NATIONAL GEOSPATIAL-INTELLIGENCE COLLEGE
FUNDAMENTALS OF **PROBLEM SOLVING** USING **PYTHON I**
HDN

Lesson 4: Ordered Data Structures

Table of Contents

4.1. 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

---

4.2. Overview

The following material is divided into the following parts:

- Lesson
- Guided Exercise
- Practical Exercises

Instructor Guidance: Refer back to Lesson 1 and relate the four steps of problem-solving using Computational Thinking (Decomposition, Pattern Recognition, Abstraction, & Algorithm Design) to lessons, exercises, examples, student questions/comments, etc., as appropriate throughout this lesson.

4.3. Review

4.3.1. Boolean Expressions

Exercise: Write one boolean expression that evaluates to `True` and one that evaluates to `False`. Try to use a combination of comparison, membership, and logical operators.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
bool_1 = 3+4 > 5
bool_2 = 'h' in 'Hello!'

bool_1, bool_2
```

4.3.2. If Statements

Exercise: What will the value of `my_val` be after the code below is run?

In [ ]:

```
my_val = 'test string'

if None == 0:
    my_val = 'zero'

elif my_val[0] == 'T':
    my_val = 'one'

elif my_val[-1] == 'e':
    my_val = 'two'

elif my_val.lower() == my_val:

    if ' ' in my_val:
        my_val = my_val.upper()

    else:
        my_val = my_val * 2

else:
    my_val = 'three'
```

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## no code required ##

## `my_val` will be equal to ``'TEST STRING'`` because the first nested `if` statement chan
## show this by running the cell above and then in a new cell type `my_val` and run it
```

Exercise: Use an `if` block to build a password checker. In the first cell below, create a username and

password and store them in two variables. In the second cell below, create two more variables to hold your username and password "attempts", and build an `if` block to test if it is the correct login. If the guess is correct, inform the user they have "unlocked" the account. If not, inform the user whether it is the username or the password that is incorrect.

In [ ]:

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

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
un = 'coolguy1234'
pw = 'password1234'
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
login_un = 'coolguy1234'
login_pw = 'password1234'

if login_un != un:
    print('Incorrect Username. Please try again.')

elif login_pw != pw:
    print('Incorrect Password. Please try again.')

else:
    print('Successful login!')
```

4.3.3. String Manipulations

Exercise: Take the string below and complete the following tasks:

1. Make it all lowercase. HINT: `.lower()`
2. Remove all the `$` characters. HINT: `.replace()`
3. Separate the string into two sentences. HINT: `.split()`
4. Get the word count of each sentence. HINT: `.split()` and `len()`

In [ ]:

```
practice_string = """Hi, I'm a $stude$nt in CSCI201$1. $I lov$e $python."""
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
practice_string = practice_string.lower()
print('lowercase: ' + practice_string)

practice_string = practice_string.replace('$', '')
print('remove $ characters: ' + practice_string)

print('first sentence len: '+ str(len(practice_string.split('.')[0].split(' '))))
print('second sentence len: '+ str(len(practice_string.split('.')[1].split(' '))))
```

4.3.4. Numbers Review

Exercise: Take the following two strings, convert them into number data types, and then use floor division.

In [ ]:

```
first = '23.5214'
second = '2'
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
first = '23.5214'
second = '2'

float(first) // int(second)
```

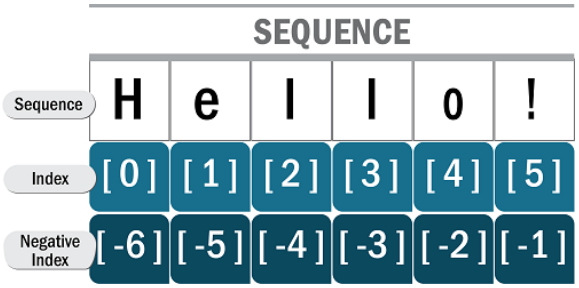
4.4. Lesson: Ordered Data Structures

4.4.1. Sequences

References:

- Python: Data Structures (https://docs.python.org/3.4/tutorial/datastructures.html#data-structures)

A sequence in Python is an ordered grouping of elements. Each element in a sequence is assigned an index, which is a number that marks its place in the sequence. The first item in a Python sequence has index `[0]`, the second has index `[1]`, and so on. Note that Python supports negative indexing too, with the last element in the sequence having the index `[-1]`. That means each element in a sequence can be accessed with two indexes, a positive one and a negative one.



4.4.2. Lists

References:

- Python: More on Lists (https://docs.python.org/3.4/tutorial/datastructures.html#more-on-lists)
- Effbot: An Introduction to Python Lists (http://www.effbot.org/zone/python-list.htm)

Lists are containers that hold objects in a given order. Lists are *mutable*, meaning they allow you to add, remove, or overwrite individual elements inside them. In Python, lists are always wrapped in square brackets: `[]`.

4.4.2.1. Creating Lists

To initialize or create an empty list, we can use the built-in `list()` function. Or we can simply use a set of empty square brackets.

In [ ]:

```
empty_list_1 = list()
empty_list_1
```

In [ ]:

```
empty_list_2 = []
empty_list_2
```

You can also create a list with elements in it. Just wrap the items, separated by commas, in a set of brackets. Lists can contain elements of any data type.

In [ ]:

```
numbers_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
numbers_list
```

In [ ]:

```
str_list = ['Talk', 'is', 'cheap.',
            'Show', 'me', 'the', 'code.',
            ' - Linus Torvalds']
str_list
```

Lists can also contain other collections, such as lists. Lists stored inside other lists are often called *nested* lists.

In [ ]:

```
list_of_lists = ['a', 'b', 'c', ['d', 'e', 'f']]
list_of_lists
```

Exercise: Create a list and store it in a variable.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
new_list = [1, 4, 16, 32]
new_list
```

4.4.2.2. List Methods and Operations

There are a number of methods (i.e. functions) that can be applied to lists. Let's initialize a list of agencies in the Intelligence Community to explore some of these methods.

In [ ]:

```
ic_agencies = ['CIA', 'DIA', 'NSA', 'NRO', '25AF', 'MI', 'MCIA', 'ONI',
               'OICI', 'I&A', 'CGI', 'FBI', 'ONSI', 'INR', 'TFI']
ic_agencies
```

Writing a set of brackets containing an integer after a list or list variable will access an element from the list. The integer inside the brackets refers to the index location of the item to access. This is sometimes referred to as *bracket notation*. You can also use an equals sign (=) to overwrite the element at that location with a new value.

Syntax	Description
`list_variable[n]`	Accesses the item at index `[n]`
`list_variable[n] = x`	Sets the item at index `[n]` to `x`

In [ ]:

```
ic_agencies[4]
```

In [ ]:

```
ic_agencies[4] = 'AF-25'
ic_agencies
```

The `.append()` method adds an item to the end of a list. The `.extend()` method is similar to `.append()`, but instead of adding a single element to a list, it adds multiple items, one by one. Because `.extend()` expects to add multiple items, you must pass it a sequence (such as a list) as an argument. The `.insert()` method also adds an item to a list, but it adds it in a position we specify. The new item will be placed at the index we put in the first argument, moving all later elements in the list to make room.

Syntax	Description
`list_variable.append(x)`	Adds `x` to the end of the list
`list_variable.extend(my_list)`	Appends each item in `my_list` to the list.
`list_variable.insert(i, x)`	Inserts `x` at index `i` in the list

In [ ]:

```
ic_agencies.append('DI')
ic_agencies
```

In [ ]:

```
ic_agencies.extend(['MI5','MI6'])
ic_agencies
```

In [ ]:

```
ic_agencies.insert(0, 'NGA')
ic_agencies
```

While the `.append()` method and `.extend()` method are very similar, they are also very different. Look at the examples below and note how the two work differently.

In [ ]:

```
ic_agencies.append(['IB','PET'])
ic_agencies
```

In [ ]:

```
ic_agencies.extend('CSIS')
ic_agencies
```

As we see here, `.append()` will always add the argument to the list as a single item, even if your argument is a list of things. On the other hand, `.extend()` will always treat the argument as a sequence of things and try to add each individual item to the list.

The `.remove()` method searches a list for a value specified and removes the first instance of that value from the collection. The `.pop()` method also removes an item from the list. But instead of specifying a value, we have to give it an index location. Furthermore, `.pop()` removes the item but also *returns* it, meaning that it is made available to be, for instance, stored in a variable.

Syntax	Description
`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 item

Let's clean up the mistake we made above when using the `.extend()` method with a single item ('CSIS'). We'll have to remove each letter one by one.

In [ ]:

```
ic_agencies.remove('C')
ic_agencies.remove('S')
ic_agencies.remove('I')
ic_agencies
```

Notice that we still need to remove the second 'S'. Why? We already asked it to remove 'S'. What is causing this behavior?

In [ ]:

```
ic_agencies.remove('S')
ic_agencies
```

In [ ]:

```
item = ic_agencies.pop(-1)
item
```

In [ ]:

```
ic_agencies
```

The `.index()` method searches a list for a given value, and returns the index position of the first instance of

the value. If the value is not in the list, this method throws an error. The `.count()` method counts the number of times a value appears in a list.

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

In [ ]:

```
ic_agencies.index('FBI')
```

In [ ]:

```
ic_agencies.count('ONI')
```

The `.sort()` method sorts a list. By default, it sorts the list in ascending order (least to greatest), however, to sort in descending order pass it the optional `reverse=True` argument.

Syntax	Description
`list_variable.sort(reverse=False)`	Sorts the list

In [ ]:

```
ic_agencies.sort()
ic_agencies
```

In [ ]:

```
ic_agencies.sort(reverse=True)
ic_agencies
```

Exercise: Use list methods on the list `ic_agencies` to remove `'CGI'` and add `'ATF'` to the end of the list.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ic_agencies.remove('CGI')
ic_agencies.append('ATF')

ic_agencies
```

Exercise: Create a list of fruits containing the following fruits: apple, banana.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits = ['apple', 'banana']
```

Exercise: Extend the fruits list by the second list of fruits.

In [ ]:

```
fruits2 = ['kiwi', 'mango']
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits.extend(fruits2)
```

Exercise: Append another type of fruit to the list.

In [ ]:

```
## YOUR SOLUTION GOES HERE ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits.append('orange')
```

Exercise: Sort the fruits list reverse-alphabetically.

In [ ]:

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

In [ ]:

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

4.4.3. Tuples

References:

- Python: Tuples and Sequences (https://docs.python.org/3.4/tutorial/datastructures.html#tuples-and-sequences)
- TutorialsPoint: Python - Tuples (https://www.tutorialspoint.com/python/python_tuples.htm)

Tuples are immutable sequences. Like all sequences, items in a tuple are accessible by a numeric index. Like all immutable objects, tuples will not allow you to add, subtract, or edit items without overwriting the entire structure. This makes tuples nice for storing data that you don't want to be edited piecemeal, such as coordinate pairs or days of the week. If there's something you don't want to change, making that fact explicit by not allowing it to change is best practice. Tuples are always wrapped in parentheses: `()`.

4.4.3.1. Creating Tuples

We can create tuples by wrapping two or more pieces of data of any type in a set of parentheses, separated by commas. Here, we create a tuple containing days of the week.

In [ ]:

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

Notice what happens if we try to modify one of the elements of a tuple. Python throws an error because tuples are immutable.

In [ ]:

```
multi_element_tuple[0] = 'Lunes'
```

Exercise: Create a tuple and store it in a variable.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
t = (1, 2, 3)
t
```

Exercise: Create a tuple of information about Ron Swanson from Parks and Rec with his first and last name, age, job title, marital status, and email, and store it in a variable.

- remember to use different data types

Name	Age	Position	Married?	email
Ron Swanson	45	Parks Director	Yes	ron.swanson@parks.gov (mailto:ron.swanson@parks.gov)

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ron = ('Ron', "Swanson", 45, 'Parks Director', True, 'ron.swanson@parks.gov')
```

Exercise: Create another tuple for Ron Swanson's coworker, Leslie Knope, and store it in a variable. Then, add these two tuples into a list.

Name	Age	Position	Married?	email
Leslie Knope	35	Deputy Director	Yes	leslie.knope@parks.gov (mailto:leslie.knope@parks.gov)

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
parksdept = []
parksdept.append(ron)
parksdept.append(('Leslie', 'Knope', 35, 'Deputy Director', True, 'leslie.knope@parks.gov
parksdept
```

4.4.3.2. Mutable vs. Immutable

Before we move on, it is important to discuss mutability. Not all Python objects handle changes the same way.

- A *mutable* object can be altered without overwriting the entire structure
- An *immutable* object cannot be altered without overwriting the entire structure

You can make changes to individual pieces of of mutable objects, such as lists. In contrast, the only way to change immutable objects is to overwrite the entire structure. The code below shows how you would change the second item in a list (mutable) versus a tuple (immutable). You can overwrite the individual item in the list, but you need to overwrite the entire tuple to change a single item.

Mutability of Common Types

Immutable	Mutable
int	list
float	dictionary
bool	set
string	

Immutable	Mutable
tuple	

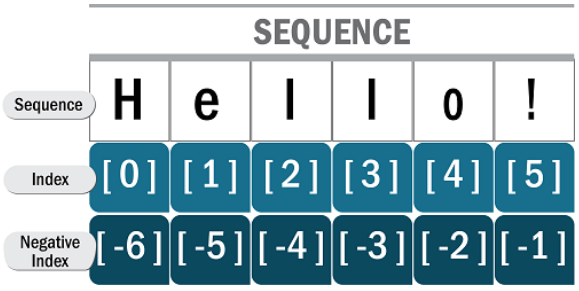
In [ ]:

```
my_tuple = (11, 15, 2018)
my_tuple = (11, 30, 2018)

my_tuple
```

In [ ]:

```
my_list = [11, 15, 2018]
my_list[1] = 30

my_list
```

4.4.4. Indexing and Slicing



4.4.4.1. Indexing

Recall that we can use numeric indexes to access individual items and subsets of strings. We can do the same with lists and tuples. You can access elements of any sequence by location using *bracket notation*. Remember in Python we start counting from zero.

Syntax	Description
`sequence[i]`	Returns the element at index `i`

In [ ]:

```
sample_list = [11, 14, 6, 2, 7, 5, 1, 12, 15, 10]
sample_tuple = (11, 14, 6, 2, 7, 5, 1, 12, 15, 10)
```

In [ ]:

```
sample_list[0]
```

In [ ]:

```
sample_tuple[1]
```

In [ ]:

```
sample_tuple[-1]
```

Exercise: Access the 5th item in `sample_tuple` using bracket notation.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
sample_tuple[4]
```

4.4.4.2. Slicing

Slicing is similar to indexing except you are accessing multiple elements instead of a single element. A slice can be of any size from zero elements up to the length of the entire collection, and we create one by specifying a start and stop index.

Syntax	Description
`sequence[start:stop]`	Returns the elements from `start` up to but not including `stop`

If we leave either index out of a slice, Python will interpret that as a slice including all elements before or after a certain spot. The two examples below illustrate this.

In [ ]:

```
sample_list[:5]
```

In [ ]:

```
sample_tuple[5:]
```

Exercise: Print the last 3 items in `sample_list` using slicing.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
sample_list[-3:]
```

4.4.5. Common Operations Across Data Structures

There are certain operations you can do on any collection of data we cover in this course. For example, Python has built-in functions for finding the number of elements in a collection and for finding its largest and smallest elements.

The `max()` function will take any collection and return the maximum (largest) value it contains. If your collection has mixed types (e.g. integers and strings), this function will throw an error.
The `min()` function will take any collection and return the minimum (smallest) value it contains. If your collection has mixed types (e.g. integers and strings), this function will also throw an error.
The `sum()` function finds the sum of the elements of a collection. All the members of the collection must be numeric.
The `len()` function finds the number of items in a collection.
The `sorted()` function is like the `.sort()` list method, but it works with any data structure. It takes the collection, sorts the elements, and gives you back a new list.

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

In [ ]:

```
my_tuple = (168, 162, 182, 180, 163, 192, 167, 199, 194, 187, 187, 178)

max(my_tuple)
```

In [ ]:

```
min(my_tuple)
```

In [ ]:

```
my_list = [45, 42, 57, 55, 46, 60, 50, 44, 40, 43, 52, 54]

sum(my_list)
```

In [ ]:

```
sorted(my_list)
```

In [ ]:

```
my_string = '38.876096, -77.072946'

len(my_string)
```

Exercise: Find the length of `my_list`. Then, find the sum of all the elements in `my_tuple`.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(my_list), sum(my_tuple)
```

4.5. Guided Exercise: Employee Hours

You manage a team of employees, and store vital business data in Python data structures. For example, you have a list of employees and how many hours they have worked so far this year. That list is below.

In [ ]:

```
hours = [('Akua', 1717), ('Rasel', 1540), ('James', 1654), ('Oralie', 1759), ('Emilia', 1
```

Problem 1: You want to sum up the total number of hours your team has worked this year. This means you need to access the hour numbers. Access the hours for the first employee in the list.

In [ ]:

```
hours[0][1]
```

Problem 2: Create an empty list to hold on to all the hours numbers. Then, add the number of hours for each employee to the list.

In [ ]:

```
hrs = []

hrs.append(hours[0][1])
hrs.append(hours[1][1])
hrs.append(hours[2][1])
hrs.append(hours[3][1])
hrs.append(hours[4][1])

hrs
```

Problem 3: Use the `sum()` function to find the total number of hours your team worked.

In [ ]:

```
sum(hrs)
```

Problem 4: Use the `len()` function to count how many employees are on your team, and use that number and the total number of hours you found in problem 4 to calculate the average number of hours worked per employee.

In [ ]:

```
sum(hrs) / len(hrs)
```

Problem 5: The company is going through some employee turnover. Rasel and Emilia have quit and need to be debriefed. Create a new list to hold the names of employees who need debriefing, then remove them from the current employee list and add them to the debriefing list.

In [ ]:

```
debrief = []

emp1 = hours.pop(1)
debrief.append(emp1[0])

emp2 = hours.pop(-1)
debrief.append(emp2[0])

debrief
```

4.6. Practical Exercises

4.6.1. Practical Exercise 1: Sequence Questions

Problem 1: List the types of sequences

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## strings, lists, tuples
```

Problem 2: What is an index? What is a negative index? When would you use indexes?

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## - index is a number telling the position where an element of a sequence is
## - negative index is the number position as well but starting count from the end of the
## - use them in order to get a certain element of a sequence or in slicing
```

Problem 3: The code cell below contains an example of slicing. In your own words, what is slicing?

In [ ]:

```
sample_list = [7975, 1177, 5255, 1455, 4475, 9296, 1345, 3370, 9372, 8403]

sample_list[2:5]
```

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## slicing is taking a portion of a sequence
```

Problem 4: What are the actions you can do with lists?

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTOR SOLUTIONS ##
## adding items (just one item or list of items), removing items, turning it into a strin
## of an item, and find the index where an item first appears
```

Problem 5: In your own words, what is the difference between the `.append()` and `.extend()` methods?

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## append adds one item to the end of a list
## extend adds an entire list of items individually to the list
```

Problem 6: In your own words, what is the difference between immutable and mutable?

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## immutable -- cannot be changed, only overwritten
## mutable -- can be edited and portions can be altered
```

Problem 7: What data types can lists and tuples hold?

In [ ]:

```
## YOUR ANSWER GOES HERE -- NO CODE NECESSARY ##
```

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## any/all data types
```

4.6.2. Practical Exercise 2: Practice Lists

Problem 1: Create two lists of strings and save them to variables.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list_1 = ['hey', 'how', 'are', 'you?']
my_list_2 = ['I', 'am', 'good']
```

Problem 2: Use list methods to combine the lists of strings from above into one big list of strings.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list_1.extend(my_list_2)
my_list_1
```

Problem 3: How many elements are in the combined list you created above?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
len(my_list_1)
```

Problem 4: Retrieve one item from the list using its index location.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list_1[2]
```

Problem 5: Change one item in the list using its index location and assignment.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list_1[0] = 'Hey'
```

Problem 6: Now that we have a list of strings, combine that into one string with spaces in between each word.

HINT: Refer to Lesson 2 String Methods

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
" ".join(my_list_1)
```

4.6.3. Practical Exercise 3: Practice Tuples

Problem 1: It's common to store coordinates in tuples because they are made up of two elements: a latitude and a longitude. How many coordinates are in the list below? Don't count them: use `len()` to find the answer.

In [ ]:

```
coords = [(37.08, 73.03), (37.04, 73.08), (37.02, 73.06), (37.06, 73.05),
          (37.07, 73.08), (37.04, 73.03), (37.02, 73.03), (37.06, 73.05),
          (37.02, 73.02), (37.08, 73.02), (37.06, 73.09), (37.02, 73.05)]
```

In [ ]:

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

In [ ]:

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

Problem 2: Create your own coordinate and store it in a tuple. Then, add your new coordinate to the list of coordinates above.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
c = (15.5, 82.1)
coords.append(c)
coords
```

Problem 3: Dates are also made up of multiple elements (year, month, day). Thus, tuples are good data structures to store date information. Below is a tuple that holds a date. Create your own tuple to store a different date, storing the year as the first element, the month as the second, and the day as the third.

In [ ]:

```
date1 = (2019, 6, 15)
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date2 = (2018, 1, 1)
```

Problem 4: Try to change an element of your tuple using the same approach you used to change an element in your list in Practical Exercise 2. What issues do you run into?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date1[2] = 16
# Error saying that 'tuple' object does not support item assignment
```

4.6.4. Practical Exercise 4: Negative List Index

Write a script that takes a list and an index (int) as inputs and outputs the corresponding negative index.

HINT: The negative index will depend on the list's length.

Bonus: Before you find the negative index, first check if the input index is valid. If not, print a helpful error message.

Example Input	Expected Output

Example Input	Expected Output
my_list = ['a','b','c','d','e','f'] i = 2	-4
my_list = ['a','b','c','d','e','f','g','h'] i = 7	-1
my_list = ['a','b','c','d','e','f','g','h','i','j'] i = 0	-10
my_list = ['a','b','c','d'] i = 6	'Invalid index'

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list = [1, 2, 3, 4, 5, 6]
i = 2

output = -(len(my_list) - i)

## BONUS SOLUTION ##
if 0 <= i < len(my_list):
    output = -(len(my_list) - i)
else:
    output = 'Invalid index'

output
```

4.6.5. Practical Exercise 5: Dates and Tuples

Problem 1: Write some code to determine whether two tuples that hold date information come from the same year. Assume the year is the first element in the tuple.

Example Input	Expected Output
date1 = (2019, 6, 15) date2 = (2018, 1, 1)	False
date1 = (2019, 6, 15) date2 = (2019, 1, 1)	True
date1 = (1965, 7, 27) date2 = (1965, 7, 3)	True

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date1 = (1965, 7, 27)
date2 = (1965, 7, 3)

date1[0] == date2[0]
```

Problem 2: Extend the code you wrote above to print the number of months between the two dates if the dates come from the same year (simply find the difference between the two month numbers). If the dates are from different years, print a message saying that.

Example Input	Expected Output
`date1 = (2019, 6, 15)` `date2 = (2018, 1, 1)`	`different years`
`date1 = (2019, 6, 15)` `date2 = (2019, 1, 1)`	`5`
`date1 = (1965, 7, 27)` `date2 = (1965, 7, 3)`	`0`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
date1 = (2019, 6, 15)
date2 = (2018, 1, 1)

if date1[0] == date2[0]:
    print(date1[1] - date2[1])
else:
    print('different years')
```

4.7. Appendix

List Methods and Operations

Syntax	Description	Example
`list_variable[n]`	Accesses the item at index `[n]`	>> strings_list = ['hey','class'] >> strings_list[1] 'class'

Syntax	Description	Example
`list_variable[n] = x`	Sets the item at index `[n]` to `x`	>> strings_list[1] = 'hi' >> strings_list ['hi', 'class']
`list_variable.append(x)`	Adds `x` to the end of the list	>> strings_list.append('python') >> strings_list ['hi','class', 'python']
`list_variable.extend(my_list)`	Appends each item in `my_list` to a list.	>> strings_list.extend(['is','is','fun']) >> strings_list ['hi', 'class', 'python', 'is', 'is', 'fun']
`list_variable.insert(i, x)`	Inserts `x` at index `i` in the list	>> string_list.insert(2, '!') >> strings_list ['hi', 'class', '!', 'python', 'is', 'is', 'fun']
`list_variable.remove(x)`	Removes the first appearance of `x` in the list	>> strings_list.remove('class') >> strings_list ['hi', '!', 'python', 'is', 'is', 'fun']
`list_variable.pop(i)`	Removes the item at index `i` in the list and returns that item	>> strings_list.pop(1) '!' >> strings_list ['hi', 'python', 'is', 'is', 'fun']
`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	>> strings_list.index('fun') 4
`list_variable.count(x)`	Counts the number of times `x` appears in the list	>> strings_list.count('is') 2
`list_variable.sort(reverse=False)`	Sorts the list	>> strings_list.sort(reverse=False) >> strings_list ['fun', 'hi', 'is', 'is', 'python']

Mutability of Common Types

Immutable	Mutable
int	list
float	dictionary
bool	set
string	
tuple	

Indexing and Slicing Any Sequence

Syntax	Description
`sequence[i]`	Returns the element at index `i`

Syntax	Description
`sequence[start:stop]`	Returns the elements from `start` up to but not including `stop`

Functions for Any Data Structure

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