NATIONAL GEOSPATIAL-INTELLIGENCE COLLEGE

FUNDAMENTALS OF **PROBLEM SOLVING** USING **PYTHON I**

HDN

Lesson 5: Unordered Data Structures

Table of Contents

5.1 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

---

5.2. 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.**

5.3. Review

5.3.1. Lists

Exercise: Create a list of at least 5 elements, then print the last three elements using slicing.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list = [5, 7, 23, 65, 8584, 53, 2345, 23452]
my_list[-3:]
```

Exercise: Describe the difference between the `.append()` and `.extend()` list functions. Then write some code to show how they each work using an example.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## if you append a list with another list, the last element in the list that got appended
## be a list, but if you extend a list with another list, then the list that got extended
## every individual element added to the end of the list

my_list.append([2,4,5])
print(my_list)
my_list.extend([2,4,5])
print(my_list)
```

Exercise: Overwrite the second element, fourth element, and last element of the list you created in the first part of this exercise. You may overwrite the second, fourth, and last elements with whatever data value or data structure you want.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list[1] = 'word'
my_list[3] = 'word'
my_list[-1] = 'word'
my_list
```

5.3.2. Tuples

Exercise: The tuple below has some misspelled objects. You can’t overwrite them individually because tuples are immutable, but you can overwrite the entire structure. Create a new tuple with the correct spellings and save it to the same variable as the one below.

In [ ]:

```
my_tuple = ('Monday', 'Tuesay', 'Wednesda', 'Thrsday', 'Friyay')
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_tuple = ('Monday', 'Tuesay', 'Wednesda', 'Thrsday', 'Friyay')
my_tuple = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
my_tuple
```

Exercise: What are some examples of data that would be good to store in a tuple? No code is required for this exercise.

In [ ]:

```
## YOUR ANSWER GOES HERE ##
```

5.3.3. Boolean Conditions

Exercise: Write a boolean condition that evaluates to `True` and uses a mix of conditional, membership, and logical operators.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
2 != 0 and 2 not in [1,4,5,7,8]
```

Exercise: Write a boolean condition that evaluates to `False` and uses a mix of conditional, membership, and logical operators.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
2 == 0 or 2 in [1,4,5,7,8]
```

5.3.4. If Statements

Exercise: Run the cell below and debug the output. Is the answer what you expected? What could be wrong with the syntax of the boolean expression?

In [ ]:

```
coord = '357682W'
if 'N' or 'S' in coord:
    print('There IS an "N" or "S" in the coordinate:', coord)
else:
    print('There IS NOT an "N" or "S" in the coordinate:', coord)
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
# The output is the string 'N' because it is not a complete boolean
# expression. There needs to be a complete boolean on either side of
# the `or` operator.
```

Exercise: Create an `if` block that uses the `if` , `elif` , and `else` keywords.

Bonus: Use multiple different operators (membership, conditional, logical)

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
weather = '23'

if float(weather) < 30:
    print('stay inside')
elif float(weather) > 30 and float(weather) < 60:
    print('wear layers')
else:
    print('enjoy!')
```

Exercise: Create an `if` block that checks the weight of your luggage, and if it is greater than 50 pounds tells you to remove something. For this example, your "luggage" will be represented by the list of weights below.

HINT: the total weight of your "luggage" is the `sum()` of all of the individual weights.

In [ ]:

```
luggage = [5, 1.2, 2.5, 7, 3, 2.5, 9.3, 1, 2, 6.1, 5.9, 1.3, 4.2, 8.1, 6.7, 1.5, 0.8]
```

In [ ]:

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

In [ ]:

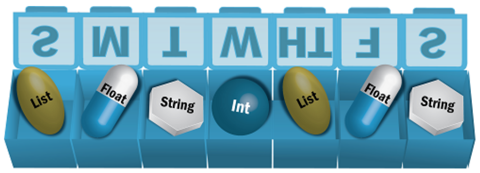
```
## INSTRUCTION SOLUTION(S) ##
weight = sum(luggage)

if weight > 50:
    item = luggage.pop(-1)
    print('Removed an item that weighed', item, 'pounds! Try again.')
else:
    print("Your luggage is under 50 pounds. You're good to go!")
```

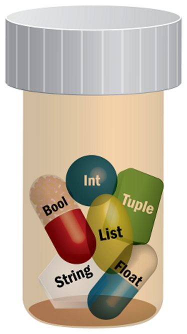
5.4. Lesson: Data Structures

5.4.1. 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.



5.4.2. Sets

References:

- [Python: Sets (https://docs.python.org/3.4/tutorial/datastructures.html#sets)](https://docs.python.org/3.4/tutorial/datastructures.html#sets)

Sets are mutable, unordered collections of unique elements. Like all unordered collections, sets are not considered sequences and do not support indexing or slicing. Sets also support mathematical operations like union, intersection, difference, and symmetric difference. Basic uses of sets include membership testing and deduplication. Sets are always wrapped in curly braces: `{}`

5.4.2.1. Creating Sets

There's only one way to create an empty set, and that is to call the `set()` function with no arguments. To create a set with elements in it, wrap multiple elements in curly braces.

In [ ]:

```
my_set = set()
my_set
```

In [ ]:

```
my_set = {1, 'a', 3.1415}
my_set
```

One special feature of sets is that they automatically drop duplicate values. All values in a set must be unique, so Python strips out unneeded duplicates. Elements of a set must also be immutable.

In [ ]:

```
my_set = {1, 1, 1, 3, 4, 5}
my_set
```

A common way to create a set is to cast a different type of collection. Remember, this deduplicates the collection automatically.

In [ ]:

```
my_list = [14, 7, 12, 8, 12, 5, 7, 12, 5, 13, 13, 11, 14, 12, 9]

my_set = set(my_list)
my_set
```

Exercise: Create a set with the names of your favorite TV shows.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fav_shows = {'Friends', 'Parks and Recreation', 'Scandal'}
```

Exercise: Create a set using casting and store it in a variable.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_list = [1, 1, 2, 3]
my_set = set(my_list)
my_set
```

5.4.2.2. Set Methods and Operations

To add elements to a set, use the `.add()` method. To remove individual elements from a set, use the `.remove()` method.

Syntax	Description
`set_variable.add(x)`	Adds `x` to the set
`set_variable.remove(x)`	Removes `x` from the set
`set_variable.pop()`	Removes and returns a random item from the set

In [ ]:

```
my_set.add(297)
my_set
```

In [ ]:

```
my_set.remove(297)
my_set
```

Calling `.pop()` on a set will return an item at random. Because sets don't have an index, you can't use an argument to tell Python which item to get.

In [ ]:

```
item = my_set.pop()
print(item, my_set)
```

We can test for membership with the `in` keyword.

In [ ]:

```
297 in my_set
```

Exercise: Add another TV show to the set of favorite TV shows and delete another one.

In [ ]:

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

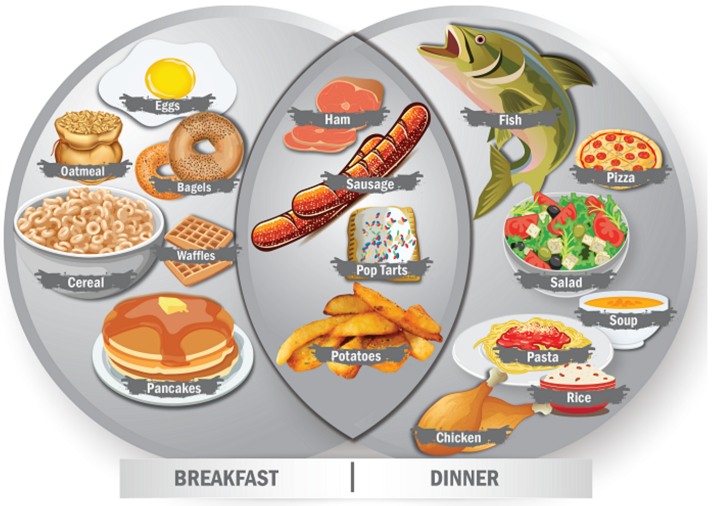
In [ ]:

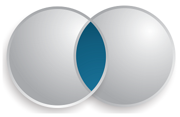
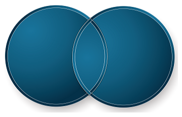
```
## INSTRUCTION SOLUTION(S) ##
fav_shows.add('The Office')
fav_shows
```

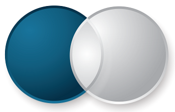
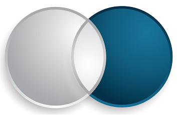
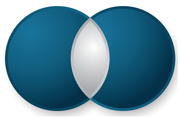
In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fav_shows.remove('Scandal')
fav_shows
```

Sets in Python support set-specific mathematical operations. You may be familiar with set arithmetic from working with Venn diagrams in math class. We sometimes refer to set operations as *cross referencing*.



Code	Mathematical Operation	Description	Graphical Representation
`a.intersection(b)`	Intersection	Returns elements that appear in both set `a` and set `b`	
`a.union(b)`	Union	Returns elements in either set `a` or set `b`	

Code	Mathematical Operation	Description	Graphical Representation
`a.difference(b)`	Left Set Difference	Returns elements from set `a` that are not in set `b`	
`b.difference(a)`	Right Set Difference	Returns elements from set `b` that are not in set `a`	
`a.symmetric_difference(b)`	Symmetric Set Difference	Returns elements in set `a` or set `b`, but not in both	

As an example, let's look at two sets of employee names. Who still needs to complete mandatory training? To find out, we could take the set difference.

In [ ]:

```
team_members = {'Smith', 'Daniels', 'Robertson', 'Chen', 'Winters', 'Baker', 'Johnson', '

completed_mandatory_training = {'Winters', 'Robertson', 'Smith'}
```

In [ ]:

```
team_members.difference(completed_mandatory_training)
```

Exercise: Talk to your neighbor, discuss how sets can be useful in the work that you do.

In [ ]:

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

Exercise: Create a set of your neighbors favorite TV shows.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
neigh_shows = {'Scandal', "That 70's Show", 'The Office'}
```

Exercise: Check if you have any TV shows in common with your neighbor.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fav_shows.intersection(neigh_shows)
```

Exercise: Make a set of recommendations for your instructors by creating a combined set of all the items in your and your neighbor's favorite shows set.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
recs = fav_shows.union(neigh_shows)
recs
```

Exercise: Take the set of the recommendations and convert it into a list.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
list(recs)
```

5.4.3. Dictionaries

References:

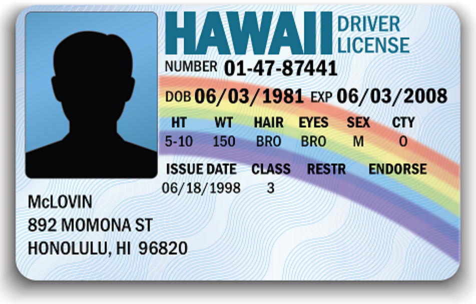
- Python: Dictionaries (https://docs.python.org/3.4/tutorial/datastructures.html#dictionaries)

Dictionaries are unordered collections of key-value pairs. Unlike sequences, which are indexed, dictionaries are accessible by their keys, which can be of any immutable type. Strings, numbers, and tuples can all serve as keys in a dictionary.

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

A driver's license is a great example of a common object in our daily lives that uses key-value pairs. Each data point is linked to a key that describes what it is.



5.4.3.1. Creating Dictionaries

Like sets, dictionaries use curly braces as wrappers. However, a set of empty curly braces gets interpreted as an empty dictionary, not a set. We can also set up a dictionary that's populated with entries. Note that each entry in the dictionary below is a pair of values separated by a colon (`:`). To the left of the colon is the entry's *key*, and to the right is the *value*. Dictionary entries are always structured like this: `key: value` , with commas separating each `key: value` pair.

In [ ]:

```
empty_dictionary = {}
empty_dictionary
```

In [ ]:

```
english_french_dictionary = {'hello': 'bonjour',
                             'world': 'monde',
                             'thank you': 'merci'}
english_french_dictionary
```

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

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
d = {'one': 1, 'two': 2, 'three': 3}
```

Exercise: Create an empty dictionary.

In [ ]:

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

In [ ]:

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

5.4.3.2. Dict Methods and Operations

To access a value inside a dictionary, we use bracket notation as we would for a list, tuple, or string. But instead of putting a numeric index inside the brackets, we put a key. Dictionaries are mutable, so we can change individual values without overwriting the entire data structure.

Syntax	Description
`dict_variable[key]`	Accesses the item at key `[key]`
`dict_variable[key] = x`	Sets the item at key `[key]` to `x`

> NOTE: Dictionary keys can be any immutable data type.

Below, we access the value stored at the key `'hello'` in our `english_french_dictionary` .

In [ ]:

```
english_french_dictionary['hello']
```

Below, we add an entry to our dictionary using bracket notation and assignment.

In [ ]:

```
english_french_dictionary['good bye'] = 'au revoir'
english_french_dictionary
```

Instructor Guidance: Adding new values to a list is a little easier to understand than dictionary value assignment via a new key. Emphasize that the `dictionary_name[key]` syntax is really just a fancy variable name for the value that you've assigned to that key. Sometimes it helps to think of dictionaries as just groups of variables. This is a good place to get the students to start thinking about the "layered" approach, and considering what value/type they're working with at each point in a formula.

We can also overwrite an existing value in a dictionary using the same syntax as above.

In [ ]:

```
english_french_dictionary['hello'] = ['bonjour', 'salut']
english_french_dictionary
```

Deleting a key-value pair from a dictionary requires the `del` keyword. Be careful, as deletions using `del` permanently wipe the entry from memory.

Syntax	Description
`del dict_variable[n]`	Delete the entry at key `[n]`

In [ ]:

```
del english_french_dictionary['good bye']

english_french_dictionary
```

We can quickly glance through all of the keys in a dictionary using the `.keys()` method. Additionally, we can look at the values in a dictionary using the `.values()` method. These methods return list-like objects.

Syntax	Description
`dict_variable.keys()`	Return a list-like object containing all keys
`dict_variable.values()`	Return a list-like object containing all values

In [ ]:

```
english_french_dictionary.keys()
```

In [ ]:

```
english_french_dictionary.values()
```

Exercise: Access the value at the key `'thank you'` in `english_french_dictionary` .

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
english_french_dictionary['thank you']
```

Exercise: Add the following entry to `english_french_dictionary` : `{'learn': 'apprendre'}`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
english_french_dictionary['learn'] = 'apprendre'
english_french_dictionary
```

Exercise: Delete a key-value pair from the dictionary.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
del english_french_dictionary['learn']
english_french_dictionary
```

5.4.4. Common Operations Across Data Structures

The functions in the table below also apply to sets and dictionaries. When called on dictionaries, these functions consider only the keys. Because of this, calling `max()` on a dictionary would give you its largest (highest value) key. Keep in mind also that `sorted()` will always return a list, even if you call it on a set or dictionary.

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 number of elements in the collection
`sorted(collection)`	Returns a sorted list of the collections elements

In [ ]:

```
my_set = {'F15', 'P17', 'T18', 'E14', 'A18', 'D13', 'D19', 'Z16', 'K17', 'X16', 'V13', 'U

max(my_set)
```

In [ ]:

```
my_set = {'F15', 'P17', 'T18', 'E14', 'A18', 'D13', 'D19', 'Z16', 'K17', 'X16', 'V13', 'U

min(my_set)
```

In [ ]:

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

sum(my_list)
```

In [ ]:

```
my_dict = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9

len(my_dict)
```

In [ ]:

```
my_dict = {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6, 'H': 7, 'I': 8, 'J': 9

sorted(my_dict, reverse=True)
```

Exercise: Find the sum of all the values in `my_dict`. HINT: You need to pull out the values before calling the `sum()` function.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
sum(my_dict.values())
```

5.5. Guided Exercise: Analyzing Crime Data

We've received some data about criminal activity, and have been tasked with doing some analysis of the data. The dataset is rather small, but we expect to be receiving similar reports on a recurring basis and will always want to perform the same analysis. This is a perfect case to write some Python code to do our analysis for us. The data is in the code cell below, and we want to answer the following questions:

1. What dates do we have data for?
2. How many crimes were reported on the first and last dates in the data set?
3. How many unique coordinates are there? Are there coordinates that appear more than once?
4. Did 11-10-2016 have more or less crimes than average for that day on record?

In [ ]:

```
crime_dates = {'11-07-2016': ['110716AT001', '110716AT010', '110716DN011', '110716DN014',
                              '110716NA039', '110716NA049', '110716NA051', '110716NA054',
               '11-08-2016': ['110816AT003', '110816AT006', '110816AT014', '110816DN004',
                              '110816NA028', '110816NA039', '110816NA040', '110816NA049',
               '11-09-2016': ['110916AT000', '110916DN009', '110916DN019', '110916DN021',
                              '110916NA001', '110916NA036', '110916NA046', '110916RB004',
               '11-10-2016': ['111016DN013', '111016DN015', '111016DN022', '111016DN036',
                              '111016NA029', '111016NA040', '111016RB010', '111016RB014',
               '11-11-2016': ['111116AT004', '111116DN008', '111116DN015', '111116DN030',
                              '111116NA031', '111116RB005', '111116RB012', '111116TF004',
              }
```

In [ ]:

```
crime_coords = {'110716AT001': (39.29, -76.65), '110716AT010': (39.29, -76.64), '110716DN
                '110716NA003': (39.29, -76.68), '110716NA023': (39.29, -76.68), '110716NA
                '110716RB004': (39.28, -76.65), '110716RB008': (39.30, -76.66), '110716TF
                '110816AT014': (39.29, -76.68), '110816DN004': (39.28, -76.64), '110816DN
                '110816NA028': (39.28, -76.65), '110816NA039': (39.29, -76.66), '110816NA
                '110816TF015': (39.29, -76.65), '110816TF032': (39.30, -76.67), '110816TF
                '110916DN019': (39.30, -76.64), '110916DN021': (39.30, -76.67), '110916DN
                '110916NA001': (39.30, -76.65), '110916NA036': (39.30, -76.65), '110916NA
                '110916TF013': (39.30, -76.64), '110916TF017': (39.29, -76.65), '110916TF
                '111016DN022': (39.29, -76.66), '111016DN036': (39.29, -76.64), '111016DN
                '111016NA029': (39.30, -76.65), '111016NA040': (39.28, -76.65), '111016RB
                '111016TF015': (39.30, -76.65), '111016TF024': (39.28, -76.64), '111016TF
                '111116DN030': (39.30, -76.65), '111116DN033': (39.28, -76.65), '111116DN
                '111116RB005': (39.28, -76.67), '111116RB012': (39.30, -76.67), '111116TF
               }
```

In [ ]:

```
historic_avg = {'11-07': 20, '11-08': 18, '11-09': 15, '11-10': 14, '11-11': 20}
```

Question 1: What dates do we have data for?

To answer this question, we simply need to access all the keys in `crime_dates` . The easiest way to do that is to use the `.keys()` method. Since dictionary keys are unordered and we always want to display dates in order, we can use `sorted()` to sort the dates from oldest to newest.

Pseudocode:

1. Access the keys in `crime_dates` (`.keys()`)
2. Display dates as a sorted list (`sorted()`)

In [ ]:

```python
dates = sorted(crime_dates.keys())
dates
```

Question 2: How many crimes were reported on the first and last dates in the data set?

To check how many incidents were reported on a given day, we can just check the length of the value associated with that day in `crime_dates`. To find the first and last dates, we can use indexing on our `dates` list.

Pseudocode:

1. Get the first and last dates from the list we created above
2. Access the values in `crime_dates` for each date
3. Find the length of the values for each date

In [ ]:

```python
first_date = dates[0]
last_date = dates[-1]

crimes_first = len(crime_dates[first_date])
crimes_last = len(crime_dates[last_date])

print('First date:', crimes_first)
print('Last date:', crimes_last)
```

Question 3: How many unique coordinates are there? Are there coordinates that appear more than once?

To answer this, we need to look at `crime_coords` dictionary. We know we can isolate the coordinates with `.values()`, but how do we find how many unique coordinates there are? Remember that sets only hold unique items, so casting the values to a set and then finding the length will answer the first question. After finding how many unique coordinates we have, we can compare that to the total number of coordinates to determine if there are any duplicates.

Pseudocode:

1. Extract the coordinates from `crime_coords` (`.values()`)
2. Cast the values to a set
3. Find the length of the set
4. Find the length of the entire list of coordinates, and compare

In [ ]:

```python
coords = crime_coords.values()

unique_coords = set(coords)

len(unique_coords)
```

In [ ]:

```python
len(coords)
```

Question 4: Did 11-10-2016 have more or less crimes than average for that day on record?

For this question, we need to check how many crimes were reported on the date in question. Then, we need to compare that number to the average number of crimes reported for that date in `historic_avg` .

Pseudocode:

1. Save the date `11-10-2016` as a string to a variable
2. Extract the month and year from the date (slicing)
3. Find the number of crimes reported on `11-10-2016` (`len()` on `crime_dates` values)
4. Find the average number of crimes reported on `11-10` in `historic_avg`
5. Compare the two values from steps 3 and 4 (`if` block)

In [ ]:

```python
date = '11-10-2016'

date_no_year = date[:5]

num_crimes = len(crime_dates[date])
avg_crimes = historic_avg[date_no_year]

if num_crimes < avg_crimes:
    print(date, 'had fewer crimes than average')
elif num_crimes > avg_crimes:
    print(date, 'had more crimes than average')
else:
    print(date, 'had an average number of crimes')
```

Instructor Guidance: Refer back to Lesson 1 and relate the four steps of problem-solving using Computational Thinking (Decomposition, Pattern Recognition, Abstraction, & Algorithm Design) as appropriate throughout these exercises.

Instructor Guidance: The practical exercises deemed most important due to content and/or a cumulative result, which should be completed first in the interest of maximum training value in relation to time are Practical Exercises 1-3. Ensure you go over the exercise solutions and (as necessary) the processes to arrive at the solutions with the students.

Instructor Guidance: Follow-up questions are designed to be asked by the facilitators individually as each student completes the task and has it looked at by a facilitator.

5.6. Practical Exercises

5.6.1. Practical Exercise 1: Knowledge Check: Sets

Problem 1: What are the differences between sets and lists? When would you use a set rather than a list?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## Sets are unique and unordered
## You would use sets when you want to be able to use specific set operations or when you
## collection to be unique and order doesn't matter
```

Problem 2: Jot down all the actions that you can do using sets.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
# 1) get all unique elements from a list
# 2) add elements to a set
# 3) remove elements from a set
# 4) get all unique elements from two sets
# 5) get the overlapping elements of two sets
# 6) get elements only contained within one list
```

5.6.2. Practical Exercise 2: Practice Sets

Problem 1: Create an empty set and save it to a variable called `fruits` .

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits = set()
```

Problem 2: Add three different fruits to the fruits set using `.add()` .

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits.add('strawberry')
fruits.add('banana')
fruits.add('kiwi')
```

Problem 3: Create a set with everyday groceries and save it to a variable called `groceries` .

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
groceries = {'bread', 'ketchup', 'tomato', 'potato', 'onion', 'spinach', 'banana', 'straw
```

Problem 4: Find the length of the grocery list.

In [ ]:

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

In [ ]:

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

Problem 5: Find all the items in your fruits set that are in the groceries set. That is, find the intersection of the two sets.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fruits.intersection(groceries)
```

Problem 6: Create the set of all the groceries that aren't in the fruits set. That is, find the difference between the groceries and the fruits.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
groceries.difference(fruits)
```

5.6.3 Practical Exercise 3: Knowledge Check: Dictionaries

Problem 1: In the broadest terms possible, what are dictionaries made up of?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
# key value pairs
```

Problem 2: In one dictionary, can we have multiple different keys with the same values? Why or why not?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## yes because values aren't compared to one another in dictionaries
```

Problem 3: In one dictionary, can there be duplicate keys linked to different values? Why or why not?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## no because keys are compared to one another and therefore must be unique
```

Problem 4: What kind of data types and data structures can be used as *values* in a key-value pair?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## all data types and data structures can be used as values
```

Problem 5: When applying common functions on a dictionary (such as the `sum()` function), is the function applied to the dictionary's keys or values?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
## it is applied on keys of a dictionary
```

5.6.4. Practical Exercise 4: Practice Dictionaries

Problem 1: Create a dictionary with 6 people and their ages, where the age is the key and the name is the value. Account for a few people having the same age.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ppl_dict = {33: 'Jack', 23: 'Christophe',
            21: 'Mel', 49: 'Mary', 54: ['Aiden', 'John']}
```

Problem 2: Add the following key-value pair to your dictionary: `{43: 'Laura'}`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ppl_dict[43] = 'Laura'
```

Problem 3: Print the name of the youngest person (or people) in the dictionary.

HINT: Try using the `min()` function on the dictionary and see what that gives you.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ppl_dict[min(ppl_dict)]
```

Problem 4: Print the name of the oldest person (or people) in the dictionary.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
ppl_dict[max(ppl_dict)]
```

5.6.5. Practical Exercise 5: Deduplicated List

Create a list of data, and be sure to include duplicates. Store your list in a variable. Then, write a script to create a list with all of the unique values from your list.

HINT: Cast to a set to remove duplicates.

In [ ]:

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

In [ ]:

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

uniques = list(set(my_list))
uniques
```

5.6.6. Practical Exercise 6: Employee Availability

Your employees have provided you a list of days that they are available to work; you've complied this data into the dictionary seen in the code block below.

Problem 1: Write a script that takes a name and a day as inputs, and outputs `True` if that employee can work on that day and `False` otherwise.

Example Input	Expected Output
`name = 'Lucy'` `day = 'Friday'`	`False`
`name = 'Tom'` `day = 'Monday'`	`True`
`name = 'Sara'` `day = 'Sunday'`	`False`
`name = 'Alex'` `day = 'Saturday'`	`True`

In [ ]:

```
availability = {'Alex': ['Monday', 'Tuesday', 'Thursday', 'Saturday'],
                'Apollo': ['Wednesday', 'Sunday'],
                'Cait': ['Monday', 'Tuesday', 'Wednesday', 'Friday', 'Saturday', 'Sunday'
                'Giulia': ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Satur
                'Jelle': ['Wednesday', 'Friday', 'Sunday'],
                'John': ['Tuesday', 'Wednesday', 'Friday', 'Sunday'],
                'Lasse': ['Monday', 'Thursday', 'Friday', 'Saturday', 'Sunday'],
                'Lucy': ['Monday', 'Tuesday'],
                'Nico': ['Monday', 'Tuesday', 'Wednesday', 'Sunday'],
                'Rachel': ['Monday', 'Wednesday', 'Thursday'],
                'Sara': ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'],
                'Shawn': ['Tuesday', 'Wednesday', 'Sunday'],
                'Tara': ['Friday'],
                'Tom': ['Monday', 'Thursday', 'Friday', 'Saturday', 'Sunday'],
                'Zack': ['Monday', 'Tuesday', 'Friday', 'Saturday']}
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
name = 'Tom'
day = 'Monday'

day in availability[name]
```

Problem 2: One of your employees calls in sick on Wednesday and you need to find an employee that is available to come in as soon as possible. Your order of preference is Tara, Lucy, Tom, Alex, Rachel, Sara, Zach, then after that no preference. Who is the most preferred employee available? Hint: Use your code from above, set the day equal to 'Wednesday'. Enter the names above in your name variable until you find someone who can work on Wednesdays.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
name = 'Rachel'
day = 'Wednesday'

day in availability[name]
```

5.6.7. [Challenge] Practical Exercise 7: Symmetric Difference

Write a script that does the same thing as `.symmetric_difference()` using only the other set methods from this lesson: `.union()` , `.intersection()` , and `.difference()` . HINT: The symmetric difference is equal to the difference between the union and the intersection, or union of the differences between each set and the other.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
set_a = {3, 4, 6, 8, 10, 125}
set_b = {1, 2, 3, 4, 6}

set_a.difference(set_b).union(set_b.difference(set_a))

## ALTERNATE SOLUTION(S) ##
set_a.union(set_b).difference(set_a.intersection(set_b))
```

5.7. Appendix

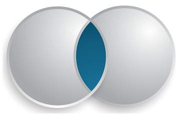
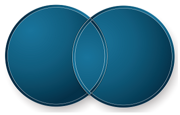
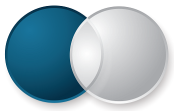
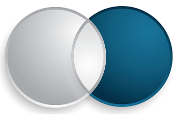
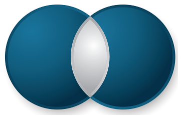
Indexing and Slicing Any Sequence

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

Set Methods and Operations

Syntax	Description
`set_variable.add(x)`	Adds `x` to the set
`set_variable.remove(x)`	Removes `x` from the set
`set_variable.pop()`	Removes and returns a random item from the set

Set Math Operations

Code	Mathematical Operation	Description	Graphical Representation
`a.intersection(b)`	Intersection	Returns elements that appear in both set `a` and set `b`	
`a.union(b)`	Union	Returns elements in either set `a` or set `b`	
`a.difference(b)`	Left Set Difference	Returns elements from set `a` that are not in set `b`	
`b.difference(a)`	Right Set Difference	Returns elements from set `b` that are not in set `a`	
`a.symmetric_difference(b)`	Symmetric Set Difference	Returns elements in set `a` or set `b`, but not in both	

Dictionary Methods and Operations

Syntax	Description
`dict_variable[key]`	Accesses the item at key `[key]`

Syntax	Description
`dict_variable[key] = x`	Sets the item at key `[key]` to `x`
`del dict_variable[n]`	Delete the entry at key `[n]`
`dict_variable.keys()`	Return a list-like object containing all keys
`dict_variable.values()`	Return a list-like object containing all values

Functions for Any Sequence

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