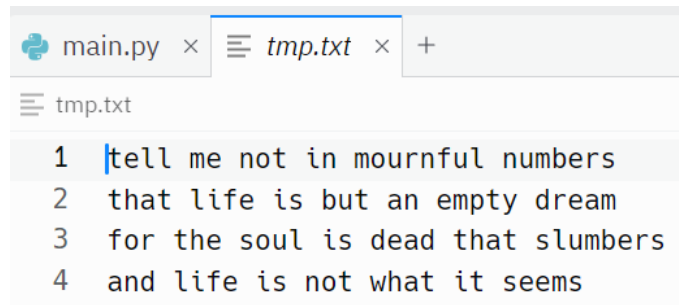


**PART 1: FILE FUNDAMENTALS.** The data structures in our program disappear after the program ends. But data stored in files can persist after the program closes. There are four basic file operations: open, read, write, and close. To gain access to a file, use `open()`. Do some reading to discover the various file types and modes. Consider the code to the right. Line 5 opens a file named "tmp.txt" for "writing". "tmp.txt" will be in the same folder as the source code. If another folder is desired, the path must be provided as part of the filename string. The file will be a text file by default so the "mode" is "write text". "write" mode will create a new file. If there is a file named "tmp.txt" in the current directory, it will be erased when we open the file for writing. The variable `f` is a file object. To continue to work with this file, we use file object methods. Line 6 uses the `write()` method to write the poem to the text file. Line 7 closes the file and assures all the data is written to it.

```
1 poem = """tell me not in mournful numbers
2 that life is but an empty dream
3 for the soul is dead that slumbers
4 and life is not what it seems"""
5 f = open('tmp.txt','w')
6 f.write(poem)
7 f.close()
```



Failure to close a file after writing can result in a corrupted file ... so it's important to always be mindful to do this. Python can automatically close a file when you're done with it if you use a "with" block. "with" is a construct called a "context manager". If you use it to open a file, Python will close the file when program flow leaves the block. Here's the same code using "with". To avoid file issues, I recommend always using the context manager. OK?

```
5 with open('tmp.txt','w') as f:
6     f.write(poem)
7     print("The file is closed.")
```

Reading data from a text file is a similar process. The code below reads back the poem and verifies the text ...

```
1 poem = """tell me not in mournful numbers
2 that life is but an empty dream
3 for the soul is dead that slumbers
4 and life is not what it seems"""
5 with open('tmp.txt','w') as f:
6     f.write(poem)
7 with open('tmp.txt','r') as f:
8     data=f.read()
9     print(data)
10    print(data==poem)
```

```
tell me not in mournful numbers
that life is but an empty dream
for the soul is dead that slumbers
and life is not what it seems
True
```

## Final Project: Acquiring and selecting data

So, that's basic text file handling. There's a lot more to know that you can read about but we're here to talk about data handling. Files are a way to get data into our programs and there are a couple different useful types of text file formats to explore. We've previously discussed how data is often communicated using a table consisting of columns and rows where each row is a "data record". Here, it may be helpful to use the "readlines" method (instead of read) because readlines() returns a list where each item is one line of the text file ...

```
7 with open('tmp.txt','r') as f:
8     data=f.readlines()
9     print(data,'\n')
10    print(*data)
```

```
['tell me not in mournful numbers \n', 'that life\n', 'is but an empty dream \n', 'for the soul is dead t\n', 'hat slumbers \n', 'and life is not what it seems']
```

```
tell me not in mournful numbers
that life is but an empty dream
for the soul is dead that slumbers
and life is not what it seems
```

PART 2: CSV FILES / WORLD CITIES. One important file format that specifies data in rows and columns is called CSV (comma separated values) format. Earlier in the semester, you did an exercise with a file containing world cities. Here's the code for reading that data ...

```
12 with open('worldcitiesF23.csv','r') as f:
13     data=f.readlines()
14     print(f'Loaded data for {len(data)} cities.')
15     print(*data[:10])
```

... and here's the result ...

```
Loaded data for 44692 cities.
city,lat,lng,country,code,pop
Tokyo,35.6897,139.6922,Japan,JPN,37732000
Jakarta,-6.175,106.8275,Indonesia,IDN,33756000
Delhi,28.61,77.23,India,IND,32226000
Guangzhou,23.13,113.26,China,CHN,26940000
Mumbai,19.0761,72.8775,India,IND,24973000
Manila,14.5958,120.9772,Philippines,PHL,24922000
Shanghai,31.1667,121.4667,China,CHN,24073000
Sao Paulo,-23.55,-46.6333,Brazil,BRA,23086000
Seoul,37.56,126.99,South Korea,KOR,23016000
```

... it's a big file containing data on over 44,000 cities. Note that the first line has column headings ... we could use those to create a dictionary. Note that the geographical coordinates for each city are given (lat=latitude and lng=longitude) ... those belong together ... maybe we could create a tuple and point to it with a key called "location". But before we jump into that, let's notice a few things. It may not be completely apparent, but there are some characters here that may cause us some problems later ...

- there's a newline character at the end of each record
- there's (apparently) a space character at the beginning of each record (except for the first)

## Final Project: Acquiring and selecting data

We can easily clean this up using a string method called `strip()` on each list item (line 14). In line 16, we need to add the newline to see the individual records.

Cleaning this data was pretty easy. That's not always the case. Let's move forward and look at some standard modules that may help us read and manipulate data easier.

There's a module called "csv" that can be helpful. It works a bit different than the standard file operations. The csv module understands that each line is composed of "data fields" and can make each line (data record) a list of fields. It uses a structure called a "reader" that iterates, line-by-line, through the file. Here's some code that uses the "reader". Note that the file must be open (inside the context manager) to use the reader. Line 22 is a list comprehension

```
12 with open('worldcitiesF23.csv','r') as f:
13     data=f.readlines()
14     data = [s.strip() for s in data]
15     print(f'Loaded data for {len(data)} cities.')
16     print(*data[:10],sep='\n')
```

```
Loaded data for 44692 cities.
city,lat,lng,country,code,pop
Tokyo,35.6897,139.6922,Japan,JPN,37732000
Jakarta,-6.175,106.8275,Indonesia,IDN,33756000
Delhi,28.61,77.23,India,IND,32226000
Guangzhou,23.13,113.26,China,CHN,26940000
Mumbai,19.0761,72.8775,India,IND,24973000
Manila,14.5958,120.9772,Philippines,PHL,24922000
Shanghai,31.1667,121.4667,China,CHN,24073000
Sao Paulo,-23.55,-46.6333,Brazil,BRA,23086000
Seoul,37.56,126.99,South Korea,KOR,23016000
```

```
19 import csv
20 with open('worldcitiesF23.csv', 'r') as f:
21     reader = csv.reader(f)
22     data = [row for row in reader]
23     print(f'Loaded data for {len(data)} cities.')
24     print(*data[:10],sep='\n')
```

that traverses the csv reader to fill our data list. Check out how clean the data is ...

```
['city', 'lat', 'lng', 'country', 'code', 'pop']
['Tokyo', '35.6897', '139.6922', 'Japan', 'JPN', '37732000']
['Jakarta', '-6.175', '106.8275', 'Indonesia', 'IDN', '33756000']
['Delhi', '28.61', '77.23', 'India', 'IND', '32226000']
['Guangzhou', '23.13', '113.26', 'China', 'CHN', '26940000']
['Mumbai', '19.0761', '72.8775', 'India', 'IND', '24973000']
['Manila', '14.5958', '120.9772', 'Philippines', 'PHL', '24922000']
['Shanghai', '31.1667', '121.4667', 'China', 'CHN', '24073000']
['Sao Paulo', '-23.55', '-46.6333', 'Brazil', 'BRA', '23086000']
['Seoul', '37.56', '126.99', 'South Korea', 'KOR', '23016000']
```

... but csv can do even more. The csv module's `DictReader()` can use the column headers to create a reader that returns dictionaries. This code produces a list of dictionaries ...

```
27 with open('worldcitiesF23.csv', 'r') as f:
28     reader = csv.DictReader(f)
29     data = [row for row in reader]
30     print(f'Loaded data for {len(data)} cities.')
31     print(*data[:10],sep='\n')
```

## Final Project: Acquiring and selecting data

... the output looks like this ...

```
Loaded data for 44691 cities.
{'city': 'Tokyo', 'lat': '35.6897', 'lng': '139.6922', 'country': 'Japan', 'code': 'JPN', 'pop': '37732000'}
{'city': 'Jakarta', 'lat': '-6.175', 'lng': '106.8275', 'country': 'Indonesia', 'code': 'IDN', 'pop': '33756000'}
{'city': 'Delhi', 'lat': '28.61', 'lng': '77.23', 'country': 'India', 'code': 'IND', 'pop': '32226000'}
{'city': 'Guangzhou', 'lat': '23.13', 'lng': '113.26', 'country': 'China', 'code': 'CHN', 'pop': '26940000'}
{'city': 'Mumbai', 'lat': '19.0761', 'lng': '72.8775', 'country': 'India', 'code': 'IND', 'pop': '24973000'}
{'city': 'Manila', 'lat': '14.5958', 'lng': '120.9772', 'country': 'Philippines', 'code': 'PHL', 'pop': '24922000'}
{'city': 'Shanghai', 'lat': '31.1667', 'lng': '121.4667', 'country': 'China', 'code': 'CHN', 'pop': '24073000'}
{'city': 'Sao Paulo', 'lat': '-23.55', 'lng': '-46.6333', 'country': 'Brazil', 'code': 'BRA', 'pop': '23086000'}
{'city': 'Seoul', 'lat': '37.56', 'lng': '126.99', 'country': 'South Korea', 'code': 'KOR', 'pop': '23016000'}
{'city': 'Mexico City', 'lat': '19.4333', 'lng': '-99.1333', 'country': 'Mexico', 'code': 'MEX', 'pop': '21804000'}
```

Cool! Now let's do some manipulation. Let's make a dictionary called "cityData" where the city is the primary key that refers to another dictionary that contains all the other related data.

Consider this code ...

```
34 print(f"{data[0]=}") # the (initial) record for Tokyo
35 cityData = {} # initialize the dictionary
36 # build the dictionary
37 for d in data: # traverse list of dictionaries
38     key = d.pop('city') # remove 'city' to use as key
39     cityData[key] = d # include in cityData
40 # now we can look up some values
41 print(f"{cityData['Tokyo']=}")
42 print(f"{cityData['Tokyo']['country']=}")
43 print(f"{data[0]=}\n") # BEWARE: data is changed
```

Hopefully, you get the gist of what's happening here. We initialize the new dictionary (line 35) and traverse the list of dictionaries (line 37). For each dictionary item in the list (d), we pop the city so we can use it as a key (line 38) and use the key to refer to what's left of the dictionary item (line 39). In the process, we print 4 lines. Let's take a look ...

```
data[0]={'city': 'Tokyo', 'lat': '35.6897', 'lng': '139.6922', 'country': 'Japan', 'code': 'JPN', 'pop': '37732000'}
cityData['Tokyo']={'lat': '35.6897', 'lng': '139.6922', 'country': 'Japan', 'code': 'JPN', 'pop': '37732000'}
cityData['Tokyo']['country']='Japan'
data[0]={'lat': '35.6897', 'lng': '139.6922', 'country': 'Japan', 'code': 'JPN', 'pop': '37732000'}
```

It looks like everything went like we wanted it to ... but it's important to notice that, in the process, we changed our original data (compare the first line with the last). Keep in mind that lists and dictionaries are both mutable. So, when we popped the city from d, we also popped it from the records in data. This means we can no longer find city names in our data list.

Way back on a previous page, I suggested that maybe latitude and longitude could merge into a single tuple called "location". How can we do that? Consider the code below ...

```
47 print(f"{cityData['Tokyo']['lat']=}")
48 print(f"{cityData['Tokyo']['lng']=}")
49 for k, v in cityData.items():
50     cityData[k]['location'] = (v.pop('lat'), v.pop('lng'))
51 print(f"{cityData['Tokyo']=}")
52 print(f"{cityData['Tokyo']['location']=}\n")
--
```

In line 49, we traverse the items in the list and establish the key (k) and value (v) for each item. In line 50 (left side) we establish the new key 'location' in the 2nd level dictionary and (on the right side) associate the tuple containing the latitude and longitude to the new key by popping them off the value. We print 4 lines to observe what we have done ...

## Final Project: Acquiring and selecting data

```
cityData['Tokyo']['lat']='35.6897'  
cityData['Tokyo']['lng']='139.6922'  
cityData['Tokyo']={'country': 'Japan', 'code': 'JPN', 'pop': '37732000', 'location': ('35.6897', '139.6922')}  
cityData['Tokyo']['location']=('35.6897', '139.6922')
```

The 'lat' and 'lng' keys are gone and the new key 'location' refers to the tuple. One thing I'd like to get across through this experiment is the versatility of mutable structures and how that can be helpful but also may have unintended consequences. Just try to be aware of what you're doing. And speaking of awareness, maybe you're wondering if the changes we made to cityData also changed our list of dictionaries (data) ... well, let's take a look ... here's the first 10 records ...

```
{'country': 'Japan', 'code': 'JPN', 'pop': '37732000', 'location': ('35.6897', '139.6922')}  
{'country': 'Indonesia', 'code': 'IDN', 'pop': '33756000', 'location': ('-6.175', '106.8275')}  
{ 'lat': '28.61', 'lng': '77.23', 'country': 'India', 'code': 'IND', 'pop': '32226000'}  
{ 'country': 'China', 'code': 'CHN', 'pop': '26940000', 'location': ('23.13', '113.26')}  
{ 'country': 'India', 'code': 'IND', 'pop': '24973000', 'location': ('19.0761', '72.8775')}  
{ 'country': 'Philippines', 'code': 'PHL', 'pop': '24922000', 'location': ('14.5958', '120.9772')}  
{ 'country': 'China', 'code': 'CHN', 'pop': '24073000', 'location': ('31.1667', '121.4667')}  
{ 'country': 'Brazil', 'code': 'BRA', 'pop': '23086000', 'location': ('-23.55', '-46.6333')}  
{ 'country': 'South Korea', 'code': 'KOR', 'pop': '23016000', 'location': ('37.56', '126.99')}  
{ 'country': 'Mexico', 'code': 'MEX', 'pop': '21804000', 'location': ('19.4333', '-99.1333')}
```

Hmm. One of these is not like the other. If you go back a couple of pages and look, you'll see that the third line is for "Delhi". Let's look for cityData["Delhi"] ...

```
cityData['Delhi']={'country': 'United States', 'code': 'USA', 'pop': '10921', 'location': ('37.4306', '-120.7759')}
```

Ooooh. That's not right ... you thought this was going to be easy, right? Nope. What went wrong? Is it possible there are two cities with the same name? Yeah. Probably.

PART 3: AMBIGUOUS DATA. If we go back to the original list of dictionaries, we can use filter to see how many cities are named "Delhi" ... I'm guessing there's more than one and one of them is in the US.

So, we execute this: `delhi = list(filter(lambda x:x['city']=='Delhi',data))` and see this ...

```
{'city': 'Delhi', 'lat': '28.61', 'lng': '77.23', 'country': 'India', 'code': 'IND', 'pop': '32226000'}  
{'city': 'Delhi', 'lat': '37.4306', 'lng': '-120.7759', 'country': 'United States', 'code': 'USA', 'pop': '10921'}
```

So, now, we've got to figure out how to deal with this (and the possibility that there are others). If we want to use city names as keys, we have to find a way to disambiguate them. We could make the keys a combination of city and country code: "Delhi, IND" and "Delhi, USA". But, this is a really big list and it's likely that there may be other cases where cities with the same name are in the same country. Let's check that out. I think there's probably many US cities with the name "Springfield" ...

```
springfield = list(filter(lambda x:x['city']=='Springfield',data))
```



## Final Project: Acquiring and selecting data

```
{'city': 'Delhi', 'lat': '28.61', 'lng': '77.23', 'country': 'India', 'code': 'IND', 'pop': '32226000'}
{'city': 'Delhi', 'lat': '37.4306', 'lng': '-120.7759', 'country': 'United States', 'code': 'USA', 'pop': '10921'}
{'city': 'Springfield', 'lat': '42.1155', 'lng': '-72.5395', 'country': 'United States', 'code': 'USA', 'pop': '620494'}
{'city': 'Springfield', 'lat': '37.1943', 'lng': '-93.2916', 'country': 'United States', 'code': 'USA', 'pop': '298722'}
{'city': 'Springfield', 'lat': '39.7709', 'lng': '-89.654', 'country': 'United States', 'code': 'USA', 'pop': '162588'}
{'city': 'Springfield', 'lat': '39.93', 'lng': '-83.7959', 'country': 'United States', 'code': 'USA', 'pop': '85121'}
{'city': 'Springfield', 'lat': '44.0538', 'lng': '-122.9811', 'country': 'United States', 'code': 'USA', 'pop': '62138'}
{'city': 'Springfield', 'lat': '38.781', 'lng': '-77.1839', 'country': 'United States', 'code': 'USA', 'pop': '31769'}
{'city': 'Springfield', 'lat': '39.9281', 'lng': '-75.3362', 'country': 'United States', 'code': 'USA', 'pop': '24851'}
{'city': 'Springfield', 'lat': '40.0986', 'lng': '-75.2016', 'country': 'United States', 'code': 'USA', 'pop': '20590'}
{'city': 'Springfield', 'lat': '36.4949', 'lng': '-86.8711', 'country': 'United States', 'code': 'USA', 'pop': '18561'}
{'city': 'Springfield', 'lat': '40.6994', 'lng': '-74.3254', 'country': 'United States', 'code': 'USA', 'pop': '17004'}
{'city': 'Springfield', 'lat': '49.9292', 'lng': '-96.6939', 'country': 'Canada', 'code': 'CAN', 'pop': '15342'}
{'city': 'Springfield', 'lat': '43.2907', 'lng': '-72.4809', 'country': 'United States', 'code': 'USA', 'pop': '9089'}
```

... we have a problem! Clearly, we can't disambiguate city names by country. The only reliable way to disambiguate is by location ... two cities can't be in the same place ... but we would have to add the location to the city key and then you couldn't look up a city without knowing the latitude and longitude.

This is a real-life problem. Just like there are cities with the same name, there are people with the same name. There are people with the same birthday. We can't exclude any of them. Thankfully, this is an academic exercise, we can do whatever we want. We know that there may be multiple cities in different locations with the same name. So, one possible thing to try is to have each key (city name) associated with a list of cities with that name. Back to square 1 ... but this time we'll go faster.

Here's how to create a dictionary where each city name is a key that associates with a list of dictionaries ... one for each city with that name.

```
9 cityData = {} # initialize the dictionary
10 # build the dictionary
11 for d in data: # traverse list of dictionaries
12     key = d.pop('city') # remove 'city' to use as key
13     if key in cityData.keys():
14         cityData[key].append(d) # append to the list
15     else:
16         cityData[key] = [d] # new key, make the list
17 # now we can look up some values
18 print(f"{cityData['Tokyo']=}")
19 print(f"{cityData['Tokyo'][0]['country']=}")
20 print('Springfield list ...', *cityData['Springfield'], '\n', sep='\n')
```

Here's the result ...

```
cityData['Tokyo']=[{'lat': '35.6897', 'lng': '139.6922', 'country': 'Japan', 'code': 'JPN', 'pop': '37732000'}]
cityData['Tokyo'][0]['country']='Japan'
Springfield list ...
{'lat': '42.1155', 'lng': '-72.5395', 'country': 'United States', 'code': 'USA', 'pop': '620494'}
{'lat': '37.1943', 'lng': '-93.2916', 'country': 'United States', 'code': 'USA', 'pop': '298722'}
{'lat': '39.7709', 'lng': '-89.654', 'country': 'United States', 'code': 'USA', 'pop': '162588'}
{'lat': '39.93', 'lng': '-83.7959', 'country': 'United States', 'code': 'USA', 'pop': '85121'}
{'lat': '44.0538', 'lng': '-122.9811', 'country': 'United States', 'code': 'USA', 'pop': '62138'}
{'lat': '38.781', 'lng': '-77.1839', 'country': 'United States', 'code': 'USA', 'pop': '31769'}
{'lat': '39.9281', 'lng': '-75.3362', 'country': 'United States', 'code': 'USA', 'pop': '24851'}
{'lat': '40.0986', 'lng': '-75.2016', 'country': 'United States', 'code': 'USA', 'pop': '20590'}
{'lat': '36.4949', 'lng': '-86.8711', 'country': 'United States', 'code': 'USA', 'pop': '18561'}
{'lat': '40.6994', 'lng': '-74.3254', 'country': 'United States', 'code': 'USA', 'pop': '17004'}
{'lat': '49.9292', 'lng': '-96.6939', 'country': 'Canada', 'code': 'CAN', 'pop': '15342'}
{'lat': '43.2907', 'lng': '-72.4809', 'country': 'United States', 'code': 'USA', 'pop': '9089'}
```

## Final Project: Acquiring and selecting data

PART 5: STRINGS ARE NOT NUMBERS / MISSING DATA. Notice a couple of things here. The population data for Springfield is in descending order. But the "numbers" are text. It would be better if they were integers for comparison purposes. Also, latitude and longitude ought to be floats. Let's do those conversions. If we do them now, we can avoid surprises later. While we're at it, I'll make tuples of the latitude and longitude to use as the 'location'.

```
22 # create location tuples
23 for k, v in cityData.items(): # traverse the city names
24     for item in v: # traverse the list of cities
25         lat = float(item.pop('lat'))
26         lng = float(item.pop('lng'))
27         item['location'] = (lat,lng) # establish location
28         item['pop'] = int(item['pop'])
29 print(f"{cityData['Tokyo']=}")
30 print(f"{cityData['Tokyo'][0]['country']=}")
31 print('Springfield list ...',*cityData['Springfield'],'\n',sep='\n')
```

Unfortunately (or fortunately, depending on how you look at it) line 28 crashes. It looks like one (or more) population field is empty ... good thing

```
Traceback (most recent call last):
  File "/home/runner/cityDemo2/main.py", line 28, in <module>
    item['pop'] = int(item['pop'])
ValueError: invalid literal for int() with base 10: ''
```

we found it now. Let's use try / except to identify the record(s) and handle them ...

```
21 badRecords = []
22 # create location tuples
23 for k, v in cityData.items(): # traverse the city names
24     for item in v: # traverse the list of cities
25         lat = float(item.pop('lat'))
26         lng = float(item.pop('lng'))
27         item['location'] = (lat,lng) # establish location
28     try:
29         item['pop'] = int(item['pop'])
30     except ValueError:
31         badRecords.append((k,item))
32         item['pop'] = 0
33 print(f"{len(badRecords)} missing population data set to 0")
34 print(f"{cityData['Tokyo']=}")
35 print(f"{cityData['Tokyo'][0]['country']=}")
36 print('Springfield list ...',*cityData['Springfield'],'\n',sep='\n')
```

I set the 307 records missing population data to zero. My decision is based on wanting to preserve the location data. If you were studying populations, you would probably want to exclude the records entirely because zeroes would affect your results.

## Final Project: Acquiring and selecting data

```
307 missing population data set to 0
cityData['Tokyo']=[{'country': 'Japan', 'code': 'JPN', 'pop': 37732000, 'location': (35.6897, 139.6922)}]
cityData['Tokyo'][0]['country']='Japan'
Springfield list ...
{'country': 'United States', 'code': 'USA', 'pop': 620494, 'location': (42.1155, -72.5395)}
{'country': 'United States', 'code': 'USA', 'pop': 298722, 'location': (37.1943, -93.2916)}
{'country': 'United States', 'code': 'USA', 'pop': 162588, 'location': (39.7709, -89.654)}
{'country': 'United States', 'code': 'USA', 'pop': 85121, 'location': (39.93, -83.7959)}
{'country': 'United States', 'code': 'USA', 'pop': 62138, 'location': (44.0538, -122.9811)}
{'country': 'United States', 'code': 'USA', 'pop': 31769, 'location': (38.781, -77.1839)}
{'country': 'United States', 'code': 'USA', 'pop': 24851, 'location': (39.9281, -75.3362)}
{'country': 'United States', 'code': 'USA', 'pop': 20590, 'location': (40.0986, -75.2016)}
{'country': 'United States', 'code': 'USA', 'pop': 18561, 'location': (36.4949, -86.8711)}
{'country': 'United States', 'code': 'USA', 'pop': 17004, 'location': (40.6994, -74.3254)}
{'country': 'Canada', 'code': 'CAN', 'pop': 15342, 'location': (49.9292, -96.6939)}
{'country': 'United States', 'code': 'USA', 'pop': 9089, 'location': (43.2907, -72.4809)}
```

Now that all the numerical data actually consists of numbers, let's sort each city list by population so the first location in each list represents the city with the highest population. It turns out that the city with the largest number of entries is "Santa Cruz" so I print out all of those instances.

```
38 v for k,v in cityData.items():
39     # sort each city list by population
40     v.sort(key=lambda x:x['pop'], reverse=True)
41     print('\nSanta Cruz',*cityData['Santa Cruz'], '\n', sep='\n')
```

```
Santa Cruz
{'country': 'Bolivia', 'code': 'BOL', 'pop': 3151676, 'location': (-17.7892, -63.1975)}
{'country': 'Spain', 'code': 'ESP', 'pop': 208688, 'location': (28.4667, -16.25)}
{'country': 'United States', 'code': 'USA', 'pop': 165933, 'location': (36.9789, -122.0346)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 126735, 'location': (14.5998, 120.9802)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 123574, 'location': (14.2833, 121.4167)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 101125, 'location': (6.8333, 125.4167)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 63839, 'location': (15.7667, 119.9167)}
{'country': 'Costa Rica', 'code': 'CRI', 'pop': 55104, 'location': (10.2358, -85.6408)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 54692, 'location': (13.4833, 122.0333)}
{'country': 'Portugal', 'code': 'PRT', 'pop': 43005, 'location': (32.6833, -16.8)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 42417, 'location': (13.0831, 120.7192)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 41366, 'location': (17.0853, 120.4553)}
{'country': 'Brazil', 'code': 'BRA', 'pop': 35797, 'location': (-6.2289, -36.0228)}
{'country': 'Chile', 'code': 'CHL', 'pop': 34914, 'location': (-34.6372, -71.3631)}
{'country': 'Venezuela', 'code': 'VEN', 'pop': 29773, 'location': (10.1819, -67.5025)}
{'country': 'Ecuador', 'code': 'ECU', 'pop': 11262, 'location': (-0.5333, -90.35)}
{'country': 'Philippines', 'code': 'PHL', 'pop': 0, 'location': (14.1167, 121.2833)}
```

There are a lot of cities named Santa Cruz in the Philippines.

Let's do a *sanity check* of this data. We could, for example, add up all the populations and see what number we get. The world population is estimated to be around 8 billion souls. Let's write the code to see how many people live in these cities. The total is: 5,076,372,338 ... not bad considering I have no idea when this data was compiled. I encourage you to write the code and see if you get the same result.



## Final Project: Acquiring and selecting data

TASK #1. Write a function called `getCityData()` that reads the "world cities" data file (`worldcitiesF23.csv`) and returns a dictionary containing the data. The dictionary should have two levels. The highest level uses the tuple representing latitude and longitude as a key that refers to a dictionary containing the rest of the data. Doing this avoids the problem with cities having the same name.

These Python statements ...

```
# read the data file and return a dictionary
cityDict = getCityData()
# check by printing the first five items
for k, v in list(cityDict.items())[:5]:
    print(f"location: {k}")
    print(f"    {v}")
```

... should print this ...

```
location: (35.6897, 139.6922)
{'city': 'Tokyo', 'country': 'Japan', 'iso3': 'JPN', 'pop': 37732000}
location: (-6.175, 106.8275)
{'city': 'Jakarta', 'country': 'Indonesia', 'iso3': 'IDN', 'pop': 33756000}
location: (28.61, 77.23)
{'city': 'Delhi', 'country': 'India', 'iso3': 'IND', 'pop': 32226000}
location: (23.13, 113.26)
{'city': 'Guangzhou', 'country': 'China', 'iso3': 'CHN', 'pop': 26940000}
location: (19.0761, 72.8775)
{'city': 'Mumbai', 'country': 'India', 'iso3': 'IND', 'pop': 24973000}
```

Note that the numbers are not strings. The numbers in the tuple are float and the population is int .

## PART 6: TURN THE TABLES

Now that we've got all that sorted out. Let's think about a different problem. Suppose you want to know the nearest city to where you are. The issue here is you need to calculate how far away you are from each city in the list. You might think that you could just use the difference between latitude and longitude. That will give you angular distance but angular distance based on latitude and longitude is only part of the story. At the equator, a difference of 5 degrees in latitude (~550 km) is longer than the same angular distance anywhere else on the globe (~400 km at 45 degrees latitude). To know how far one set of coordinates is from another, one needs to use something called the Haversine formula. This is not a class in analytic geometry so we will just use a function that uses the Haversine Formula to calculate the distance between two locations given in geodetic coordinates. To get the distance in km just use the [code in the appendix](#) . Call `havDis()` and pass in the two tuples that represent geodetic coordinates.

Let's try to find the cities in the list that are within 50 km of Worcester. We can use the list of dictionaries (data) or the dictionary of lists (cityData). Let's use the list of dictionaries ... but we'll need to rework it in light of what we learned. Basically, we need to (1) convert numerical strings

## Final Project: Acquiring and selecting data

to numbers, (2) identify missing population data and set it to zero, (3) combine latitude and longitude into a tuple that is referred to with the key 'location'. Let's go.

Read the data and convert ...

```
3 ✓ with open('worldcitiesF23.csv', 'r') as f:
4     reader = csv.DictReader(f)
5     data = [row for row in reader]
6     # print(f'Loaded data for {len(data)} cities.')
7     # print(*data[:10],sep='\n')
8     # print()
9 ✓ for d in data:
10     # handle missing population data
11     d['pop'] = 0 if d['pop'] == '' else int(d['pop'])
12     # convert latitude and longitude to coordinate tuple
13     d['location'] = ( float(d.pop('lat')), float(d.pop('lng')) )
```

Now compile a list of close cities ...

```
16 worcester = (42.2626, -71.798889) # target coordinates
17 closeCities = []
18 ✓ for item in data: # traverse the cities
19     # compute the distance
20     distance = havDist(worcester, item['location'])
21 ✓     if distance < 50.0: # compile the list
22         closeCities.append((item["city"], round(distance,1), item["pop"]))
23 closeCities.sort(key=lambda x:x[1]) # sort by distance
```

Now, we can calculate the total population in a 50 km radius ...

```
26 print(f'total population = {sum([x[2] for x in closeCities])}') 
```

Go through the exercise yourself to find out how many cities and the total population. When I downloaded this dataset, I actually had no idea that all of these towns were there. Your hometown is probably in there too.

**TASK #2.** Write a function called `findCities()` that accepts a target location, your dictionary of cities, and a radius value as input and returns a list of cities (and associated data as a dictionary) that surround the target location up to the prescribed distance away. Each item in the list is a dictionary that contains this information: (1) the city name (key='city'), (2) the country name (key='country'), (3) the population (key='pop'), and (4) the distance from the target (key='distance'). Use the `havDist()` function provided at the [end of this document](#) to compute the distance.

## Final Project: Acquiring and selecting data

Executing these Python statements ...

```
worcester = (42.2626, -71.798889)
# finding cities within 20 km Worcester
closeCities = findCities(worcester, cityDict, 20)
# compute the total population
total = sum([x['pop'] for x in closeCities])
# print the results
print('\nClosest cities to Worcester')
print('\t', *closeCities, f'{total=}', sep='\n\t')
```

Should print this ...

Closest cities to Worcester

```
{'city': 'Worcester', 'country': 'United States', 'pop': 573573}
{'city': 'Shrewsbury', 'country': 'United States', 'pop': 38291}
{'city': 'Westborough', 'country': 'United States', 'pop': 21213}
{'city': 'Holden', 'country': 'United States', 'pop': 19659}
{'city': 'Grafton', 'country': 'United States', 'pop': 19540}
{'city': 'Auburn', 'country': 'United States', 'pop': 16826}
{'city': 'Northbridge', 'country': 'United States', 'pop': 16291}
{'city': 'Northborough', 'country': 'United States', 'pop': 15605}
{'city': 'Clinton', 'country': 'United States', 'pop': 15221}
{'city': 'Millbury', 'country': 'United States', 'pop': 13827}
{'city': 'Oxford', 'country': 'United States', 'pop': 13399}
{'city': 'Charlton', 'country': 'United States', 'pop': 13312}
{'city': 'Spencer', 'country': 'United States', 'pop': 11963}
{'city': 'Leicester', 'country': 'United States', 'pop': 11077}
{'city': 'Sutton', 'country': 'United States', 'pop': 9334}
{'city': 'Rutland', 'country': 'United States', 'pop': 8988}
total=818119
```

PART 7: WORKING WITH TIME. Working with locations on the surface of the earth is sort of a two-dimensional problem ... I'm not advocating for "flat-earth" here ... I'm just saying we are using two numbers to identify the location and we give no thought of time. But time is important to consider. Populations, for example, change over time. Keeping track of time in a computer program is not exactly straightforward. Maybe you heard of the "Y2K" panic? Some older systems were not equipped to deal with the year changing from 1999 to 2000. Look it up. Let's talk about time.

We often use strings to express time. Time consists of a time-of-day (ToD) and also a date. ToD can be expressed as a 12 hour day (with AM or PM) or a 24 hour day. Dates can be expressed with the year first or the month first. Day, month, year can be separated by hyphens or slashes. We can use one digit, two digits, or four digits, or words. People adapt, but machines need some standard. In Python, we use a module called `datetime` to keep track of time using standard datetime objects. In Python, date and time are merged into one single datetime object. Every date has a time and every time has a date ... they are inseparable. Let's write some code.

Here, line 2 creates a datetime object, `dt` which is a representation of the time at the moment the

```
1 import datetime
2 dt = datetime.datetime.now()
3 print(dt)
```

statement was executed created by the `datetime.now()` method . Printing it out reveals a date and time. For me it's ...

```
2023-11-14 05:07:25.933823
```

This is the standard format. "yyyy-mm-dd hh:mm:ss.ssssss" which makes sense because it goes from slowest-changing (years) to fastest-changing (seconds). That's the time now. But what about sometime in the past ... or a date or time given in a datafile ... how do I represent those? My dog's birthday is November 6, 2009. How can I get a datetime object for that? We use the "strptime" method ... "strp" stands for "string parse". We give a string representing the date and then provide a format descriptor. Like this ...

```
4 bDay = datetime.datetime.strptime('2009-11-06', '%Y-%m-%d')
5 print(bDay)
```

... and the result is ...

```
2009-11-06 00:00:00
```

Note, the time. I didn't give a time ... but like I said ... date and time are connected.

There are many various format codes. You can find them [here](#) .

Here, I give the time ...

```
bDay = datetime.datetime.strptime('2009-11-06 10:27:00',
                                   '%Y-%m-%d %H:%M:%S')
```

... which results in ...

```
2009-11-06 10:27:00
```

So, how old is my dog? Well, that's the beauty of datetime objects ... I can do some arithmetic with them.

```
2 dtm = datetime.datetime.now()
3 bDay = datetime.datetime.strptime('2009-11-06 10:27:00',
4                                   '%Y-%m-%d %H:%M:%S')
5 age = dtm - bDay
6 print(age)
7 print(type(dtm))
8 print(type(bDay))
9 print(type(age))
```

My dog's age is ...

```
5120 days, 19:37:56.971051
<class 'datetime.datetime'>
<class 'datetime.datetime'>
<class 'datetime.timedelta'>
```

But wait ... that's not what I want. Notice that the dates are datetime objects but the age is a timedelta object. So far, we've been observing the default print behavior. But there's a lot more we can do. A timedelta object has an attribute called `days`. We can use that to calculate years.

## Final Project: Acquiring and selecting data

```
2  dtm = datetime.datetime.now()
3  bDay = datetime.datetime.strptime('2009-11-06 10:27:00',
4                                     '%Y-%m-%d %H:%M:%S')
5  age = dtm - bDay
6  print(age) # default print behavior
7  print(type(dtm))
8  print(type(bDay))
9  print(type(age))
10 print(age.days) # get the days
11 print(age.days//365, 'years') # calculate years
```

```
5120 days, 19:46:20.512054
<class 'datetime.datetime'>
<class 'datetime.datetime'>
<class 'datetime.timedelta'>
5120
14 years
```

All datetime objects have a method named "strftime()" that provides a way to generate a flexible date/time string using various format codes. Here's an example ...

```
print(dtm.strftime("%A, %B %d, %Y"))
```

Tuesday, November 14, 2023

See the [documentation](#) for information on format specifiers.

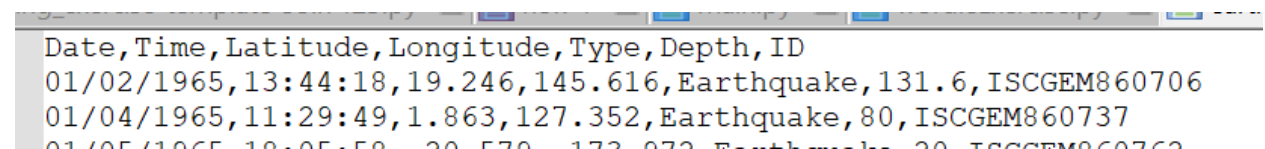
When we encounter a date and/or time in our data, we must convert that to a datetime object. The text (i.e. string) that represents date/time is inconsistent across various application domains. Text-oriented dates/times cannot be directly compared, sorted, or searched efficiently. Python datetime objects give us the ability to work with time in our applications. To create a datetime object from text, we use the `strptime()` method with an appropriate format specification. To create text from a datetime object we use the `strftime()` method with an appropriate format specification. For more on datetime objects, check out the code that demonstrates the essentials of working with datetime objects [HERE](#).

Let's take a look at a real-world application.

### PART 8: QUAKES

We will be working with a data file that provides information on earthquakes. I trust that at this point, you are able to read such a file and create a dictionary with location as the primary key. The data file contains datetime information as two separate fields. Here's a portion of a data file

...



```
Date,Time,Latitude,Longitude,Type,Depth,ID
01/02/1965,13:44:18,19.246,145.616,Earthquake,131.6,ISCGEM860706
01/04/1965,11:29:49,1.863,127.352,Earthquake,80,ISCGEM860737
01/05/1965,10:05:50,20.570,173.070,Earthquake,20,ISCGEM860760
```



## Final Project: Acquiring and selecting data

From this, you can see that the date format is: %m/%d/%Y and the time format is %H:%M:%S. If we do a dictRead() of the file, we will have Date and Time in separate fields "Date": "01/02/1965" and "Time": "13:44:18". To create a datetime object you must concatenate these strings (I suggest using f-strings for this purpose) and use strptime(). There is no guarantee that every date / time in the file is formatted this way. But, just as we did with the city data, you can respond intelligently to the exception and determine how to handle it. Aside from the date, you will need to convert some strings to numbers ... 'Latitude' and 'Longitude' (of course) and also 'Magnitude'.

**TASK #3.** Just as you did for the "world city" data, write a function called `getQuakeData()` that reads the "earthquake" data file (earthquakesF23.csv) and returns a dictionary containing the data. The dictionary should have two levels. The highest level uses the tuple representing latitude and longitude as a key that refers to a dictionary containing the rest of the data. In creating the dictionary, strings must be converted to numbers (floats, ints) where appropriate and dates and times must be converted to datetime objects. Be aware that some datafile rows may have dates and times in different formats.

These Python statements ...

```
qDict = getQuakeData()
# check by printing the first five items
print('\n*** Earthquake Data ***')
for n, (k, v) in enumerate(list(qDict.items())[:5]):
    if n == 0:
        print(f"keys: {list(v.keys())}") # print keys once
        print(f"location: {k}")
        print(f" {list(v.values())}")
```

Produce these results ...

```
*** Earthquake Data ***
keys: ['Type', 'Depth', 'Magnitude', 'Magnitude Type', 'datetime']
location: (19.246, 145.616)
['Earthquake', '131.6', 6.0, 'MW', datetime.datetime(1965, 1, 2, 13, 44, 18)]
location: (1.863, 127.352)
['Earthquake', '80', 5.8, 'MW', datetime.datetime(1965, 1, 4, 11, 29, 49)]
location: (-20.579, -173.972)
['Earthquake', '20', 6.2, 'MW', datetime.datetime(1965, 1, 5, 18, 5, 58)]
location: (-59.076, -23.557)
['Earthquake', '15', 5.8, 'MW', datetime.datetime(1965, 1, 8, 18, 49, 43)]
location: (11.938, 126.427)
['Earthquake', '15', 5.8, 'MW', datetime.datetime(1965, 1, 9, 13, 32, 50)]
```

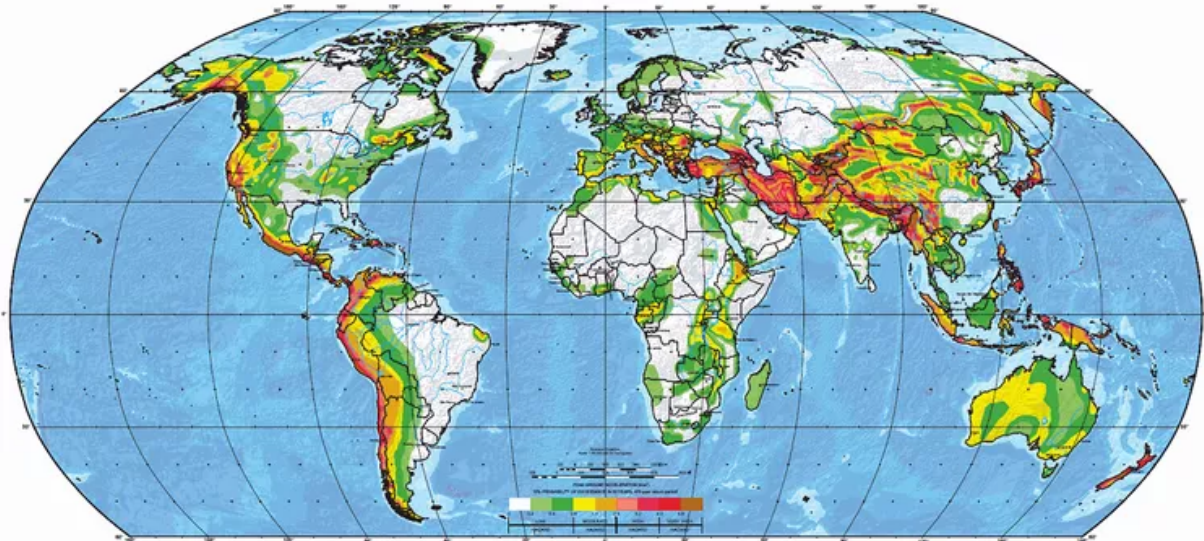
## PART 9: SELECTING DATA

Hopefully it's pretty clear that there are many criteria that could be used to select data records for analysis. You ought to be prepared to select data based on values associated with any of the keys in the earthquake data: Location, datetime, Type, Depth, Magnitude, and Magnitude Type. Then, for every earthquake selected, the affected cities could be located using the "world cities" dictionary and the `findCities()` function you developed in completing task 2.

## Final Project: Acquiring and selecting data

You have seen several ways to create lists of selected data using for loops, list comprehensions, filter(), and map(). The filter() and map() functions work well when the data is relatively simple. But lambda functions are not well-suited to such complications. I prefer to use a list comprehension if I'm selecting by a single factor. For multiple factors, a for loop makes for code that's more readable and maintainable.

Selecting by location. The earth can be divided into zones based on latitude and longitude. Latitude-based regions may be as large as north/south hemispheres or zones based on climate: north tropical (0 to 23.5), south tropical (0 to -23.5), north temperate (23.5 to 66.5), south temperate (-23.5 to -66.5), arctic (66.5 to 90), antarctic (-66.5 to -99). It can also be divided by longitude (or meridians) in various ways: time zones (15 degree increments), hemispheres, semi-hemispheres, etc. Nothing prevents us from simply creating some arbitrary system of our own. In the seismic zone map below, the earth appears to be divided into 12 zones based on latitude ... each is two time-zones wide..



Regardless of how we divide the globe, filtering location requires us to use two separate criteria. In code, we can try to apply latitude and longitude criteria together or separately. Applying them all together could result in some code that is difficult to read (use a for loop). Let's use list comprehensions to apply latitude and longitude criteria separately.

In the code below, I select earthquake data records that have latitudes in the northern temperate region.

```
qDict = getQuakeData()  
# Select northern temperate zone from 23.5 to 66.5 latitude  
lat1, lat2 = 23.5, 66.5  
# select based on latitude // list of tuples: (<location>,<data>)  
selected = [(lat,lng),v) for (lat,lng), v in qDict.items() if lat1 < lat < lat2 ]
```

## Final Project: Acquiring and selecting data

The data records are selected from the original earthquake dictionary and appended to a list called `selected`. There are over 6000 data records that meet the latitude criteria. Here's how I report the selected records ...

```
127 selected = [(lat,lng),v) for (lat,lng), v in qDict.items() if lat1 < lat < lat2 ]
128 print(f"\n\nAfter filtering for latitudes {lat1} to {lat2}")
129 print(f"Total number of earthquakes found was {len(selected)}.")
130 print(f"\nFirst five earthquakes found are ...")
131 for n, (k,v) in enumerate(selected[:5]): # print just the first 5
132     if n == 0: # print the keys only once
133         print(f"keys: {list(v.keys())}") # print keys once
134         print(f"location: {k}") # print the coordinates
135         print(f" {list(v.values())}") # followed by the data values
```

... and here's the result ...

After filtering for latitudes 23.5 to 66.5  
Total number of earthquakes found was 6115.

First five earthquakes found are ...

```
keys: ['Type', 'Depth', 'Magnitude', 'Magnitude Type', 'datetime']
location: (27.357, 87.867)
['Earthquake', '20', 5.9, 'MW', datetime.datetime(1965, 1, 12, 13, 32, 25)]
location: (54.636, 161.703)
['Earthquake', '55', 5.5, 'MW', datetime.datetime(1965, 1, 29, 9, 35, 30)]
location: (37.523, 73.251)
['Earthquake', '15', 6.0, 'MW', datetime.datetime(1965, 2, 2, 15, 56, 51)]
location: (51.251, 178.715)
['Earthquake', '30.3', 8.7, 'MW', datetime.datetime(1965, 2, 4, 5, 1, 22)]
location: (51.639, 175.055)
['Earthquake', '30', 6.0, 'MW', datetime.datetime(1965, 2, 4, 6, 4, 59)]
```

Further filtering will work to reduce the size of the `selected` list. Let's filter based on longitude now and restrict the `selected` results to continental US time zones ...

```
137 # Select continental US time zones from -60 to -120 longitude
138 lng1, lng2 = -120, -60
139 # select based on longitude // list of tuples: (<location>,<data>)
140 selected = [(lat,lng),v) for (lat,lng), v in selected if lng1 < lng < lng2 ]
141 print(f"\n\nAfter additional filtering for longitudes {lng1} to {lng2}")
142 print(f"Total number of earthquakes found was {len(selected)}.")
143 print(f"\nFirst five earthquakes found are ...")
144 for n, (k,v) in enumerate(selected[:5]):
145     if n == 0:
146         print(f"keys: {list(v.keys())}") # print keys once
147         print(f"location: {k}")
148         print(f" {list(v.values())}")
```

This reduces the number of data records to 196...

## Final Project: Acquiring and selecting data

After additional filtering for longitudes -120 to -60  
Total number of earthquakes found was 196.

First five earthquakes found are ...

```
keys: ['Type', 'Depth', 'Magnitude', 'Magnitude Type', 'datetime']
location: (28.133, -112.208)
['Earthquake', '10', 6.0, 'MW', datetime.datetime(1965, 2, 27, 7, 46, 25)]
location: (24.97, -109.085)
['Earthquake', '20', 5.9, 'MW', datetime.datetime(1966, 5, 18, 7, 32, 6)]
location: (31.0395, -113.7773333)
['Earthquake', '6', 6.35, 'ML', datetime.datetime(1966, 8, 7, 17, 36, 12)]
location: (37.38, -114.157)
['Earthquake', '10', 5.7, 'MW', datetime.datetime(1966, 8, 16, 18, 2, 36)]
location: (37.3021667, -116.4083333)
['Nuclear Explosion', '1.2', 5.62, 'ML', datetime.datetime(1966, 12, 20, 15, 30, 1)]
```

Selecting by year. Finally, let's choose the year 1995 ...

```
150 year = 1995
151 # Select based on year // list of tuples: (<location>,<data>)
152 selected = [(lat,lng),v for (lat,lng), v in selected if v['datetime'].year == year ]
153 print("\n\nAfter additional filtering for year ...")
154 print(f"Total number of earthquakes in {year} was {len(selected)}.")
155 print(f"\nAll selected earthquakes found in {year}")
156 for n, (k,v) in enumerate(selected):
157     if n == 0:
158         print(f"keys: {list(v.keys())}") # print keys once
159         print(f"location: {k}")
160         print(f" {list(v.values())}")
```

Now, we're left with 6 records ...

After additional filtering for year ...  
Total number of earthquakes in 1995 was 6.

All selected earthquakes found in 1995

```
keys: ['Type', 'Depth', 'Magnitude', 'Magnitude Type', 'datetime']
location: (30.285, -103.347)
['Earthquake', '17.8', 5.7, 'MW', datetime.datetime(1995, 4, 14, 0, 32, 56)]
location: (24.688, -110.228)
['Earthquake', '10', 6.2, 'MW', datetime.datetime(1995, 6, 30, 11, 58, 57)]
location: (24.386, -110.265)
['Earthquake', '10', 5.7, 'MW', datetime.datetime(1995, 6, 30, 13, 41, 43)]
location: (26.092, -110.284)
['Earthquake', '12.2', 6.6, 'MWB', datetime.datetime(1995, 8, 28, 10, 46, 12)]
location: (35.76, -117.64)
['Earthquake', '5', 5.5, 'MWC', datetime.datetime(1995, 9, 20, 23, 27, 36)]
location: (35.761, -117.638)
['Earthquake', '4.688', 5.75, 'ML', datetime.datetime(1995, 9, 20, 23, 27, 36)]
```

At any time, you ought to be able to determine the data record(s) that identify the strongest earthquake (based on the magnitude value). You ought to be able to select earthquakes based on "type". Every record in this data file is associated with one of four "types". What are they?

If an earthquake can be localized, one can estimate the communities that were affected ... just use the findCities() function that you have already written ... but for that, you will need to (1)

## Final Project: Acquiring and selecting data

compute a radius. The effective radius depends on magnitude. One way to compute effective radius (n kilometers) is to use this formula:  $10^{(0.5 * \text{magnitude} - 2)}$ <sup>1</sup>.

The strongest earthquake in our list above has no cities within the effective radius. If an earthquake occurs away from a populated area it can still be detected and its magnitude can be calculated. In this case, we can find the city that's closest by creating a list of cities within a 5000 km radius and then calling `min()` to determine the city closest to the target.

Here are my results ...

```
Largest quake is at (26.092, -110.284)
{'Type': 'Earthquake', 'Depth': '12.2', 'Magnitude': 6.6, 'Magnitude Type': 'MWB', 'datetime': datet
ime.datetime(1995, 8, 28, 10, 46, 12)}
0 affected cities within 19.952623149688787 km ...
Closest city is ...
{'city': 'Higuera de Zaragoza', 'country': 'Mexico', 'pop': 8976, 'distance': 101.23210316970648}
```

## PART 10: THE FINAL PROJECT

In the document above, I've demonstrated the fundamental machinery associated with the final project. Hopefully, by now, you have an appreciation for the depth of data acquisition and selection process that is essential to getting started. Your submission must have a sound user interface that provides a way for a user to enter selection parameters ...

- location: a range of latitude and longitude coordinates to include
  - minimum and maximum latitude
  - minimum and maximum longitude
  - user may enter nothing to indicate "all" locations
- time: a range of dates to include
  - earliest date
  - latest date
  - user may enter nothing to indicate "all" dates
- severity: a range of magnitudes to include
  - smallest magnitude
  - largest magnitude
  - user may enter nothing to indicate "all" magnitudes

---

<sup>1</sup> This expression is just an estimate for us to use in this project. There are many such formulations. None of them are very accurate.



## Final Project: Acquiring and selecting data

First steps. A good user interface is essential to the project. Here's a demonstration of my data selection interface ...

Notice a few things ...

The range of acceptable values is provided for each parameter.

Values outside the range are rejected.

The number of selected records is given before moving to the next step. The selections are cumulative.

At each step, the user is asked to verify before moving forward. A response is necessary. An affirmative response moves forward, a negative response repeats the step.

If the user enters the larger number first, the numbers are reversed.

If the user enters nothing, the full range is selected.

```
*** Earthquake Data ***
SELECT latitude : enter two values separated by comma
range is -77.08 through 86.005
Enter minimum/maximum latitude values: -20,100
One or more values out-of-range: <(-20.0, 100.0)>
Enter minimum/maximum latitude values: -20,20
Accepted ...
{'min': -20.0, 'max': 20.0}
Selected 11379 records.
```

```
Want to move on to longitude?
Please respond. Want to move on to longitude? y
```

```
SELECT longitude : enter two values separated by comma
range is -179.997 through 179.998
Enter minimum/maximum longitude values: 120,0
Accepted ...
{'min': 0.0, 'max': 120.0}
Selected 1645 records.
```

```
Want to move on to dates? ok
```

```
SELECT date mm/dd/yyyy: enter two values separated by comma
range is 01/02/1965 through 12/30/2016
Enter minimum/maximum date values: 1/1/1990,1/1/2000
Accepted ...
{'min': '01/01/1990', 'max': '01/01/2000'}
Selected 264 records.
```

```
Want to move on to magnitude? sure
```

```
SELECT magnitude : enter two values separated by comma
range is 5.5 through 9.1
Enter minimum/maximum magnitude values: 7,9
Accepted ...
{'min': 7.0, 'max': 9.0}
Selected 7 records.
```

```
Want to move on to analysis? no
```

```
SELECT magnitude : enter two values separated by comma
range is 5.5 through 9.1
Enter minimum/maximum magnitude values: 5,7
One or more values out-of-range: <(5.0, 7.0)>
Enter minimum/maximum magnitude values: 6,7
Accepted ...
{'min': 6.0, 'max': 7.0}
Selected 78 records.
```

### TASK #4: BUILD THE USER INTERFACE.

Build a user interface that prompts the user for the four parameters described above and selects earthquake data to analyze based on the user's choices. Once the data has been selected, you'll be ready to implement a variety of analyses.

Testing is important. Thinking about the data is important. You need to be able to determine what makes sense and what doesn't. If you have 5,000 data records and you select a subset of that, the size of the subset ought to be smaller. Hopefully, the many demonstrations I provide above will help you ascertain correctness, but there's really no substitute for doing your own thinking.

## PART 11: LOOKING AHEAD

Once the data of interest has been selected we move on to analysis. We can aggregate, analyze, and visualize the data in numerous ways. As we did in the histogram project earlier we can quantize a parameter simply count events. For example, if we quantize longitude, we could count the number of events occurring across a range of time zones. If we quantize latitude, we could count the number of events per climate region. If we quantize date, we could count the number of events per year (or month). If we quantize magnitude, we could count the number of events of various levels of severity.

What do I mean by "quantize"? I mean define the size of a parameter range (a "bin") that we can use to count events. For example, if our selected range is over a year, we could count the events occurring in each month. That gives us 12 "bins" to accumulate events in. In performing that analysis, we may accumulate the number of events, calculate average severity, and/or count the number of people affected. But the bins could be of any size ... we could divide the year into quarters if we like.

Anyway, I'm getting way ahead of myself here. The important thing is that data selection is an essential first step. It involves interacting with a user, reading the data files, organizing the data into convenient data structure(s), and selecting data records for analysis. We'll get into visualization next time.

## DATETIME DEMONSTRATION

```
1 # import datetime.datetime
2 from datetime import datetime as dt
3 ### CREATE DATETIME OBJECTS ###
4 # specify a date and time as a string
5 dtStr1 = "10/27/2011 5:45:00"
6 # convert dtStr1 to a datetime object using strptime()
7 date1 = dt.strptime(dtStr1, "%m/%d/%Y %H:%M:%S") # note the format
8 # date1 <class 'datetime.datetime'> has attributes (integers / not strings)
9 print(f"date attributes\n\t{date1.year=}\n\t{date1.month=}\n\t{date1.day=}")
10 print(f"time attributes\n\t{date1.hour=}\n\t{date1.minute=}\n\t{date1.second=}")
11 # date1 <class 'datetime.datetime'> has methods
12 print(f"method calls\n\t{date1.time()=}\n\t{date1.date()=}")
13 # another date string (no time given)
14 dtStr2 = "2012-2-27"
15 date2 = dt.strptime(dtStr2, "%Y-%m-%d")
16 # date2 methods ... if no time is specified, it's set to 00:00:00
17 print(f"method calls\n\t{date2.time()=}\n\t{date2.date()=}")
18 # Creating a datetime object directly ...
19 date3 = dt(year=2009, month=3, day=21, hour=12, minute=30)
20 # put the dates in a list ...
21 dateList = [date1, date2, date3]
22 ### FORMATTING ###
23 # print the list of dates ...
24 print("the three dates ...", *dateList, sep="\n\t")
25 # format the dates (use the strftime() method) and map() ...
26 dateStrings = map(lambda x:x.strftime("%Y-%m-%d"), dateList)
27 # print the dates from a list comprehension ...
28 print('dates formatted "%Y-%m-%d"')
29 _ = [ print('\t', dStr) for dStr in dateStrings ]
30 ### SORTING ###
31 # datetime objects can be sorted
32 dateList.sort()
33 # format the dates (use the strftime() method) and map() ...
34 dateStrings = map(lambda x:x.strftime("%m/%d/%Y"), dateList)
35 # print the dates from a list comprehension ...
36 print('dates sorted and formatted "%m/%d/%Y"')
37 _ = [ print('\t', dStr) for dStr in dateStrings ]
38 ### TIME DIFFERENCE ###
39 # Compute the difference between datetime objects (duration)
40 delta = date2 - date1
41 # duration is a timedelta object
42 print(f"{type(delta)=}")
43 # delta is a "timedelta" object <class 'datetime.timedelta'>
44 print("time difference")
45 print(f"\t{type(delta)=}\n\t{delta}")
46 # a timedelta object has attributes: days, seconds
47 print(f"time delta attributes\n\t{delta.days=}\n\t{delta.seconds=}")
```

## DATETIME DEMO RESULTS

```
date attributes
  datel.year=2011
  datel.month=10
  datel.day=27
time attributes
  datel.hour=5
  datel.minute=45
  datel.second=0
method calls
  datel.time()=datetime.time(5, 45)
  datel.date()=datetime.date(2011, 10, 27)
method calls
  date2.time()=datetime.time(0, 0)
  date2.date()=datetime.date(2012, 2, 27)
the three dates ...
  2011-10-27 05:45:00
  2012-02-27 00:00:00
  2009-03-21 12:30:00
dates formatted "%Y-%m-%d"
  2011-10-27
  2012-02-27
  2009-03-21
dates sorted and formatted "%m/%d/%Y"
  03/21/2009
  10/27/2011
  02/27/2012
type(delta)=<class 'datetime.timedelta'>
time difference
  type(delta)=<class 'datetime.timedelta'>
  122 days, 18:15:00
time delta attributes
  delta.days=122
  delta.seconds=65700
```

### Haversine Distance Function

```
2  from math import radians, cos, sin, asin, sqrt
3  def coord2rad(location):
4      '''
5      INPUT:
6          location : coordinate in degrees (tuple: Lat,Long)
7      RETURNS
8          location : coordinates in radians (dict: Lat,Long)
9      '''
10     coordinates = {'Lat':radians(location[0]),'Long':radians(location[1])}
11     return coordinates
12
13 def havDist(loc1, loc2, unit='km'):
14     '''
15     INPUTS:
16         loc1 : coordinate in degrees (tuple: Lat,Long)
17         loc2 : coordinate in degrees (tuple: Lat,Long)
18         unit : 'km' for kilometers, otherwise miles
19     Returns:
20         distance between two locations
21     '''
22     # convert coordinate tuples to dictionaries in radians
23     loc1 = coord2rad(loc1)
24     loc2 = coord2rad(loc2)
25     # Haversine formula
26     dlon = loc2['Long'] - loc1['Long']
27     dlat = loc2['Lat'] - loc1['Lat']
28     a = sin(dlat / 2)**2 + \
29         cos(loc1['Lat']) * cos(loc2['Lat']) * \
30         sin(dlon / 2)**2
31     c = 2 * asin(sqrt(a))
32     # Radius of earth in kilometers.
33     r = 6371 if unit == 'km' else 3956 # miles if not km
34     # calculate the result
35     return(c * r)
36
37 if __name__ == "__main__":
38     worcester = (42.2626, -71.798889)
39     opposite = (-42.2626, 180-71.798889)
40     boston = (42.361145, -71.057083)
```



Showing a location on a map

```
1  import folium
2  import webbrowser
3  import os
4
5  def generate_map(coordinates, output_file="map.html"):
6      # Create a map centered at the given latitude and longitude
7      my_map = folium.Map(location=coordinates, zoom_start=10)
8      # Add a marker at the specified location
9      folium.Marker(coordinates, popup="Your Location").add_to(my_map)
10     # Save the map as an HTML file
11     my_map.save(output_file)
12     # print(f"Map saved to {output_file}")
13     # Open the HTML file in the default web browser
14     webbrowser.open('file://' + os.path.realpath(output_file))
15
16  if __name__ == "__main__":
17     # Example usage:
18     worcester = (42.2626, -71.798889)
19     opposite = (-42.2626, 180-71.798889)
20     generate_map(worcester)
```