Lesson 8: Prepare for the Final Exercise

Table of Contents

8.1. Objectives

- Examine the implications of using computation to solve a problem
  - Discuss best practices for using computation to solve a problem
  - Suggest types of problems that can be solved through computation
  - Show how computation can solve a problem

- Recognize key computer science concepts
  - Identify data types used in Python scripting
  - Identify data structures used in Python scripting
  - Define variables and strings
  - Recognize how queries operate

- Demonstrate the ability to build basic scripts using Python scripting language
  - Use various data types and structures in Python scripting
  - Collect data using Python scripting
  - Extract data using Python scripting
  - Develop advanced data structures using Python scripting

Instructor Guidance: Technical Facilitators found that this lesson works best when allowing students at least an hour at the beginning of the session to work through the lesson at their own pace and to identify issues they may have. These issues can be addressed either one-on-one, or with the entire class.

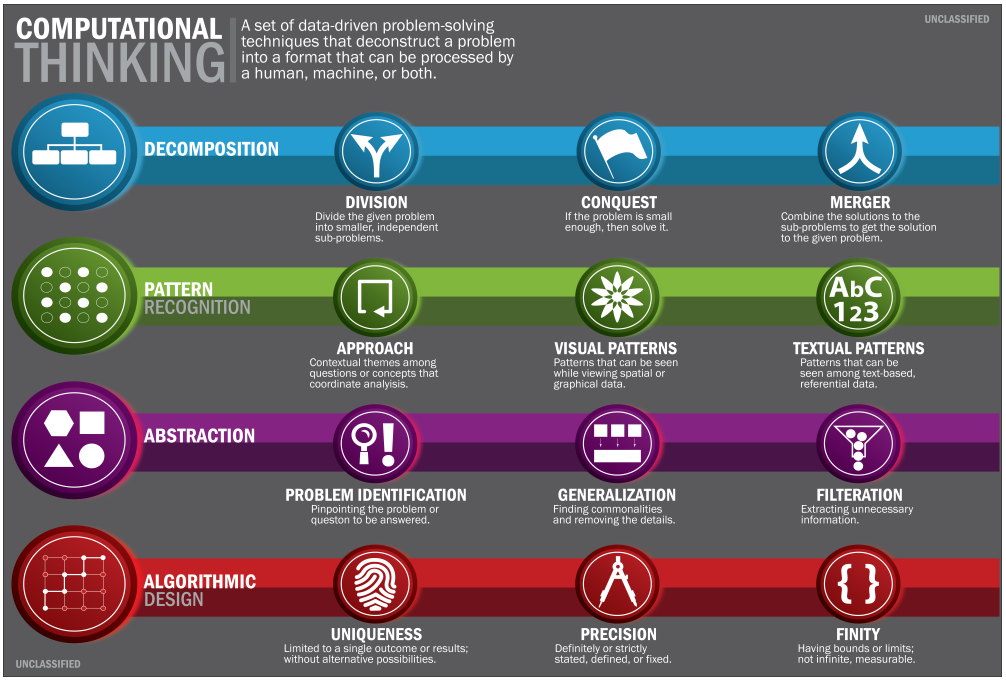
8.2. Overview

You've made it to Lesson 8. We have covered a lot of material in a very short time, so it is time to do a comprehensive review. In this lesson we are going to review everything that has been covered in the course:

- Computational Thinking
- Python Data Types and Structures
- Python Loops and Functions
- Getting Data
- Working with Data

8.3. Computational Thinking

Computers are not smart, but they are fast. They need explicit instructions in order to solve a problem. This means that when we, as programmers, are presented with a problem, we have to provide the computer a complete set of instructions for it to do what we want. This requires a special mode of reasoning that we call *computational thinking*.



The basic steps for approaching a problem with computational thinking are:

1. **Decomposition:** Breaking down data, processes, or problems into smaller, manageable parts
    A. Make sure you understand the problem
        a. Identify the inputs
        b. Identify what the output should look like
    B. If it's a large scale problem, identify possible smaller components

2. **Pattern Recognition:** Observing patterns, trends, and regularities in data
    A. Inspect your data
        a. Make sure the inputs and outputs you've identified make sense
        b. Look for trends or possible fringe cases

3. **Abstraction:** Identifying the general principles that generate these patterns
    A. Work out a problem by hand
        b. As you work out by hand, attempt to think beyond a single example

4. **Algorithm Design:** Developing the step by step instructions for solving the problem
    A. Begin to formalize your steps into pseudocode
    B. Identify available software tools
        a. These may be available packages, things you've written in the past, or built-in python functionality
    C. Implement your pseudocode

Exercise: For each exercise in this notebook, write your own pseudocode before you begin coding a solution.

8.4. Python Data Types

See the table below for descriptions of each type.

Plain Language Description	Python Data Type	`type()` Output
True or False	Boolean	`bool`
Whole Numbers	Integer	`int`
Decimal Numbers	Float	`float`
Sequences of characters including letters, numbers, formatting characters, and/or punctuation	String	`str`
Nothingness, non-existence; not the same as zero or empty	None	`NoneType`

8.4.1. Boolean Values and Operators

Exercise: Below, create two Boolean values and store them in variables.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
b1 = True
b2 = False

print(b1, b2)
```

8.4.1.1. Comparison Operators

You can compare values in a number of different ways, shown in the table below.

Operator	Operation
==	equals
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

In [ ]:

```
grade = 50
```

Above, we initialize a variable that holds an integer. Can you guess the result of the expressions below? Run each cell to output the result. You can also change the value of `grade` and see how the results change.

In [ ]:

```
grade > 50
```

In [ ]:

```
grade <= 50
```

Exercise: Define two variables that store numeric values (either `int` or `float`), and write expressions that check whether the values are equal to, not equal to, greater than, or less than each other.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
num_1 = 67
num_2 = 188.5

print(num_1 == num_2)
print(num_1 != num_2)
print(num_1 > num_2)
print(num_1 < num_2)
```

8.4.1.2. Membership Operators

The membership operators below are used to write membership tests.

Operator	Operation
`in`	Checks if the left side of expression contained in the right side of expression
`not in`	Checks if the left side of expression is not contained in the right side of expression

In [ ]:

```
NATO = ['Belgium', 'Canada', 'Denmark', 'France', 'Iceland', 'Italy', 'Luxembourg', 'Neth
        'Norway', 'Portugal', 'United Kingdom', 'United States', 'Greece', 'Turkey', 'Ger
        'Spain', 'Czech Republic', 'Hungary', 'Poland', 'Bulgaria', 'Estonia', 'Latvia',
        'Lithuania', 'Romania', 'Slovakia', 'Slovenia', 'Albania', 'Croatia', 'Montenegro
```

Above, we save a list of strings to a variable. Below, we can run membership tests on it. Guess the results and then run each expression to output the result.

In [ ]:

```
'United Kingdom' in NATO
```

In [ ]:

```
'Denmark' not in NATO
```

Exercise: Create a variable that stores a string, and write a membership test to check whether that string is contained in the list `NATO`. Does changing the capitalization in your string affect the outcome?

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_string = 'Slovakia'

my_string in NATO
```

Exercise: Create a variable that stores a number, and write a membership test to check whether that value is contained in the list below. Does using a float vs. an integer affect your result?

In [ ]:

```
test_list = [493, 168, 75, 361, 297, 71, 308, 5, 36, 174, 378, 155, 252, 348, 151, 354, 1
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_num = 36.0

my_num in test_list
```

8.4.1.3. Logic Operators

Logical operators are used to combine Boolean expressions or values into larger Boolean expressions.

Operator	Operation
and	Checks if two statements are *both* True

Operator	Operation
`or`	Checks if *either* of two statements are True
`not`	Negates the value of a boolean statement

Guess the result of the Boolean expression below.

In [ ]:

```
not ('h' != 'H' and 1 >= 2) or False
```

Exercise: In the cells below, replace the `?` 's with Python operators or keywords that make each of the statements output **True**.

In [ ]:

```
('Estonia' in NATO) ? ('Atlantis' in NATO)
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
('Estonia' in NATO) or ('Atlantis' in NATO)
```

In [ ]:

```
100 ? 55
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
100 > 55
```

In [ ]:

```
'Russia' ? NATO
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
'Russia' not in NATO
```

In [ ]:

```
78 ? 73
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
78 != 73
```

8.4.2. Reserved Python Keywords

There are words within Python that have specific use cases and cannot be used as variable names. These words show up green and bold in Jupyter Notebook code cells. Run the following cell to output a list of all of Python's reserved keywords. Look through the list and define in your own words any keyword you've learned in this class.

In [ ]:

```
import keyword

print(keyword.kwlist)
```

8.4.2.1. Built-In Functions

Beginning programmers should never overwrite built-in function names. Once you re-assign the function's name to something else, you have lost the ability to use that function, which is bad because these functions are fundamental to some core Python capabilities. Moreover, errors stemming from accidental overwrites of Python functions can be very hard to debug. Here are some built-in functions you are already familiar with. Do you remember what they do?

```
print()
type()
max()
min()
int()
list()
dict()
str()
```

8.4.3. Numbers (Integers and Floats)

Guess the output of the cells below.

In [ ]:

```
2 == 2.0
```

In [ ]:

```
type(2) == type(2.0)
```

Doing mathematical operations with integers and floats returns a float as the result.

In [ ]:

```
4 + 4.0 + 10
```

8.4.3.1. Arithmetic Operators

Operator	Operation
+	addition
-	subtraction
/	division
*	multiplication
//	floor division
%	modulus - returns only the remainder
**	exponent

Examples of calling the exponent operator:

In [ ]:

```
2 ** 10
```

In [ ]:

```
25 ** 0.5
```

Examples of calling the modulus operator:

In [ ]:

```
17 % 4
```

In [ ]:

```
16 % 7
```

In [ ]:

```
248 % 2
```

Examples of floor division:

In [ ]:

```
17 // 4
```

In [ ]:

```
5 // 2
```

In [ ]:

```
-5 // 2
```

8.4.3.2. Assignment Operators

Assignment operators combine arithmetic and assignment into a single statement.

Operator	Operation
+=	add right operand to the left operand and assign the result to left operand
-=	subtract right operand from the left operand and assign the result to left operand
*=	multiply right operand and the left operand and assign the result to left operand

Examples of assignment operators:

In [ ]:

```
number = 4

number += 5
number
```

In [ ]:

```
number = 7

number *= 3
number
```

8.4.3.3. Casting

Guess the output of the cell below.

In [ ]:

```
'11.186' + 2
```

Exercise: Fix the code in the cell above so it runs without throwing an error. HINT: Use casting.

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
float('11.186') + 2
```

Exercise: In the code cells below, replace the `???` 's with casting functions to make the code run without errors.

In [ ]:

```
7 ** ???('2')
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
7 ** int('2')
```

In [ ]:

```
print("My team has won " + ???(20) + " awards collectively.")
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
print("My team has won " + str(20) + " awards collectively.")
```

In [ ]:

```
PI = '3.14159'
int(???(PI) * 100) / 100
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
PI = '3.14159'
int(float(PI) * 100) / 100
```

In [ ]:

```
my_num = 654321
print("The first three digits in my_num are " + ???(my_num)[:3])
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_num = 654321
print("The first three digits in my_num are " + str(my_num)[:3])
```

8.4.4. None

The `None` data type is used to denote the absence of data.

In [ ]:

```
type(None)
```

One helpful note about the `None` data type is that when cast to a Boolean it evaluates to `False`

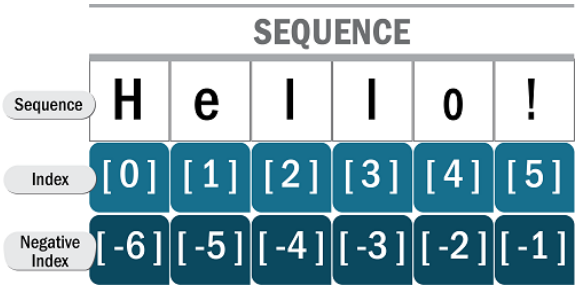
In [ ]:

```
bool(None)
```

8.5. Python Data Structures

8.5.1. Sequences

A sequence is an ordered grouping of elements. Each element of a sequence is assigned a number which is its position or index.



8.5.2. Lists

Lists are mutable sequences.

Syntax	Description
`list2 = []`	Initializes an empty list
`list3[0]`	Indexes a list
`list3[1:2]`	Slices a list
`list_variable.append(x)`	Appends `x` to the end of the list
`list_variable.extend(my_list)`	Appends each element of `my_list` to the list
`list_variable.insert(i, x)`	Inserts `x` at index `i` in the list
`list_variable.remove(x)`	Removes the first appearance of `x` in the list
`list_variable.pop(i)`	Removes the item at index `i` in the list and returns that value

Syntax	Description
`list_variable.index(x)`	Returns the index where `x` first appears in the list; throws an error if `x` is not contained in the list
`list_variable.count(x)`	Counts the number of times `x` appears in the list
`list_variable.sort(reverse=False)`	Sorts the order of the list
`list_variable.reverse()`	Reverses the order of the list

Run the following cell to save the list of strings below.

In [ ]:

```
grocery_list = ['apples', 'ice cream', 'yogurt']
```

Exercise: Append `'walnuts'` to `grocery_list`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_list.append('walnuts')
grocery_list
```

Exercise: Remove the first item from `grocery_list` and save the removed value to a variable.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
last_item = grocery_list.pop(0)

print(last_item)
print(grocery_list)
```

Exercise: Extend `grocery_list` with the items in the list below.

In [ ]:

```
new_grocery_list = ['triple sec', 'tequila', 'limes', 'salt']
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_list.extend(new_grocery_list)
grocery_list
```

You can also add two lists together, which does the same thing as `.extend()` .

In [ ]:

```
taco_night_grocery_list = ['taco shells', 'beef', 'cheese', 'salsa', 'chips']

grocery_list = grocery_list + taco_night_grocery_list
grocery_list
```

Using the list below, make the following changes.

In [ ]:

```
courses2018 = ['Chemistry 237', 'Archery 101', 'Bowling 201', 'Computer Science 211', 'Ge
```

Exercise: Remove `'Archery 101'` from the course list.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
courses2018.remove('Archery 101')
courses2018
```

Exercise: Add `'Physics 202'` to the list.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
courses2018.append('Physics 202')
courses2018
```

Exercise: Sort the list alphabetically in descending order (Z-A).

In [ ]:

```
courses2018 = ['Chemistry 237', 'Archery 101', 'Bowling 201', 'Computer Science 211', 'Ge
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
courses2018.sort(reverse=True)
courses2018
```

Using the list below, make the following changes using Python:

In [ ]:

```
num_list = [68, 5, 76, 81, 19, 33, 8, 26, 84, 47, 43, 25, 61, 22, 27]
```

Exercise: Count the number of items in the list (using code, not your fingers!).

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(num_list)
```

Exercise: Move the first item in the list to the end of the list.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
num_list.append(num_list.pop(0))
num_list
```

Exercise: Find the maximum value in the list using the built-in `max()` function.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
max(num_list)
```

Exercise: Find the maximum value in the list by sorting the list in ascending order and then grabbing the last element.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
sorted(num_list)[-1]
```

8.5.3. Tuples

Tuples are immutable sequences.

Code	Action
`my_tuple[0]`	Indexing a tuple
`x, y = my_tuple`	Multiple assignment

In [ ]:

```
weekdays = ('Monday','Tuesday','Wednesday','Thursday','Friday')
```

Why does the cell below throw an error?

In [ ]:

```
weekdays[0] = 'Lunes'
weekdays
```

Exercise: Using indexing, extract the third element from within the second-to-last element of the following tuple. HINT: The target element is the integer `22`.

In [ ]:

```
my_tuple = ((13, 2, 60, 6),
            (54, 0, 20, 37),
            (65, 62, 9, 49),
            (18, 69, 8, 54),
            (55, 52, 22, 53),
            (62, 57, 75, 20))
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_tuple[-2][2]
```

8.5.4. Strings

Strings are immutable sequences of characters.

Syntax	Description
`str(x)`	Converts `x` to a string
`string_a + string_b`	Concatenation: appends `string_b` to `string_a`
`my_string * 3`	Repeats the string 3 times
`my_string[i]`	String indexing
`my_string[begin_index:end_index]`	String slicing
`string_variable.upper()`	Converts all of the alpha characters to uppercase
`string_variable.lower()`	Converts all of the alpha characters to lowercase
`string_variable.strip()`	Removes leading and trailing whitespace from a string
`string_variable.split(sub_string)`	Splits a string using `sub_string` as a delimiter; returns a list of string fragments
`string_variable.replace(old, new)`	Replaces all occurrences of `old` with `new`
`string.find(sub_string)`	Searches a string for `sub_string` , returns the index location of first character of substring (or -1 if `sub_string` is not found)

Below, we demonstrate some of the things you can do with strings.

In [ ]:

```
string_variable = '   Hello World   '
```

Convert all the characters to uppercase.

In [ ]:

```
string_variable.upper()
```

Convert all the characters to lowercase.

In [ ]:

```
string_variable.lower()
```

Remove the leading and trailing white space from the string.

In [ ]:

```
string_variable.strip()
```

Split the words in the string.

In [ ]:

```
string_variable.split()
```

Exercise: Write a script to make a new string from a given string where all occurrences of the first character have been changed to `'$'` , except the first character itself. HINT: Use indexing, slicing and `.replace()` .

Example Input	Expected Output
restart	resta$t
pineapple	pinea$$le

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
input_string = 'restart'

output_string = input_string[0] + input_string[1:].replace(input_string[0], '$')
output_string
```

Exercise: Write a Python script to take two strings, swap the first two characters of each string, then combine them into one string separated by a space. HINT: Use slicing and string concatenation (+).

Example Input	Expected Output
s1, s2 = 'abc', 'xyz'	'xyc abz'

Example Input	Expected Output
`s1, s2 = 'magic', 'carpet'`	`'cagic marpet'`

In [ ]:

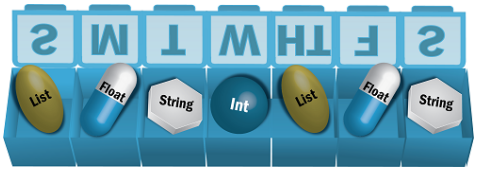
```
## YOUR CODE GOES HERE ##
```

In [ ]:

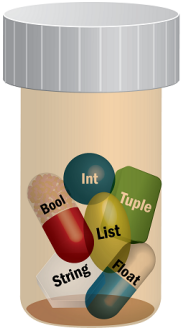
```
## INSTRUCTION SOLUTION(S) ##
s1, s2 = 'abc', 'xyz'

output_string = s2[:2] + s1[2:] + ' ' + s1[:2] + s2[2:]
output_string
```

8.5.5. Unordered Collections

Sequences are like pill boxes. They contain pieces of data in a given order, where each item is given a numeric index.



Unordered collections, on the other hand, are like pill bottles. While they also contain data elements, there is not an inherent order or addressing system.



8.5.6. Sets

Python sets are mutable, unordered collections of unique elements. They are not considered sequences and therefore do not support indexing or slicing. Sets also do not allow duplicate elements and will automatically deduplicate themselves, retaining only one instance of each unique value. Basic uses for sets include membership testing and deduplication of other sequences. Sets also support mathematical set operations like union, intersection, difference, and symmetric difference.

Sets are always wrapped in curly braces: `{}`

Syntax	Action
`set1 = set()`	Initializes an empty set
`set3 = {1, 'a', 3.1415}`	Initializes a set with multiple elements
`set2.add(x)`	Adds the value stored in `x` to `set2`
`set2.remove(x)`	Removes the value stored in `x` from `set2`
`x in set2`	Asks is `x` in `set2` ?
`set1.difference(set2)`	Values in `set1` but not in `set2`
`set1.union(set2)`	Values in either `set1` or `set2`
`set1.intersection(set2)`	Values in both `set1` and `set2`
`set1.symmetric_difference(set2)`	Values in `set1` or `set2` but not both

Instructor Guidance: Work through any questions about Python sets in the code cells below. Be sure to demonstrate the initialization of an empty set below.

To demonstrate these capabilities, first initialize values into two sets. Both sets contain geocoordinates as tuples. Use set operations to compare them.

In [ ]:

```
set1 = {(40.3359, -73.8519), (40.1395, -73.1847), (40.9018, -73.6472), (41.3114, -74.9317
set2 = {(40.7464, -73.3113), (40.4534, -73.5302), (40.1395, -73.1847), (40.9401, -73.6553
```

Show values in `set1` that are not in `set2` .

In [ ]:

```
set1.difference(set2)
```

Show values in either `set1` or `set2` .

In [ ]:

```
set1.union(set2)
```

Show the elements that belong to both sets.

In [ ]:

```
set1.intersection(set2)
```

Show values in `set1` or `set2` , but not both.

In [ ]:

```
set1.symmetric_difference(set2)
```

8.5.7. Dictionaries

Dictionaries are collections of key-value pairs. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type. Strings and numbers can always be keys. The values in a dictionary can be any data type or data structure.

Dictionaries share many properties with sets:

- Both sets and dictionaries are unordered
- Both sets and dictionaries are wrapped in curly braces
- Sets must have unique entries, dictionaries must have unique keys

There are three dictionary methods we covered in this class.

Code	Action
`.keys()`	Returns a collection of the dictionary keys
`.values()`	Returns a collection of the dictionary values
`.items()`	Returns a collection of tuples of key-value pairs that can be unpacked using multiple assignments

There are a number of ways to iterate through a dictionary, but using `.items()` is usually the easiest.

```
for key, value in dict_object.items():
    do_something_with(key)
    do_something_with(value)
```

Instructor Guidance: Work through any questions about Python dictionaries in the code cells below.

Initialize a dictionary to get some practice.

In [ ]:

```
inventory = {'apples': 430,
             'bananas': 312,
             'oranges': 525,
             'pears': 217}
```

Show the dictionary keys.

In [ ]:

```
inventory.keys()
```

Print the current stock for bananas.

In [ ]:

```
inventory['bananas']
```

Add `'avocados'` to your dictionary with a value of `543`.

In [ ]:

```
inventory['avocados'] = 543
inventory
```

Edit the value of `'oranges'` by subtracting `83` from the current value.

In [ ]:

```
inventory['oranges'] -= 83
inventory
```

Delete `'pears'` from your dictionary.

In [ ]:

```
del inventory['pears']
inventory
```

Using the provided dictionary, let's make some changes.

In [ ]:

```
grocery_prices = {'Eggs': 2.59,
                  'Milk': 3.19,
                  'Cheese': {'American': 4.80,
                             'Swiss': 4.11,
                             'Cheddar': 3.59
                            },
                  'Yogurt': {'Regular': 2.35,
                             'Sale': 1.11
                            },
                  'Butter': 2.59
                 }
```

Exercise: Egg prices have dropped. Let's change the price of `'Eggs'` to `1.99`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_prices['Eggs'] = 1.99
grocery_prices
```

Exercise: Change the `'Sale'` price of `'Yogurt'` to `1.49`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_prices['Yogurt']['Sale'] = 1.49
grocery_prices
```

Exercise: Add `'Honey'` to our dictionary, with the price of `3.49`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_prices['Honey'] = 3.49
grocery_prices
```

Exercise: We recently lost our butter supplier. Remove `'Butter'` from our dictionary.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
del grocery_prices['Butter']
grocery_prices
```

Exercise: Add `'Lobster'` to our dictionary with the value `'Market Price'`.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
grocery_prices['Lobster'] = 'Market Price'
grocery_prices
```

Exercise: We found that Swiss cheese is not selling. Remove `'Swiss'` cheese from our dictionary.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
del grocery_prices['Cheese']['Swiss']
grocery_prices
```

8.5.8. Common Operations Across Data Structures

There are certain operations you can do on any data structure. When called on dictionaries, these functions apply themselves to the dictionary's keys.

Syntax	Description
`max(my_sequence)`	Returns the maximum value contained in the collection
`min(my_sequence)`	Returns the minimum value contained in the collection
`sum(my_sequence)`	Returns the sum of the collection as an integer (only works if all items are numeric)
`len(my_sequence)`	Returns the length of the collection as an integer
`sorted(my_sequence)`	Returns a sorted copy of the collection

8.6. Flow Control

8.6.1. If Statements

The `if` statement is used for conditional execution. The below code checks the value of `stock_price` and prints a message depending on its value.

In [ ]:

```
stock_price = 14000

if stock_price > 20000:
    print('SELL!')

elif stock_price < 15000:
    print('BUY!')

else:
    print('Do Nothing.')
```

We can add as many `elif` statements as we want to an `if` block. But remember, only one clause in an `if` block will execute. When a clause evaluates to `True`, its code is run and all subsequent clauses are skipped.

In [ ]:

```
grade = 86

if 60 <= grade < 70:
    print('Student got a D')

elif 70 <= grade < 80:
    print('Student got a C')

elif 80 <= grade < 90:
    print('Student got a B')

elif grade >= 90:
    print('Student got a A')

else:
    print('Student did not pass')
```

Exercise: Create an `if` block (using `if`, `elif`, and/or `else` statements as needed) that will evaluate as follows.

Example Input	Expected Output
`weather = 'sunny'` `day = 'weekday'`	`'Have lunch outside.'`
`weather = 'rainy'` `day = 'weekday'`	`'Sigh and eat at your desk.'`
`weather = 'sunny'` `day = 'weekend'`	`'Go for a hike.'`

Example Input	Expected Output
`weather = 'rainy'` `day = 'weekend'`	`'Play board games with friends.'`

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
weather = 'sunny'
day_of_week = 'weekday'

if weather.lower() == 'sunny':

    if day_of_week.lower() == 'weekday':
        print('Have lunch outside.')
    elif day_of_week.lower() == 'weekend':
        print('Go on a hike.')
    else:
        print('Invalid input...')

elif weather.lower() == 'rainy':

    if day_of_week.lower() == 'weekday':
        print('Sigh and eat at your desk.')
    elif day_of_week.lower() == 'weekend':
        print('Play board games with friends.')
    else:
        print('Invalid input...')

else:
    if day_of_week.lower() == 'weekday':
        print('Go to work.')
    elif day_of_week.lower() == 'weekend':
        print('Stay home and read.')
    else:
        print('Invalid input...')
```

8.6.2. For Loops

The `for` statement allows you to loop over the elements of a collection. See the simple example below.

In [ ]:

```
letter_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',

for letter in letter_list:
    print(letter.lower() * 5)
```

Exercise: Write a `for` loop that will loop through the list of letters below and print only the vowels.

In [ ]:

```
letter_list = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
vowels = ['A', 'E', 'I', 'O', 'U']

for letter in letter_list:

    if letter in vowels:
        print(letter)
```

Exercise: Create a `for` loop that will print only the names of animals from `my_animals` that eat `'Shrubbery'`.

In [ ]:

```
my_animals = [{'Name': 'Giraffe', 'Weight': '2600', 'Diet': 'Shrubbery', 'Average Lifespa
              {'Name': 'Moose', 'Weight': '1300', 'Diet': 'Shrubbery', 'Average Lifespan'
              {'Name': 'Giant Panda', 'Weight': '220', 'Diet': 'Bamboo', 'Average Lifespa
              {'Name': 'Red Panda', 'Weight': '14', 'Diet': 'Bamboo', 'Average Lifespan':
              {'Name': 'Hippopotamus', 'Weight': '3800', 'Diet': 'Grass', 'Average Lifesp
              {'Name': 'Lion', 'Weight': '400', 'Diet': 'Gazelle', 'Average Lifespan': '1
              {'Name': 'Tortoise', 'Weight': '100', 'Diet': 'Grass', 'Average Lifespan':
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
for animal in my_animals:

    if animal['Diet'] == 'Shrubbery':
        print(animal['Name'])
```

Exercise: Using the same list of animals from the exercise above, loop and create a new dictionary that contains the diet food as the key, and the count of animals that have that diet as the value.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
diet_dict = {}

for animal in my_animals:
    food = animal['Diet']
    if food in diet_dict.keys():
        diet_dict[food] += 1
    else:
        diet_dict[food] = 1

diet_dict
```

Exercise: Given a list of numbers sorted in ascending order, count how many numbers in the list are less than `1000` . HINT: the answer is 98.

In [ ]:

```
nums_list = [16, 46, 48, 50, 78, 94, 96, 105, 109, 110, 115, 141, 141, 152, 154, 155, 180
             238, 262, 278, 283, 295, 304, 310, 322, 322, 334, 352, 357, 371, 383, 388, 3
             465, 480, 485, 493, 506, 514, 519, 537, 539, 553, 555, 567, 572, 600, 612, 6
             653, 660, 674, 695, 709, 710, 714, 729, 746, 763, 769, 783, 788, 793, 821, 8
             875, 891, 899, 904, 915, 916, 926, 968, 969, 973, 980, 982, 986, 1003, 1017,
             1071, 1073, 1090, 1096, 1105, 1124, 1142, 1148, 1155, 1157, 1160, 1187, 1198
             1258, 1267, 1272, 1272, 1289, 1309, 1319, 1321, 1330, 1352, 1352, 1363, 1371
             1439, 1446, 1447, 1454, 1486, 1492, 1495, 1505, 1509]
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
count = 0
for num in nums_list:
    if num < 1000:
        count += 1

count
```

8.6.3. While Loops

The `while` loop repeats a statement or group of statements while a given Boolean test expression evaluates to `True`. Here's a simple example. We initialize our `count` variable as `0`. Then we jump into a `while` loop that checks if `count` is not equal to `10`. If that evaluates to `True`, our code prints the value of `count` to the screen and then increments `count` by `1`. Try to guess what the output will look like before running the cell below.

In [ ]:

```
count = 0

while count != 10:
    print(count)
    count += 1
```

8.6.4. Break and Continue

The `break` statement in Python terminates the `for` or `while` loop it is contained in. The `continue` statement tells Python to skip any code following it in the current loop, and begin the next loop iteration.

Exercise: Given a list of numbers sorted in ascending order, count how many numbers in the list are less than `1000`. Add a `break` statement to halt the execution of your loop once you reach a number that is greater or equal to `1000`. (The series is increasing, there's no need to keep checking numbers after that.) HINT: the answer is 98.

In [ ]:

```
nums_list = [16, 46, 48, 50, 78, 94, 96, 105, 109, 110, 115, 141, 141, 152, 154, 155, 180
             238, 262, 278, 283, 295, 304, 310, 322, 322, 334, 352, 357, 371, 383, 388, 3
             465, 480, 485, 493, 506, 514, 519, 537, 539, 553, 555, 567, 572, 600, 612, 6
             653, 660, 674, 695, 709, 710, 714, 729, 746, 763, 769, 783, 788, 793, 821, 8
             875, 891, 899, 904, 915, 916, 926, 968, 969, 973, 980, 982, 986, 1003, 1017,
             1071, 1073, 1090, 1096, 1105, 1124, 1142, 1148, 1155, 1157, 1160, 1187, 1198
             1258, 1267, 1272, 1272, 1289, 1309, 1319, 1321, 1330, 1352, 1352, 1363, 1371
             1439, 1446, 1447, 1454, 1486, 1492, 1495, 1505, 1509]
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
count = 0

for num in nums_list:

    if num < 1000:
        count += 1

    else:
        break

count
```

Exercise: You are given a list of codes that have a mix of alphabetical, numerical, and special characters. From this list, create a new list of only the codes where every character is alphabetical. Use `continue` to skip items that fail to meet the condition. HINT: For each item in the list, use the string method `.isalpha()` to determine if every character is alphabetical.

Expected Output: `['gmY', 'GOx', 'tma', 'NQh', 'lgd']`

In [ ]:

```
codes_list = ['hQ ', '~$"', 'x%[', '#^l', 'N]\x0c', 'Yd\\', '.D>', '/{$', '_d]', '|0-', '
              'l@G', 'GOx', 'X"#', '1>G', 'P1#', 'f+0', 'tma', 'A<|', '-D]', '?u{', ' TW'
              '[8|', '~r+', '6@J', 'O=4', '{,]', '\t+w', 'z9!', 'NQh', ',H/', 'E5s', 'lgd
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
new_list = []

for item in codes_list:

    if not item.isalpha():
        continue

    new_list.append(item)

new_list
```

8.7. Getting Data

8.7.1. Loading a Local Data File

Another method for getting data into a Python program involves reading a local file using the built-in `open()` function. Use the `with` keyword to store the open file in a temporary variable. To read a file, the mode is `'r'` (but this is the default value so we need not specify). To write a file, the mode is `'w'`. In the following example, we read in some data and store it in our script. Remember that `.read()` always brings data in as a string.

In [ ]:

```
with open('data/earthquake/USGS_earthquake_data.xml') as my_file:
    USGS_xml_data = my_file.read()

USGS_xml_data
```

Exercise: Read in the file `'data/lesson_7/kermit.txt'` and find how many times the word `'Muppets'` appears in the file.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/lesson_7/kermit.txt') as f:
    text = f.read()

text.lower().count('muppets')
```

8.7.2. Getting Data with the CSV Library

The comma separated values (CSV) file format is a common import and export format. The `.DictReader()` object interprets CSV data as a list of dictionaries. And the `.DictWriter()` object is used to write CSV files from lists of dictionaries.

In [ ]:

```
import csv
```

In [ ]:

```
with open('data/animal/animal_info_land.csv') as f:
    reader_obj = csv.DictReader(f)
    data = list(reader_obj)

data
```

Exercise: Find how many rows are in the CSV data that we read in one cell above this (using code, not your counting skills!).

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(data)
```

Exercise: Find how many columns are in the CSV data we read in above (i.e. how many items are in each row).

HINT: Every row contains the same number of items.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(data[0])
```

Exercise: Add the following dictionary in the cell below to the CSV data we read in above. Then write your modified data structure out to a new file using `.DictWriter()` in the following location.

`data/animal/new_animals.csv`

In [ ]:

```
new_row = {'Name': 'Mouse', 'Weight': 1, 'Diet': 'Cheese', 'Average Lifespan': 5}
```

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
data.append(new_row)

with open('data/animal/new_animals.csv', 'w', newline='') as f:
    col_names = ['Name', 'Weight', 'Diet', 'Average Lifespan']
    writer = csv.DictWriter(f, col_names)
    writer.writeheader()
    writer.writerows(data)
```

8.7.3. The Glob Library

The Glob Library finds all the filepaths matching a specified pattern.

In [ ]:

```
import glob
```

In [ ]:

```
files = glob.glob('data/animal/*.csv')
files
```

Exercise: Using `.glob()` find all the CSV files from the `data/animal/` folder that include the word `info` in their filename. Then read all those files and put all the data into a single data structure.

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
files = glob.glob('data/animal/*info*.csv')

animal_info = []
for file in files:
    with open(file) as f:
        reader = csv.DictReader(f)
        data = list(reader)
        animal_info.extend(data)

animal_info
```

Exercise: How many unique animals are in the data set you created in the exercise above? HINT: The answer is `13` .

In [ ]:

```
## YOUR CODE GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
names = set()
for row in animal_info:
    names.add(row['Name'])

len(names)
```