################################

# STRUCTURES

# strings, lists, dictionaries

#

################################

# String is a data type used to represent text in double or single quotes

# In Python, strings are immutable. This means that they can't be modified

# use + to concatenate strings

# len() tells number od chars in string

# Modify the double\_word function so that it returns the same word repeated twice, followed by the length of the new doubled word. For example, double\_word("hello") should return hellohello10.

# def double\_word(word):

#    return ( word + word + str(len(word)) )

# print(double\_word("hello")) # Should return hellohello10

# print(double\_word("abc"))   # Should return abcabc6

# print(double\_word(""))      # Should return 0

# Lists in Python are defined using square brackets, with the elements stored in the list separated by

# commas: list = ["This", "is", "a", "list"]. You can use the len() function to return the number of elements in a list: len(list) would return 4.

# You can also use the in keyword to check if a list contains a certain element. If the element is present, it will return a True boolean.

# If the element is not found in the list, it will return False. For example, "This" in list would return True in our example.

# Similar to strings, lists can also use indexing to access specific elements in a list based on their position.

# You can access the first element in a list by doing list[0], which would allow you to access the string "This".

# In Python, lists and strings are quite similar. They’re both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over

# them using for loops; (2) support indexing; (3) using the len function to find the length of the sequence; (4) using the plus operator + in order to concatenate;

# and (5) using the in keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

print("#######################")

print("################STRING EXAMPLES####################")

print("#######################")

# String indexing

name= "Jaylen"

print(name[1])

# returns 'a' as index starts at 0

text="Tu is the best"

print(text[-1])

# this prints the last letter , goes from the right end

# Slice is the portion of a string that can contain more than one character, also sometimes called a substring

name= "Mei Vuong"

print(name[1:4])

# returns "ei"

# create new string based on existing

message = "Hello world"

new\_message = message[0:2] + message[3:0]

print(new\_message)

# A METHOD is a function associated with a specific class

# this is a function that applies to a variable

# this is the index method

pets="Cats & Dogs"

print(pets.index("&"))

# returns 5

# use IN to see if a substring is in a string

print("Cats" in pets)

# will return true

# function to replace email in old domain to new

# def replace\_domain(email, old\_domain, new\_domain)

#    if "@" + old\_domain in email:

#            index = email.index("@ + old\_domain")

#            new\_email= email[:index] + "@" + new\_domain

#            return new\_email

# return email

# useful string METHODS

# word.upper()   ; convert to uppercase

# word.lower()   ; lower case

# word.strip()   ; remove surrounding whitespaces,tabs etc

# word.lstrip()  ; remove to the left side

# word.rstrip()   ; remove to the right side

# word.count()   ; how many times a substring appears in a string

# word.isnumeric()   ; if its a numnber

# word.join()   ; similar to + to concatenate strings

# word.split()  ; splits by whitespaces into a list

# len(string) - Returns the length of the string

# for character in string - Iterates over each character in the string

# if substring in string - Checks whether the substring is part of the string

# string[i] - Accesses the character at index i of the string, starting at zero

# string[i:j] - Accesses the substring starting at index i, ending at index j minus 1. If i is omitted, its value defaults to 0.

# If j is omitted, the value will default to len(string).

# String methods

# string.lower() - Returns a copy of the string with all lowercase characters

# string.upper() - Returns a copy of the string with all uppercase characters

# string.lstrip() - Returns a copy of the string with the left-side whitespace removed

# string.rstrip() - Returns a copy of the string with the right-side whitespace removed

# string.strip() - Returns a copy of the string with both the left and right-side whitespace removed

# string.count(substring) - Returns the number of times substring is present in the string

# string.isnumeric() - Returns True if there are only numeric characters in the string. If not, returns False.

# string.isalpha() - Returns True if there are only alphabetic characters in the string. If not, returns False.

# string.split() - Returns a list of substrings that were separated by whitespace (whitespace can be a space, tab, or new line)

# string.split(delimiter) - Returns a list of substrings that were separated by whitespace or a delimiter

# string.replace(old, new) - Returns a new string where all occurrences of old have been replaced by new.

# delimiter.join(list of strings) - Returns a new string with all the strings joined by the delimiter

# FORMATTING strings

# You can use the format method on strings to concatenate and format strings in all kinds of powerful ways.

# To do this, create a string containing curly brackets, {}, as a placeholder, to be replaced

name="Mei"

number=len(name) \* 3

print("Hello {} , your lucky number is {}".format(name,number))

def student\_grade(name, grade):

    return ("{} received {}% on the exam".format(name,grade))

print(student\_grade("Reed", 80))

print(student\_grade("Paige", 92))

print(student\_grade("Jesse", 85))

price=7.5

with\_tax=price \* 1.09

print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price,with\_tax))

# You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example,

# the formatting expression {:.2f} means that you’d format this as a float number, with two digits after the decimal dot. The colon acts

# as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: >.

# For example, the expression {:>3.2f} would align the text three spaces to the right,

# as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

# If the placeholders indicate a number, they’re replaced by the variable corresponding to that order (starting at zero).

# "{0} {1}".format(first, second)

first = "apple"

second = "banana"

third = "carrot"

formatted\_string = "{0} {2} {1}".format(first, second, third)

print(formatted\_string)

# Outputs:

# apple carrot banana

# If the placeholders indicate a field name, they’re replaced by the variable corresponding to that field name.

# This means that parameters to format need to be passed indicating the field name.

# "{var1} {var2}".format(var1=value1, var2=value2)

# "{:exp1} {:exp2}".format(value1, value2)

# If the placeholders include a colon, what comes after the colon is a formatting expression

# {:d} integer value

# '{:d}'.format(10.5) → '10'

### formatting expressions ####

# {:d} ; integer value ; '{:d}'.format(10.5) → '10'

# {:.2f} ; floating point with that many decimals ; '{:.2f}'.format(0.5) → '0.50'

# {:.2s} ; string with that many characters ; '{:.2s}'.format('Python') → 'Py'

# {:<6s} ; string aligned to the left that many spaces ; '{:<6s}'.format('Py') → 'Py    '

# {:>6s} ; string aligned to the right that many spaces ; '{:>6s}'.format('Py') → '    Py'

# {:^6s} ; string centered in that many spaces ; '{:^6s}'.format('Py') → '  Py  '

# goes through letter by letter left to right

for x2 in "hello Tu":

    print(x2 + "boo")

    break

def convert\_distance(miles):

    km = miles \* 1.6

    result = "{} miles equals {:.1f} km".format(miles,km)

    return result

print(convert\_distance(12)) # Should be: 12 miles equals 19.2 km

print(convert\_distance(5.5)) # Should be: 5.5 miles equals 8.8 km

print(convert\_distance(11)) # Should be: 11 miles equals 17.6 km

def nametag(first\_name, last\_name):

    return("{} {:.1s}".format(first\_name,last\_name))

print(nametag("Jane", "Smith"))

# Should display "Jane S."

print(nametag("Francesco", "Rinaldi"))

# Should display "Francesco R."

print(nametag("Jean-Luc", "Grand-Pierre"))

# Should display "Jean-Luc G."

def replace\_ending(sentence, old, new):

    # Check if the old string is at the end of the sentence

    if sentence.endswith(old):

        # Using i as the slicing index, combine the part

        # of the sentence up to the matched string at the

        # end with the new string

        i = sentence.split()

        # example ["It's", 'raining', 'cats', 'and', 'cats']

        new\_sentence = ' '.join(i[0:-1]) + " " + new

        # logic = The string join() method returns a string by joining all the elements of an iterable (list, string, tuple), separated by the given separator.

        # text = ['Python', 'is', 'a', 'fun', 'programming', 'language']

        # join elements of text with space

        #  print(' '.join(text))

        # Output: Python is a fun programming language

        return new\_sentence

    return sentence

print(replace\_ending("It's raining cats and cats", "cats", "dogs"))

# Should display "It's raining cats and dogs"

print(replace\_ending("She sells seashells by the seashore", "seashells", "donuts"))

# Should display "She sells seashells by the seashore"

print(replace\_ending("The weather is nice in May", "may", "april"))

# Should display "The weather is nice in May"

print(replace\_ending("The weather is nice in May", "May", "April"))

# Should display "The weather is nice in April"

weather="Rainfall"

print(weather[:4])

# prints Rain

##################

# LISTS

#

###################

print("#######################")

print("################LISTS EXAMPLES####################")

print("#######################")

x= ["Now", "we", "are" , "cooking"]

print(type(x))

# outputs <class 'list'>

print(x)

# outputs ['Now', 'we', 'are', 'cooking']

print(len(x))

# returns 4 elements

print("are" in x)

# returns true

print(x[0])

# returns Now

print(x[1:3])

# returns "we" , "are"

# Using the "split" string method from the preceding lesson, complete the get\_word function to return the {n}th word from a passed sentence.

# For example, get\_word("This is a lesson about lists", 4) should return "lesson", which is the 4th word in this sentence.

# Hint: remember that list indexes start at 0, not 1.

def get\_word(sentence, n):

    # Only proceed if n is positive

    if n > 0:

        words = sentence.split()

        # Only proceed if n is not more than the number of words

        if n <= len(words):

            return(words[n-1])

    return("")

print(get\_word("This is a lesson about lists", 4)) # Should print: lesson

print(get\_word("This is a lesson about lists", -4)) # Nothing

print(get\_word("Now we are cooking!", 1)) # Should print: Now

print(get\_word("Now we are cooking!", 5)) # Nothing

## Modifying LISTS ##

# They are mutable so can change

# append method adds new element at the end

fruits = ["Pineapple" , "Banana" , "Apple" , "Melon"]

fruits.append("kiwi")

print(fruits)

# insert method allows to place where you want

fruits.insert(0, "Orange")

# inserts at the front

fruits.remove("Melon")

# removes the first occurrence

fruits.pop(3)

# removes index 3 position

fruits[2] = "Strawberry"

# this overwrites

# While lists and strings are both sequences, a big difference between them is that lists are mutable.

# This means that the contents of the list can be changed, unlike strings, which are immutable. You can add, remove, or modify elements in a list.

## TUPLES - sequences of elements of any type, that are immutable ##

# Tuples are like lists, since they can contain elements of any data type.

# But unlike lists, tuples are immutable. They’re specified using parentheses instead of square brackets.

#

#  The position of the elements inside the tuple have meaning and order matters

# like a collection of related info

#  A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple.

#  The order of the returned values is important, and a tuple ensures that the order isn’t going to change. Storing the elements of a tuple in separate variables is called unpacking.

# This allows you to take multiple returned values from a function and store each value in its own variable.

# Let's use tuples to store information about a file: its name, its type and its size in bytes.

# Fill in the gaps in this code to return the size in kilobytes (a kilobyte is 1024 bytes) up to 2 decimal places.

def file\_size(file\_info):

    tuple1, tuple2, tuple3= file\_info

    return("{:.2f}".format(tuple3 / 1024))

print(file\_size(('Class Assignment', 'docx', 17875))) # Should print 17.46

print(file\_size(('Notes', 'txt', 496))) # Should print 0.48

print(file\_size(('Program', 'py', 1239))) # Should print 1.21

# iterating over a list example below

# index is the first tuple, person is the second tuple

## ENUMERATE ##

# The enumerate() function takes a list as a parameter and returns a tuple for each element in the list.

# The first value of the tuple is the index and the second value is the element itself.

winners = [ "Tu" , "Mei" , "Thanh"]

for index,person in enumerate(winners):

    print("{} - {}".format(index + 1,person))

# this returns

# 1 - Tu

# 2 - Mei

# 3 - Thanh

# The enumerate function associates a counter as a key to each item in the set

snekTuple = ("cobra","viper","snek")

for i in enumerate(snekTuple):

    print(i)

# this returns

# (0, ‘cobra’)

# (1, ‘snek’)

# (2, ‘viper’)

# Try out the enumerate function for yourself in this quick exercise. Complete the skip\_elements function to return every other element from the list,

# this time using the enumerate function to check if an element is in an even position or an odd position.

print("ENUMERATE EXAMPLE")

def skip\_elements(elements):

    values = []

    for i,x in enumerate(elements):

        # this technique is good as it treats each element as a number and a string

    # use for function to iterate through the list

        if i % 2 == 0:

         # this means its  even

            values.append(x)

            #build the list

    return values

print(skip\_elements(["a", "b", "c", "d", "e", "f", "g"])) # Should be ['a', 'c', 'e', 'g']

print(skip\_elements(['Orange', 'Pineapple', 'Strawberry', 'Kiwi', 'Peach'])) # Should be ['Orange', 'Strawberry', 'Peach']

## List comprehensions ##

# Let us create new lists based on sequences of ranges

multiples = []

for x in range(1,11):

    multiples.append(x\*7)

print(multiples)

# using list comprehension

multiples= [x\*7 for x in range(1,11)]

print(multiples)

# The odd\_numbers function returns a list of odd numbers between 1 and n, inclusively.

# Fill in the blanks in the function, using list comprehension. Hint: remember that list and range counters start at 0 and end at the limit minus 1.

print("#####List compehnsion#########")

def odd\_numbers(n):

    return [x for x in range(1,n+1) if x % 2 !=0 ]

print(odd\_numbers(5))  # Should print [1, 3, 5]

print(odd\_numbers(10)) # Should print [1, 3, 5, 7, 9]

print(odd\_numbers(11)) # Should print [1, 3, 5, 7, 9, 11]

print(odd\_numbers(1))  # Should print [1]

print(odd\_numbers(-1)) # Should print []

# optional to keep code clearer only

# Lists and tuples are both sequences, so they share a number of sequence operations.

# But, because lists are mutable, there are also a number of methods specific just to lists.

# This cheat sheet gives you a run down of the common operations first, and the list-specific operations second.

# Common sequence operations

# len(sequence) - Returns the length of the sequence

# for element in sequence - Iterates over each element in the sequence

# if element in sequence - Checks whether the element is part of the sequence

# sequence[i] - Accesses the element at index i of the sequence, starting at zero

# sequence[i:j] - Accesses a slice starting at index i, ending at index j-1. If i is omitted, it's 0 by default. If j is omitted, it's len(sequence) by default.

# for index, element in enumerate(sequence) - Iterates over both the indexes and the elements in the sequence at the same time

# List-specific operations and methods

# list[i] = x - Replaces the element at index i with x

# list.append(x) - Inserts x at the end of the list

# list.insert(i, x) - Inserts x at index i

# list.pop(i) - Returns the element a index i, also removing it from the list. If i is omitted, the last element is returned and removed.

# list.remove(x) - Removes the first occurrence of x in the list

# list.sort() - Sorts the items in the list

# list.reverse() - Reverses the order of items of the list

# list.clear() - Removes all the items of the list

# list.copy() - Creates a copy of the list

# list.extend(other\_list) - Appends all the elements of other\_list at the end of list

# List comprehension

# [expression for variable in sequence] - Creates a new list based on the given sequence. Each element is the result of the given expression.

# [expression for variable in sequence if condition] - Creates a new list based on the given sequence. Each element is the result of the given expression; elements only get added if the condition is true.

print("LISTS PRACTICE QUESTIONS!!!!!")

# Given a list of filenames, we want to rename all the files with extension hpp to the extension h.

# To do this, we would like to generate a new list called newfilenames, consisting of the new filenames.

# Fill in the blanks in the code using any of the methods you’ve learned thus far, like a for loop or a list comprehension.

filenames = ["program.c", "stdio.hpp", "sample.hpp", "a.out", "math.hpp", "hpp.out"]

# Generate newfilenames as a list containing the new filenames

# using as many lines of code as your chosen method requires.

#initialize our new list

newfilenames=[]

# lets iterate through the list

for i in filenames:

    # split into 2 strings to compare delimited by decimal

    word=i.split(".")

    #print(word)

    #if the last 3 chars are .hpp then match

    if word[1] == "hpp":

            print(word[1])

            newfilenames.append(word[0] + ".h")

            #break

    else:

        newfilenames.append(i)

        # note that i is the original word before being split

        #break

print(newfilenames)

# Should be ["program.c", "stdio.h", "sample.h", "a.out", "math.h", "hpp.out"]

print("Practice question 2!!!!")

#Let's create a function that turns text into pig latin: a simple text transformation that modifies each word moving the first character to the end and appending

# "ay" to the end. For example, python ends up as ythonpay.

def pig\_latin(text):

  say = ""

  # Separate the text into words

  words = text.split()

  for word in words:

    # iterate through each letter in the word

    # Create the pig latin word and add it to the list

    # say=word[0]

    # this maniupulates the string. Get from 2nd letter to the end , add first letter, add "ay"

    # say is not a list , so we cannot append it, we need to concatenate strings instead

    say += (word[1:] + word[0] + "ay" + " ")

    # Turn the list back into a phrase

  return say

print(pig\_latin("hello how are you")) # Should be "ellohay owhay reaay ouyay"

print(pig\_latin("programming in python is fun")) # Should be "rogrammingpay niay ythonpay siay unfay"

print("New Practice question")

# The permissions of a file in a Linux system are split into three sets of three permissions: read, write, and execute for the owner, group, and others.

# Each of the three values can be expressed as an octal number summing each permission, with 4 corresponding to read, 2 to write, and 1 to execute.

# Or it can be written with a string using the letters r, w, and x or - when the permission is not granted.

# For example:

# 640 is read/write for the owner, read for the group, and no permissions for the others; converted to a string, it would be: "rw-r-----"

# 755 is read/write/execute for the owner, and read/execute for group and others; converted to a string, it would be: "rwxr-xr-x"

# Fill in the blanks to make the code convert a permission in octal format into a string format.

def octal\_to\_string(octal):

    result = ""

    # result is a string and not a list. important differentiation

    value\_letters = [(4,"r"),(2,"w"),(1,"x")]

    # Iterate over each of the digits in octal

    for x in [int(n) for n in str(octal)]:

        # Check for each of the permissions values

        #print(x)

        # This would simply loop the single integer

        for value, letter in value\_letters:

            print("hi")

            # value is assigned first element in tuple , letter the second

            # eg if 7 >= 4

        #   if x >= value:

        #        result += x

        #        \_\_\_ -= value

        #    else:

        #        result="-"

    return result

print(octal\_to\_string(755)) # Should be rwxr-xr-x

print(octal\_to\_string(644)) # Should be rw-r--r--

print(octal\_to\_string(750)) # Should be rwxr-x---

print(octal\_to\_string(600)) # Should be rw-------

print("New practice Question")

# The group\_list function accepts a group name and a list of members,

# and returns a string with the format: group\_name: member1, member2, … For example, group\_list("g", ["a","b","c"]) returns "g: a, b, c".

# Fill in the gaps in this function to do that.

def group\_list(group, users):

  # this defines a string

  # techniques to join a list of elements delimited by comma

  # great technique!!!!!!!

    members =', '.join([str(elem) for elem in users])

    #print(members)

  # we dont want to return a list

  # a for loop to get all the elements is what we want

    return( group + ": " + members)

print(group\_list("Marketing", ["Mike", "Karen", "Jake", "Tasha"])) # Should be "Marketing: Mike, Karen, Jake, Tasha"

print(group\_list("Engineering", ["Kim", "Jay", "Tom"])) # Should be "Engineering: Kim, Jay, Tom"

print(group\_list("Users", "")) # Should be "Users:"

########

# Convert a List to a string methods

# differentiate in dealing with strings and a list of strings

########

# Python program to convert a list to string

###############################

# My favourites and easiest way is to use:

#

# ' '.join(list)   is the MOST EASY way to convert list elements into a string!!!!!!!

###############################

# Function to convert

print("Convert string to list method 1")

def listToString(s):

    # initialize an empty string

    str1 = ""

    # traverse in the string

    for ele in s:

        str1 += ele

    # return string

    return str1

# Driver code

s = ['Geeks', 'for', 'Geeks']

print(listToString(s))

print("Convert string to list method 2")

# Python program to convert a list

# to string using join() function

# Function to convert

def listToString(s):

    # initialize an empty string

    str1 = " "

    # return string

    return (str1.join(s))

# Driver code

s = ['Geeks', 'for', 'Geeks']

print(listToString(s))

print("Convert string to list method 3")

# Python program to convert a list

# to string using list comprehension

s = ['I', 'want', 4, 'apples', 'and', 18, 'bananas']

# using list comprehension

listToStr = ' '.join([str(elem) for elem in s])

print(listToStr)

print("Convert string to list method 4")

# using in operator

s = ['Geeks', 'for', 'Geeks']

for i in s:

  print(i,end=" ")

print("Another practice question!!!")

# The guest\_list function reads in a list of tuples with the name, age, and profession of each party guest, and

# prints the sentence "Guest is X years old and works as \_\_." for each one. For example, guest\_list(('Ken', 30, "Chef"), ("Pat", 35, 'Lawyer'),

# ('Amanda', 25, "Engineer")) should print out: Ken is 30 years old and works as Chef. Pat is 35 years old and works as Lawyer.

# Amanda is 25 years old and works as Engineer. Fill in the gaps in this function to do that.

def guest\_list(guests):

    for a,b in enumerate(guests):

        # a is now the index

        # b is now a string

        print("{} is {} years old and works as {}".format(b[0],b[1],b[2]))

# pass in tuples. We want to break each tuple into idividual elements

# The enumerate() function takes a list as a parameter and returns a tuple for each element in the list

guest\_list([('Ken', 30, "Chef"), ("Pat", 35, 'Lawyer'), ('Amanda', 25, "Engineer")])

##########################################

# DICTIONARIES

# The data inside dictionaries take the form of pairs of keys and values

# Dictionaries are mutable - can add/remove entries

##########################################

print("#######################")

print("################DICTIONARY EXAMPLES####################")

print("#######################")

file\_counts={"jpg":10, "txt":14, "csv":2, "py":23}

# {key:value}

print(file\_counts)

# prints all

print(file\_counts["txt"])

# returns 14

file\_counts["cfg"]=8

print(file\_counts)

# appends to the end additional

file\_counts["csv"]=17

# this replaces existing value

del file\_counts["cfg"]

# deletes the key value

# Dictionaries are another data structure in Python. They’re similar to a list in that they can be used to organize data into collections.

# However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values.

# You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer,

# float, or even tuples.

# When creating a dictionary, you use curly brackets: {}. When storing values in a dictionary, the key is specified first, followed by the corresponding value,

# separated by a colon. For example, animals = { "bears":10, "lions":1, "tigers":2 } creates a dictionary with three key value pairs, stored in the variable animals.

# The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2.

# You can access the values by referencing the key, like this: animals["bears"]. This would return the integer 10, since that’s the corresponding value for this key.

# You can also check if a key is contained in a dictionary using the in keyword. Just like other uses of this keyword, it will return True if the key is found in the

# dictionary; otherwise it will return False.

# Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists.

# You can add a new key value pair to a dictionary by assigning a value to the key, like this: animals["zebras"] = 2. This creates the new key in the animal dictionary

# called zebras, and stores the value 2. You can modify the value of an existing key by doing the same thing. So animals["bears"] = 11 would change the value stored in

# the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the del keyword. By doing del animals["lions"] you would remove the key value

# pair from the animals dictionary.

## Iterating over contents of a dictionary ##

file\_counts={"jpg":10, "txt":14, "csv":2, "py":23}

for extension in file\_counts:

    print(extension)

    #prints keys

# return the tuples

for ext,amount in file\_counts.items():

    print("There are {} files with the .{} extension".format(amount,ext))

# return keys and values specifically

file\_counts={"jpg":10, "txt":14, "csv":2, "py":23}

print(file\_counts.keys())

print(file\_counts.values())

for value in file\_counts.values():

    print(value)

# Example : Find how many letters appear in a text

def count\_letters(text):

    #initialize empty dictionary

    result={}

    for letter in text:

        # iterate through the STRING and check if its not already in the dictionary

        if letter not in result:

            # adding to a dictionary

            result[letter]=0

        # increment count for that letter in the dictionary

        result[letter] += 1

    return result

print(count\_letters("aaaa"))

# returns {'a': 4}

print(count\_letters("tenant"))

# returns {'t': 2, 'e': 1, 'n': 2, 'a': 1}

# You can iterate over dictionaries using a for loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary.

# If you want to access the corresponding values associated with the keys, you could use the keys as indexes.

# Or you can use the items method on the dictionary, like dictionary.items().

# This method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

# If you only wanted to access the keys in a dictionary, you could use the keys() method on the dictionary: dictionary.keys().

# If you only wanted the values, you could use the values() method: dictionary.values().

# Examples of when to use a LIST over a DICTIONARY

# ip\_addresses = ["192.168.1.1" , "127.0.0.1"]    , sequence as a list

# host\_addresses = {"router": "192.168.1.1" , "localhost": "127.0.0.1"}    keyvalue so dictionary better

# use dictionaries when you plan on searching for a specific element

# A SET is used when you want to store a bunch of elements and be certain that they're only present once

# elements in a SET must be immutable

# Dictionary Methods Cheat Sheet

# Syntax

# x = {key1:value1, key2:value2}

# Operations

# len(dictionary) - Returns the number of items in the dictionary

# for key in dictionary - Iterates over each key in the dictionary

# for key, value in dictionary.items() - Iterates over each key,value pair in the dictionary

# if key in dictionary - Checks whether the key is in the dictionary

# dictionary[key] - Accesses the item with key key of the dictionary

# dictionary[key] = value - Sets the value associated with key

# del dictionary[key] - Removes the item with key key from the dictionary

# Methods

# dict.get(key, default) - Returns the element corresponding to key, or default if it's not present

# dict.keys() - Returns a sequence containing the keys in the dictionary

# dict.values() - Returns a sequence containing the values in the dictionary

# dict.update(other\_dictionary) - Updates the dictionary with the items coming from the other dictionary. Existing entries will be replaced; new entries will be added.

# dict.clear() - Removes all the items of the dictionary

# The email\_list function receives a dictionary, which contains domain names as keys, and a list of users as values.

# Fill in the blanks to generate a list that contains complete email addresses (e.g. diana.prince@gmail.com).

def email\_list(domains):

    # we want to build a list

    # always good practice to initialize a list

    emails = []

    for x,users in domains.items():

      for user in users:

        # return the list elements + domain into a new list

        # inside the append we are building a string so use concatanate

        emails.append(user + "@" + x)

    return(emails)

print(email\_list({"gmail.com": ["clark.kent", "diana.prince", "peter.parker"], "yahoo.com": ["barbara.gordon", "jean.grey"], "hotmail.com": ["bruce.wayne"]}))

# The dict.update method updates one dictionary with the items coming from the other dictionary, so that existing entries are replaced and new entries are added.

# What is the content of the dictionary “wardrobe“ at the end of the following code?

wardrobe = {'shirt': ['red', 'blue', 'white'], 'jeans': ['blue', 'black']}

new\_items = {'jeans': ['white'], 'scarf': ['yellow'], 'socks': ['black', 'brown']}

wardrobe.update(new\_items)

print(wardrobe)

# {'shirt': ['red', 'blue', 'white'], 'jeans': ['white'], 'scarf': ['yellow'], 'socks': ['black', 'brown']}

# this adds the new scarf and socks key and also changes jeans key

# The groups\_per\_user function receives a dictionary, which contains group names with the list of users.

# Users can belong to multiple groups. Fill in the blanks to return a dictionary with the users as keys and a list of their groups as values.

# basically reverse key and values

# {admin : [local, public, administrator] , userA: [local] , userB : [public]}

# Adding to empty dictionary refresher

# dictionary['colour']= 'green'

# where colour is the key

# green is the value

print("Dictionaries example")

# The add\_prices function returns the total price of all of the groceries in the  dictionary. Fill in the blanks to complete this function.

def add\_prices(basket):

    # Initialize the variable that will be used for the calculation

    total = 0

    # Iterate through the dictionary items

    for fruit,price in basket.items():

        # Add each price to the total calculation

        # Hint: how do you access the values of

        # dictionary items?

        total += price

    # Limit the return value to 2 decimal places

    return round(total, 2)

groceries = {"bananas": 1.56, "apples": 2.50, "oranges": 0.99, "bread": 4.59,

    "coffee": 6.99, "milk": 3.39, "eggs": 2.98, "cheese": 5.44}

print(add\_prices(groceries)) # Should print 28.44

def groups\_per\_user(group\_dictionary):

    # define a dictionary

    user\_groups = {}

    # Go through group\_dictionary

    for group,users in group\_dictionary.items():

        # Now go through the users in the group

        for user in users:

                    # add new entries to the dictionary

                    # This works using the get command

                    user\_groups[user] = user\_groups.get(user,[])   + [group]

            # Now add the group to the the list of

            # groups for this user, creating the entry

            # in the dictionary if necessary

    return(user\_groups)

print(groups\_per\_user({"local": ["admin", "userA"],

        "public":  ["admin", "userB"],

        "administrator": ["admin"] }))

#####################################

# MODULE EXAM

#

#####################################

# The format\_address function separates out parts of the address string into new strings: house\_number and street\_name, and returns: "house number X on street named Y".

# The format of the input string is: numeric house number, followed by the street name which may contain numbers, but never by themselves, and could be several words long.

# For example, "123 Main Street", "1001 1st Ave", or "55 North Center Drive". Fill in the gaps to complete this function.

print("###Question 1#######")

def format\_address(address\_string):

  # TC- turn it into a list where index 0 is the number and the rest is the street name

  # Declare variables

  address\_list=address\_string.split()

  street=""

  # Separate the address string into parts

  # Traverse through the address parts

  for x in address\_list:

    # Determine if the address part is the

    # house number or part of the street name

    if x.isnumeric():

        # Need to convert list to string here!!!!!!

        # ' '.join(list)   is the MOST EASY way to convert list elements into a string!!!!!!!

        #

        street=' '.join(address\_list[1:])

    break

  # Does anything else need to be done

  # before returning the result?

  # TC - we want to return a string sequence

  # Return the formatted string

  return "house number {} on street named {}".format(x,street)

print(format\_address("123 Main Street"))

# Should print: "house number 123 on street named Main Street"

print(format\_address("1001 1st Ave"))

# Should print: "house number 1001 on street named 1st Ave"

print(format\_address("55 North Center Drive"))

# Should print "house number 55 on street named North Center Drive"

# The highlight\_word function changes the given word in a sentence to its upper-case version.

# For example, highlight\_word("Have a nice day", "nice") returns "Have a NICE day".

# Can you write this function in just one line?

print("###Question 2#######")

# use the string REPLACE function

def highlight\_word(sentence, word):

    return(sentence.replace(word, word.upper()))

print(highlight\_word("Have a nice day", "nice"))

print(highlight\_word("Shhh, don't be so loud!", "loud"))

print(highlight\_word("Automating with Python is fun", "fun"))

#def odd\_numbers(n):

#   return [x for x in range(1,n+1) if x % 2 !=0 ]

print("###Question 3 #######")

# A professor with two assistants, Jamie and Drew, wants an attendance list of the students, in the order that they arrived in the classroom.

# Drew was the first one to note which students arrived, and then Jamie took over. After the class, they each entered their lists into the computer and emailed them to the professor,

# who needs to combine them into one, in the order of each student's arrival. Jamie emailed a follow-up, saying that her list is in reverse order.

# Complete the steps to combine them into one list as follows: the contents of Drew's list, followed by Jamie's list in reverse order, to get an accurate list of the students as they arrived.

def combine\_lists(list1, list2):

  # Generate a new list containing the elements of list2

  # Followed by the elements of list1 in reverse order

  # TC - reverse list 1 . There is a specific reverse() method

  #reverse\_list1= list1.reverse()

  list1.reverse()

  # for some reason, can't assign a variable to the reverse list outcome

  # we can concatenate lists

  return(list2 + list1)

Jamies\_list = ["Alice", "Cindy", "Bobby", "Jan", "Peter"]

Drews\_list = ["Mike", "Carol", "Greg", "Marcia"]

print(combine\_lists(Jamies\_list, Drews\_list))

print("###Question 4 #######")

# Use a list comprehension to create a list of squared numbers (n\*n). The function receives the variables start and end, and returns a list of squares of consecutive numbers between start and end inclusively.

# For example, squares(2, 3) should return [4, 9].

def squares(start, end):

    return [ x \*\*2 for x in range(start,end+1) ]

print(squares(2, 3)) # Should be [4, 9]

print(squares(1, 5)) # Should be [1, 4, 9, 16, 25]

print(squares(0, 10)) # Should be [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

print("###Question 5 #######")

# Complete the code to iterate through the keys and values of the car\_prices dictionary, printing out some information about each one.

def car\_listing(car\_prices):

  result = ""

  # we want each individual key, value item in tuple

  for car,price in car\_prices.items():

    result += "{} costs {} dollars".format(car,str(price)) + "\n"

  return result

# { indicates a tuple}

print(car\_listing({"Kia Soul":19000, "Lamborghini Diablo":55000, "Ford Fiesta":13000, "Toyota Prius":24000}))

print("###Question 6 #######")

# Taylor and Rory are hosting a party. They sent out invitations, and each one collected responses into dictionaries, with names of their friends and how many guests each friend is bringing.

# Each dictionary is a partial list, but Rory's list has more current information about the number of guests. Fill in the blanks to combine both dictionaries into one,

# with each friend listed only once, and the number of guests from Rory's dictionary taking precedence, if a name is included in both dictionaries.

# Then print the resulting dictionary.

def combine\_guests(guests1, guests2):

  # Combine both dictionaries into one, with each key listed

  # only once, and the value from guests1 taking precedence

    new\_list={}

    new\_list.update(guests1)

    for guest2,number\_of\_guests\_2 in guests2.items():

        # if the name does not exist in any of the keys, add the name to the list

        if guest2 not in guests1.keys():

        #print(guest2)

            new\_list.update({guest2:number\_of\_guests\_2})

        #print(new\_list)

        #break

    return(new\_list)

Rorys\_guests = { "Adam":2, "Brenda":3, "David":1, "Jose":3, "Charlotte":2, "Terry":1, "Robert":4}  #takes precedence

Taylors\_guests = { "David":4, "Nancy":1, "Robert":2, "Adam":1, "Samantha":3, "Chris":5}

print(combine\_guests(Rorys\_guests, Taylors\_guests))