

# Steps:

# 1. Extract the x and y coordinates from each tuple `t1` and `t2`.

x1, y1 = t1

x2, y2 = t2

# 2. Compute the absolute difference between the x-coordinates and the y-coordinates.

x\_diff = abs(x2 - x1)

y\_diff = abs(y2 - y1)

# 3. Add these absolute differences together to get the Manhattan distance.

manhattan\_distance = x\_diff + y\_diff

return manhattan\_distance

# Example:

# Input: t1 = (2, 3), t2 = (5, 7)

# Output: 7

pass

def q2(a, b, c):

# 1 pt

# Task: Check if all three integers are positive and if the second integer is the largest.

# Steps:

# 1. Verify if `a`, `b`, and `c` are all greater than 0.

positive\_numbers = (a > 0) and (b > 0) and (c > 0)

# 2. Check if `b` is greater than both `a` and `c`.

largest = (b > a) and (b > c)

# 3. If both conditions are true, return `True`, otherwise return `False`.

return positive\_numbers and largest

# Example:

# Input: a = 2, b = 5, c = 3

# Output: True

pass

def q3(list\_of\_numbers):

# 1 pt

# Task: Calculate the sum of all even numbers in a list using Pandas.

# Steps:

# 1. Convert the list of numbers into a Pandas `Series`.

pandas\_series = pd.Series(list\_of\_numbers)

# 2. Use boolean indexing to filter even numbers and compute their sum.

even\_sum = pandas\_series[pandas\_series % 2 == 0].sum()

return even\_sum

# Example:

# Input: list\_of\_numbers = [1, 2, 3, 4, 5, 6]

# Output: 12

pass

def q4(list\_of\_mixed\_values):

# 2 pts

# Task: Find the minimum integer from a mixed list of integers and strings.

# Steps:

# 1. Loop through each element in the list.

integers\_only = []

# 2. Check if the element is an integer (using `type(i) == int`).

for i in list\_of\_mixed\_values:

if type(i) == int:

integers\_only.append(i)

# 3. Convert the integers into a Pandas `Series` and use the `.min()` method to find the smallest integer.

min\_value = pd.Series(integers\_only).min()

return min\_value

# Example:

# Input: list\_of\_mixed\_values = [10, "apple", 3, "banana", 5]

# Output: 3

pass

def q5(list\_of\_numbers):

# 3 pts

# Task: Return a list of perfect squares from the input list of integers.

# Steps:

# 1. Define a helper function `is\_perfect\_square` to check if a number is a perfect square.

def is\_perfect\_square(num):

if num < 0:

return False

root = int(num\*\*0.5)

return root \* root == num

# 2. Use the helper function to filter perfect squares from the list.

perfect\_squares = [num for num in list\_of\_numbers if is\_perfect\_square(num)]

return perfect\_squares

# Example:

# Input: list\_of\_numbers = [1, 4, 8, 9, 16, 20]

# Output: [1, 4, 9, 16]

pass

def q6(df, column):

# 3 pts

# Task: Replace all missing (`NaN`) values in a specified column with the column mean.

# Steps:

# 1. Compute the mean of the specified column, excluding missing values.

column\_mean = df[column].mean()

# 2. Use the `.fillna()` method to replace missing values with the computed mean.

df[column] = df[column].fillna(column\_mean)

# 3. Return the modified DataFrame.

return df

# Example:

# Input: df = pd.DataFrame({'A': [1, None, 3]}), column = 'A'

# Output: A modified DataFrame with no missing values.

pass

class A:

def \_\_init\_\_(self, a=None, b=None, c=None):

# 0 pts

# Task: Initialize an object with attributes `a`, `b`, and `c`.

# Steps:

if a is None:

self.a = r.randint(10, 20)

else:

self.a = a

if b is None:

self.b = r.randint(10, 20)

else:

self.b = b

if c is None:

self.c = r.randint(10, 20)

else:

self.c = c

def f1(self, other):

# 1 pt

# Task: Compare if the sum of attributes in `self` is greater than the sum of attributes in `other`.

if (self.a + self.b + self.c) > (other.a + other.b + other.c):

return True

return False

def f2(self, other):

# 2 pts

# Task: Compute the average of all possible sums between the attributes of `self` and `other`.

self\_values = [self.a, self.b, self.c]

other\_values = [other.a, other.b, other.c]

total\_sum = 0

count = 0

for self\_val in self\_values:

for other\_val in other\_values:

total\_sum += self\_val + other\_val

count += 1

return total\_sum / count if count > 0 else 0

def f3(self, other):

# 3 pts

# Task: Calculate the variance of the attributes of `self` and `other`.

# Steps:

# 1. Create two lists for the attributes of `self` and `other`.

list\_attributes\_self = [self.a, self.b, self.c]

list\_attributes\_other = [other.a, other.b, other.c]

# 2. Combine both lists into a Pandas Series.

combined\_series = pd.Series(list\_attributes\_self + list\_attributes\_other)

# 3. Calculate and return the variance of the combined Series.

return combined\_series.var()

def f4(self, others):

# 2 pts

# Task: Create a JSON file containing the attributes of `self` and other instances.

# Steps:

all\_instances = [self] + others

# 1. Create a list of dictionaries with the `a`, `b`, and `c` attributes of all instances.

all\_instances\_data = [{"a": i.a, "b": i.b, "c": i.c} for i in all\_instances]

# 2. Save the data as a JSON file named 'f4\_output.json'.

pd.DataFrame(all\_instances\_data).to\_json('f4\_output.json', orient='records')