Lesson 7: Getting Data From Files

Table of Contents

7.1. 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

---

7.2. 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.

7.3. Review

7.3.1. For Loops

Exercise: Take the 'jumbled_list' below and separate the strings and integers into two new lists. One list will contain all the integers, and the other will contain all the strings. HINT: To check whether an item is a string, you can use the Boolean expression `type(item) == str` .

Bonus: Join the list of strings into a single string using the '.join()' method before you print it out.

In [ ]:

```
jumbled_list = ['2', 3, 5, 10, '8', 7, '0', 9, 13, '18', 99, 34, '72', 53, '28', 8, 14, '
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##

jumbled_list = ['2', 3, 5, 10, '8', 7, '0', 9, 13, '18', 99, 34, '72', 53, '28', 8, 14, '

string_list = []
int_list = []

for item in jumbled_list:

    if type(item) == str:
        string_list.append(item)
    else:
        int_list.append(item)

print(int_list)
print(string_list)

## BONUS SOLUTION(S) ##
''.join(string_list)
```

Exercise: Starting from the beginning of the list below, how many numbers do you need to sum up before the total is greater or equal to 5,000?

In [ ]:

```
count_data = (8, 11, 20, 25, 30, 44, 45, 49, 51, 56, 62, 66, 77, 78, 80, 84, 85, 108, 110
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
my_count = 0
my_sum = 0

for n in count_data:
    my_count += 1
    my_sum += n

    if my_sum >= 5000:
        break


print(my_count)
print(my_sum)

## ALTERNATE SOLUTION ##
ix = 0
total = 0

while total < 5000:
    current_num = count_data[ix]
    total += current_num

    ix += 1


print(ix)
print(total)
```

7.4. Lesson: Getting Data From Files

7.4.1. Reading and Writing Plain Text

References:

- Python 3.5 open() (https://docs.python.org/3.5/library/functions.html#open)
- Python Tutorial - Reading and Writing Files (https://docs.python.org/2/tutorial/inputoutput.html#reading-and-writing-files)

7.4.1.1. The `open()` Function

The `open()` function creates a connection between your Python script and a file. This connection is sometimes called a *file stream*. The primary argument `open()` takes is a string: the location of the file you want to connect to. Once a file stream is opened, it must be closed for the file to be saved and accessed later.

Python can connect to a file in various modes, the most basic of which are `'r'` (read mode) and `'w'` (write mode).

Reading from (or writing to) a local file consists of three basic steps: open, read/write, and close. Remembering to close the file is an easy step to skip. Luckily, we don't have to remember to close the file if we use the `with` and `as` keywords to store the file stream in a temporary variable. Our code to read or write data is indented under the `with` statement, and the file will close automatically when the indented code runs.

Below is an example of what a `with open()` block looks like.

```
with open('path/folder/filename.ext') as f:
    variable_name = f.read()
```

Note the slashes (`/`) separating the words in the `open()` function's argument. By default, Python will look for files in your current working folder—the folder where your script or notebook is stored. If you want to read or write to a file in a different folder, you must include a *path* to that folder. Most of the time, you'll use what's called the *relative path*. The relative path tells Python all the steps to take to get from the current working folder to the destination folder. Each step is separated by a slash (`/`) after each folder name.

- To read a file, mode is `'r'` and the method is `.read()` . (This is the default mode, so if you don't pass a `mode` argument, the mode is `'r'` .)
- To write a file, mode is `'w'` and the method is `.write()` .

7.4.1.2. Read a Text File

Let's read in Kermit the Frog's Resume from the file in the location below and print it to the screen. Since we want to open the file in read mode, we don't need to pass a mode argument.

`data/lesson_7/kermit.txt`

In [ ]:

```
with open('data/lesson_7/kermit.txt') as f:
    kermit_resume = f.read()

print(kermit_resume)
```

After the code indented under `with` is executed, the file stream is closed, so we can't read from it anymore. What happens if we try?

In [ ]:

```
f.read()
```

> NOTE: `I/O` stands for Input/Output.

Exercise: Using a `with open()` block and `.read()`, read in the following file, save its contents to a variable, and then print it to the screen.

`data/resumes/01.txt`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/resumes/01.txt') as f:
    resume = f.read()

resume
```

Now that you have your file's contents stored as a string, you can use all of the tools you learned in previous lessons to analyze the data using Python. Refer back to Lesson 2 for string methods.

Let's pull out the url from Kermit's resume that we read in above. To do so, we can search the string for the characters unique to the beginning of a url: `'http:'`.

In [ ]:

```
url_start = kermit_resume.find('http:')
url_start
```

In [ ]:

```
kermit_resume[url_start:url_start+50]
```

We've found the url that we're looking for, but now we need to find where the url ends. Looking closer, we can see that the end of the url is followed by a space. `.find()` has an optional second argument that specifies at what index in the string it should start looking, so we can tell Python to look for the first space in the string after the beginning of the url.

In [ ]:

```
url_end = kermit_resume.find(' ', url_start)
url_end
```

In [ ]:

```
kermit_resume[url_start:url_end]
```

7.4.1.3. Write a Text File

Now let's write a plain text file. By default, any new file will be created in your current working folder, the same folder that holds this Jupyter Notebook. But we can change that by adding a path with folder names and slashes (`/`) to the beginning of the file name. We need something to write to our new file, so let's just write the URL we found above.

> NOTE: When we open a file for the purpose of writing to it, we need to pass the `'w'` mode argument.

In [ ]:

```
url = kermit_resume[url_start:url_end]

with open('data/lesson_7/Kermit_URL.txt', 'w') as f:
    f.write(url)
```

> WARNING: If the file already exists, opening the file in `'w'` mode will clear its current contents.

Exercise: Write a file called `my_info.txt` to the `data/lesson_6/` folder. It should contain your name and your unclassified email address. (You can make up a fake name and email if you prefer.)

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
name = 'This is only a test!!!'
email = 'DrWho@myemail.com'
with open('data/lesson_7/my_info.txt','w') as f:
    f.write(name + '\n')
    f.write(email)
```

7.4.2. Getting Data with the CSV Library

References:

- Python 3.4: CSV File Reading and Writing (https://docs.python.org/3.4/library/csv.html)

Comma separated values (CSV) is a common format for storing data. CSV files are useful for storing tabular data—data that fits into a rows-and-columns structure—and can be read by Microsoft Excel. The CSV Library provides functions to both read and write CSV files in Python scripts.

In [ ]:

```
import csv
```

7.4.2.1. Using `.DictReader()` to Read a CSV File

In the CSV Library, the `.DictReader()` object interprets CSV data as a list of dictionaries. Look at the following three representations of the same data. The first is a table similar to what you would see if you were working in Excel, the second is plain text, and the third is a Python list of dictionaries. Despite their differences, these three representations contain the same information, and we can convert one to the other using Python tools.

Name	Weight	Diet	Average Lifespan
Giraffe	2600	Shrubbery	25
Moose	1300	Shrubbery	20
Giant Panda	220	Bamboo	20

```
csv_data = '''Name,Weight,Diet,Average Lifespan\n
              Giraffe,2600,Shrubbery,25\n
              Moose,1300,Shrubbery,20\n
              Giant Panda,220,Bamboo,20\n'''
```

```
csv_data = [{'Average Lifespan': '25',
             'Diet': 'Shrubbery',
             'Name': 'Giraffe',
             'Weight': '2600'},
            {'Average Lifespan': '20',
             'Diet': 'Shrubbery',
             'Name': 'Moose',
             'Weight': '1300'},
            {'Average Lifespan': '20',
             'Diet': 'Bamboo',
             'Name': 'Giant Panda',
             'Weight': '220'}]
```

The `.DictReader()` object formats CSV data as a list of dictionaries. The keys for each dictionary are the CSV data's column headers, which are read in from the first line in the file. The `.DictReader()` object is difficult to work with, so we want to convert it to a more familiar data structure. We can do this by simply casting the object to a list using the `list()` function. This will give us a list of dictionaries, a data structure we should be comfortable working with by now.

In [ ]:

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

data
```

> NOTE: In newer versions of Python, `.DictReader()` returns a list of `OrderedDict` objects rather than a list of standard dictionaries. No problem though — you can treat them both the same.

Now we can use the tools we've learned about in class to work with the data we just read in. Below, we extract the `Name` value from the second row in our table.

In [ ]:

```
data[1]['Name']
```

Exercise: Read the following CSV file using `.DictReader()`. Save all the rows to a new list.

`data/animal/animal_info_sea.csv`

Bonus: Combine this data with the data about land animals we read in above (i.e. create one data structure with the data from both files).

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/animal/animal_info_sea.csv') as csvfile:
    reader_obj = csv.DictReader(csvfile)
    seadata = list(reader_obj)

seadata.extend(data)
seadata
```

7.4.2.2. Using `.DictWriter()` to write a CSV File

Writer objects can take data and write it to a file as plain text. The `.DictWriter()` object is the tool in the CSV Library for writing CSV files. When we call `.DictWriter()`, we also need to pass it our data's column names. Also, unlike reading CSV data, we need to use separate methods for writing column headers

(`.writeheader()`) and data rows (`.writerows()`).

```
    with open(filepath, mode='w', newline='') as csvfile:

        column_headers = list_of_header_names

        writer = csv.DictWriter(csvfile, column_headers)

        writer.writeheader()

        writer.writerows(data_you_want_to_write_out)
```

Write the data you created in the exercise above to a CSV file using `.DictWriter()`, `.writeheader()`, and `.writerows()`

In [ ]:

```
with open('data/animal/animal_dict_output.csv', mode='w', newline='') as csvfile:

    column_headers = ['Name', 'Weight', 'Diet', 'Average Lifespan']

    writer = csv.DictWriter(csvfile, column_headers)

    writer.writeheader()

    writer.writerows(data)
```

7.4.3. The Glob Library: How to Find and Read Multiple Files

References:

- Python 3.4: glob (https://docs.python.org/3.4/library/glob.html)

We've shown you how to read data from a single file, but what if we wanted to read data from a thousand files? When reading files individually is out of the question, use the Glob Library. The Glob Library allows you to grab all of the file names that match a specified pattern. Once you have the file names that you want to read, you can use a `for` loop to read each one in. The pattern that we pass to `.glob()` is a string that contains wildcard characters. The wildcards can stand in for any character or substring, thus allowing us to grab many files without typing their names explicitly. Take a look at the example below. As always, our use of Glob requires us to first import the library.

In [ ]:

```
import glob
```

In [ ]:

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

Take note of the output. The asterisk (`*`) is a catch-all for any string of characters. We passed the `.glob()` method the pattern `'data/animal/*'`, which matches any file in the `data/animal/` folder. In return, we get a list of strings, each string corresponding to a file on our computer.

Discussion Questions

1. What is the `'*'`?
2. What type of data structure does glob return?
3. How many filenames does `.glob()` return?
4. What are the file types of the files that matched the pattern?
5. How could you use the above list of file names in a script?

To get more specific, we could make our pattern `'*.csv'`, which would only collect files with the `.csv` extension.

In [ ]:

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

> NOTE: The double backslash (`'\\'`) is the way `.glob()` separates folders and files when it runs in a Windows environment. When Python code runs on Windows computers, `'\\'` and `'/'` are interchangeable as path separators.

7.4.3.1. Combining the Glob and CSV Libraries

How do the Glob and CSV Libraries fit together? To read in data from a file, we pass that file's location to the `open()` function to create the connection between our script and the file. Then the data is available to read using one of the methods from this lesson. In other words, to read in data, all we need is a string that locates the file on your computer. Helpfully, that's exactly what `.glob()` produces. The only trick is that `.glob()` gives you a list of those strings, and you have to loop through them and read each file individually. This means taking all of the code we use to read in a CSV file and indenting it underneath a `for` statement. In each run of our loop, we'll read in a single file and add its contents to a larger data collection.

In [ ]:

```
files_list = glob.glob('data/fr24/UR82073_*.csv')
files_list
```

In [ ]:

```
all_data = []
for file in files_list:
    with open(file) as f:
        reader_obj = csv.DictReader(f)
        data = list(reader_obj)
        all_data.extend(data)

len(all_data)
all_data
```

Exercise: Read the contents of all the files that match the following pattern into a single list using `.glob()` and `.DictReader()` .

`data/fr24/RA64520_d*.csv`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
all_data2 = []

fileslist = glob.glob('data/fr24/UR82073_*.csv')

for file in fileslist:
    with open(file) as csvfile:
        reader_obj = csv.DictReader(csvfile)
        data = list(reader_obj)
        all_data2.extend(data)

all_data2
```

7.5. Guided Exercise: Data Exploration and Engineering

References:

- FlightRadar 24 (https://www.flightradar24.com)
- Wikipedia: Automatic Dependent Surveillance-Broadcast (ADS-B) (https://en.wikipedia.org/wiki/Automatic_dependent_surveillance_%E2%80%93_broadcast)
- Wikipedia: Volga-Dnepr Airlines (https://en.wikipedia.org/wiki/Volga-Dnepr_Airlines)
- Wikipedia: Antonov AN-124 Condor (https://en.wikipedia.org/wiki/Antonov_An-124_Ruslan)
- JetPhotos: Volga-Dnepr AN-124 RA-82074 (https://www.jetphotos.com/photo/8636024)

The following dataset is from the online flight tracking portal FlightRadar24, which leverages the data collected from beacons on commercial aircraft. The filenames are composed of an aircraft tail number and a flight ID. Our task is to clean and process this data for analysis. We'll develop a script using the following file as an example.

`data/fr24/RA64520_ccbc1f8.csv`

Pseudocode:

1. Read in the data
2. Split the data in the `Position` column into separate columns for latitude and longitude
3. Cast Latitude, Longitude, Altitude, Speed, and Direction data to numeric data types
4. Add a column to identify what file the data came from (there are many files in this data set)

Step 1: First, read the data in using the CSV Library.

Pseudocode:

1. Open the file (`with open()`)
2. Read in the data (`.DictReader()`)
3. Cast the data to a list and store in a variable

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
filename = 'data/fr24/RA64520_ccbc1f8.csv'

with open(filename) as f:
    track = list(csv.DictReader(f))

track
```

When processing a lot of data, sometimes it's helpful to test your code on a small part of that data. If the code works on a single piece of the data set, it will likely work on the rest. We have just created a list of dictionaries from the data in our file, so let's just take one of those dictionaries out and save it to a new variable.

In [ ]:

```
row = track[0]
row
```

Step 2: Split the data in the Position column into separate columns for latitude and longitude

Pseudocode:

1. Access the `Position` column
2. Split the data on the comma (`,`)
3. The first part of the split data goes to the `Latitude` column
4. The second part of the split data goes to the `Longitude` column

In [ ]:

```
lat_lon = row['Position'].split(',')
lat_lon
```

In [ ]:

```
row['Latitude'] = lat_lon[0]
row['Longitude'] = lat_lon[1]
row
```

Step 3: Cast Latitude, Longitude, Altitude, Speed, and Direction data to numeric data types

Pseudocode:

1. Collect all the column names that we want to affect
2. Use a `for` loop to cast each column's data to a float data type

In [ ]:

```
for col in ['Latitude', 'Longitude', 'Altitude', 'Speed', 'Direction']:
    row[col] = float(row[col])

row
```

The difference is subtle, but you can see that the data we cast is no longer wrapped in quotation marks. This tells us we now have numeric values, not strings.

Step 4: Add a column to identify what file the data came from

Pseudocode:

1. Remove the path and file extension from the file name
2. Create a new key-value pair where the key is `'ID'` and the value is the name of the file

In [ ]:

```
row['ID'] = filename[10:-4]
row
```

Now that we have all the steps to clean our data, let's put them all in one cell.

In [ ]:

```
row = track[0]

## Create Latitude and Longitude columns
lat_lon = row['Position'].split(',')
row['Latitude'] = lat_lon[0]
row['Longitude'] = lat_lon[1]

## Convert values to floats
for col in ['Latitude', 'Longitude', 'Altitude', 'Speed', 'Direction']:
    row[col] = float(row[col])

## Create ID column
row['ID'] = 'RA64520_ccbc1f8'
row
```

It looks like it's working, so let's indent all this code under a `for` loop so that it affects every row of data.

Pseudocode:

1. Read in the data
2. Clean each row in the data set (use the code we just wrote)

In [ ]:

```
filename = 'data/fr24/RA64520_ccbc1f8.csv'

## Open file
with open(filename) as f:

    ## Read in data and cast to list
    track = list(csv.DictReader(f))

    ## Clean each row
    for row in track:
        lat_lon = row['Position'].split(',')
        row['Latitude'] = lat_lon[0]
        row['Longitude'] = lat_lon[1]

        for col in ['Latitude', 'Longitude', 'Altitude', 'Speed', 'Direction']:
            row[col] = float(row[col])

        row['ID'] = filename[10:-4]

track
```

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, 2, 3, and 4. 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.

7.6. Practical Exercises

7.6.1. Practical Exercise 1: Use Glob to Create a List of Filenames

Problem 1: Use the Glob library to create pull in all of the filenames of the csv files in the `data/fr24/` folder.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
fr24_files = glob.glob('data/fr24/*.csv')
fr24_files
```

Problem 2: What type of data structure did you just create? Have you read in the contents of any of the files yet?

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
# It is a list of strings. The files have not been read in yet; these
# are just the filenames.
```

7.6.2. Practical Exercise 2: Read in Multiple Files in a Loop

Using the list of filenames that you created in Practical Exercise 1, iterate over that list and read in each file. Add all the data to a single data structure.

HINT: Keep in mind the difference between `.append()` and `.extend()` when you are adding to the data structure.

> WARNING: The data structure you create in this exercise is very large, and printing it in its entirety to the screen can cause Jupyter to crash. Print only small slices of the data structure or print its length (`len()`) to check your work. The length of your combined structure should be `550271` .

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
flighttracks = []
for filename in fr24_files:

    with open(filename) as csvfile:
        reader = csv.DictReader(csvfile)
        track = list(reader)

    flighttracks.extend(track)

len(flighttracks)
```

7.6.3. Practical Exercise 3: Analyze FR24 Data

Use the data structure you made in the exercise above to answer the following questions.

Problem 1: The `Callsign` key holds a value that identifies individual aircraft. How many data points are associated with each callsign? Create a new dictionary where the keys are the callsigns from the data set and the values are a count of how many rows contained that callsign. HINT: Use a `for` loop to loop through each row.

In [ ]:

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

In [ ]:

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

for track in flighttracks:

    current_callsign = track['Callsign']

    if current_callsign in callsigns:
        callsigns[current_callsign] += 1

    else:
        callsigns[current_callsign] = 1

callsigns
```

Problem 2: Using the dictionary that you just created, find the callsigns that appear in more than 1500 rows. Add those callsigns to a new list. How many are there? HINT: The answer is 39 (your list should be 40 items long, but one of the items is the empty string (`''`))

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
countlist = []
for callsign, num_rows in callsigns.items():
    if num_rows > 1500:
        countlist.append(callsign)
len(countlist)
```

7.6.4. Practical Exercise 4: Personnel Data

Problem 1: Read in the following file using the CSV Library.

`data/lesson_7/random_personnel.csv`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/lesson_7/random_personnel.csv') as csvfile:
    personnel = list(csv.DictReader(csvfile))
personnel
```

Problem 2: For each row in the data you just read in, create a new entry called `'Area Code'` that stores the 3-digit area code from the phone number in that row. HINT: Do not create an entirely new data structure for this exercise; we are simply adding a column to our existing CSV data.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##

ac_counts = {}

with open('data/lesson_7/random_personnel.csv') as csvfile:

    reader = csv.reader(csvfile)

    data = []

    for row in reader:
        data.append(row)

    for row in data[1:]:  ## the first row is the header
        phone_num = row[1]
        area_code = phone_num[1:4]
        if area_code in ac_counts:
            ac_counts[area_code] += 1
        else:
            ac_counts[area_code] = 1

ac_counts
```

7.6.5. Practical Exercise 5: Netflix Data

Problem 1: Read in the following file using the CSV Library.

`data/lesson_7/netflix.csv`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/lesson_7/netflix.csv') as csvfile:
    netflix = list(csv.DictReader(csvfile))

netflix
```

Problem 2: Create a new dictionary that maps a release year (e.g. `2004`) to the number of titles in the data set that were released in that year. The keys in your new dictionary should be the years, and the values should be the count of titles released in that year.

In [ ]:

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

In [ ]:

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

for movie in netflix:
    year = movie['release year']
    if year in netflix_dict.keys():
        netflix_dict[year] += 1
    else:
        netflix_dict[year] = 1

netflix_dict
```

7.6.6. Practical Exercise 6: Resume Analysis

Problem 1: Read in the following file using a `with open()` block and `.read()`.

`data/resumes/02.txt`

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
with open('data/resumes/02.txt') as f:
    resume = f.read()

resume
```

Problem 2: Search for the term `'analyst'` in the resume that you just read in. If it is in the resume, print `True`, otherwise print `False`.

HINT: Watch out for different casing of words! 'Analyst' may be in the resume, but not 'analyst'. Make use of `.lower()` or `.upper()`.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
if 'analyst' in resume.lower():
    print(True)
else:
    print(False)
```

Problem 3: Use glob to create a list of all the resume filenames in the folder `data/resumes/` and store it in a variable called `resume_filenames`.

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
resume_filenames = glob.glob('data/resumes/*')
resume_filenames
```

Problem 4: Now loop through the resume filenames list you just created. As you loop, create a new dictionary. The key for each entry should be the file name without the path and extension (e.g. `'01'` instead of `'data/resumes\\01.txt'`), and the value should be the full resume text.

Sample output:

```
{'01': 'Brielle Anastasia Ma...',
 '02': '1431 Nave\nCharles Tow...',
 '...'}
```

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
resume_dict = {}
for resume in resume_filenames:
    fname = resume[-6:-4]
    with open(resume) as f:
        fcontents = f.read()
    resume_dict[fname] = fcontents

print(resume_dict)
```

Problem 5: Set a variable equal to a search term. Then, by looping over the dictionary you just created (`.items()`), search for and print out the IDs of all the resumes containing the input keyword.

Example Input	Expected Output
TS/SCI	02, 03, 04, 07, 08, 09
deployed	01, 03, 05, 07, 08, 10
Army	03, 05, 07, 09, 10

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
search_term = 'Army'

res_list = []
for key, value in resume_dict.items():
    if search_term.lower() in value.lower():
        res_list.append(key)
print(res_list)
```

Problem 6: Edit the code from the last section to use a variable containing a string of multiple keywords, separated by commas. Print out resume names that contain all the keywords.

HINT: Strip whitespace from the beginning and end of each keyword.

Example Input	Expected Output
TS/SCI, Intelligence, Analyst, All Source	03, 04
Analyst, Army, Polygraph	05

In [ ]:

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

In [ ]:

```
## INSTRUCTION SOLUTION(S) ##
search_terms = 'Analyst, Army, Polygraph'
search_list = search_terms.split(', ')

res_list = []
for key, value in resume_dict.items():
    flag = True
    for search_term in search_list:
        if search_term.lower() not in value.lower():
            flag = False

    if flag:
        res_list.append(key)

print(res_list)
```

In [ ]:

```
## ALTERNATE INSTRUCTION SOLUTION(S) ##
search_terms = 'Analyst, Army, Polygraph'
search_list = search_terms.split(', ')

res_list = []
for key, value in resume_dict.items():
    terms_pres = []
    for search_term in search_list:
        if search_term.lower() in value.lower():
            terms_pres.append(search_term)

    if len(terms_pres) == len(search_list):
        res_list.append(key)

print(res_list)
```

7.7. Appendix

Working With Files

Syntax	Description
`with open(name) as f:`	Opens the file `name` and saves the file connection to `f`
`f.read()`	Reads the data in `f` as a string

Syntax	Description
`f.write(text)`	Writes the string `text` to `f`
`csv.DictReader(f)`	Reads CSV data stored in `f`, creates a list of dictionaries
`csv.DictWriter(f, cols)`	Creates writer object to write CSV data with columns `cols` to `f`
`writer.writeheader()`	Writes a header row to `f`
`writer.writerows(data)`	Writes CSV data to `f`
`glob.glob(file_pattern)`	Creates a list of every file path that matches the pattern; pattern usually contains wildcard characters (`*`)