Lesson 23. Capstone project: analyze your friends

After reading lesson 23, you'll be able to

- Write a function to read a file line by line
- Save numbers and strings from the file in variables
- Write a function to analyze the stored information

The only two ways you've seen so far to input data are to (1) to predefine variables in your program or (2) to ask the user to input data one-by-one. But when users have a lot of information to input into your program, you can't expect them to enter it in real time. It's often useful to have them give you the information in a file.

Computers are great at doing many computations quickly. A natural use for computers is to write programs that can read in large amounts of data from files and to perform simple analyses on that data. For example, you can export your own data from Microsoft Excel spreadsheets as files, or you can download data (such as weather or election data). After you're given a file structured in a certain way, you can use knowledge of the structure to write a program to sequentially read and store the information from the file. With the data stored in your program, you can analyze it (for example, to find averages, maximums/minimums, and duplicates).

In addition to reviewing the concepts in this unit, this lesson will show you how to read data from a file.

The problem

Write a program that reads input from a file in a specific format, regarding all your friends' names and phone numbers. Your program should store that information and analyze it in some way. For example, you can show the user where their friends live based on the area code of the phone numbers, and the number of states where they live.

23.1. READING A FILE

You'll write a function named `read_file` to go through each line and put the information from each line into a variable.

23.1.1. File format

This function assumes that the user gives you information in the following format, with a different piece of information on each line:

```
Friend 1 name
Friend 1 phone number
Friend 2 name
Friend 2 phone number
<and so on>
```

It's important that each piece of information is on a separate line, implying that your program will have a newline character as the final character on each line. Python has a way to deal with this, as you'll soon see. Knowing this is the format, you can read the file line by line. You store every other line, starting with the first line, in a tuple. Then you store every other line, starting with the second line, in another tuple. The tuples look like this:

```
(Friend 1 name, Friend 2 name, <and so on>)
(Friend 1 phone, Friend 2 phone, <and so on>)
```

Notice that at index 0, both tuples store information regarding Friend 1; at index 1, both tuples store info regarding Friend 2, and so on.

You have to go through every line. This should trigger the idea to use a loop that goes through each line. The loop reads each line from the file as a string.

23.1.2. The newline character

A special hidden character is at the end of every line, the newline character. The representation of this character is `\n`. To see the effect of this character, type the following in your console:

```
print("no newline")
```

The console prints the phrase `no newline` and then gives you the prompt to type something in again. Now type in the following:

```
print("yes newline\n")
```

Now you see an extra empty line between what was printed and the next prompt. This is because the special character combination of the backslash and the letter *n* tells Python that you want a new line.

23.1.3. Remove the newline character

When you're reading a line from the file, the line contains all the characters you can see plus the newline character. You want to store everything except that special character, so you need to remove it before storing the information.

Because each line you read in is a string, you can use a string method on it. The easiest thing to do is to replace every occurrence of `\n` with the empty string "". This will effectively remove the newline character.

The following listing shows how to replace a newline character with an empty string and save the result into a variable.

Listing 23.1. Remove the newline character

```
word = "bird\n"
print(word)
word = word.replace("\n", "")
print(word)
```

- ***1* Creates a variable whose value is a string with a newline character**
- ***2* Prints the word with an extra newline**
- ***3* Replaces newline with an empty string, and assigns the result back to the same variable**
- ***4* Prints without an extra line**

Thinking like a programmer

What's intuitive to one programmer may not be to another. Often there's more than one way to write a piece of code. When faced with writing a line of code, browse the Python documentation to see what functions you can use before writing your own. For example, listing 23.1 replaces newline characters with the empty space character, using `replace` on strings. The Python documentation has another function that would be appropriate to use: `strip`. The `strip` function removes all instances of a certain character from the beginning and end of a string. The following two lines do the same thing:

```
word = word.replace("\n", "")
word = word.strip("\n")
```

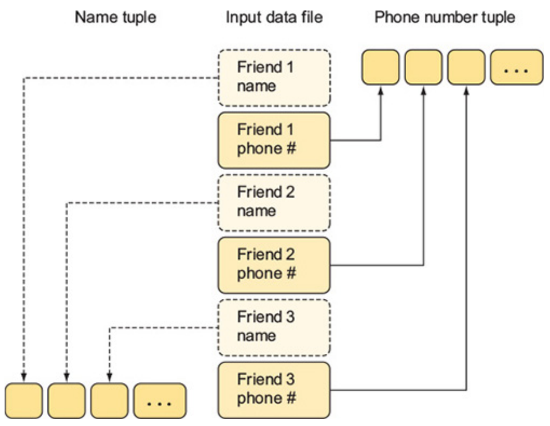
23.1.4. Using tuples to store information

Now that each line is cleaned up of newline characters, you're left with the pure data, as strings. The next step is to store it in variables. Because you'll have a collection of data, you should use one tuple to store all the names together and another tuple to store all the phone numbers together.

Every time you read in a line, add the new information to the tuple. Recall that adding an item to a tuple gives you a tuple that contains the old information, with the thing you just added at the end of the tuple. Now you have what the old tuple had, plus the new information you just read in that line. Figure 23.1 shows which lines of the file are stored in which tuple. In the next section, you'll see the code for this.

Figure 23.1. The input data contains lines of data. The first line is the name of a friend, and the second is that friend's phone number. The third is the name of your

from the second line, take all the phone numbers and store those in a separate tuple.



23.1.5. What to return

You're writing a function that does the simple task of reading a file, organizing the information, and giving the organized information back. Now that you have two tuples (one with all the names and the other with all the phone numbers, as shown in figure 23.1), return a tuple of tuples, like so:

```
((Friend1 Name, Friend2 Name, ...), (Friend1 phone, F
 ---------------------------------  -----------------
          one tuple                                oth
```

You have to return a tuple of tuples because a function can return only one thing. Recall from lesson 21 that returning a tuple with multiple elements allows you to get around this!

The following listing shows you the code for the function to read in the data. The function `read_file` takes in a file object; you'll see what this means later in this lesson. It iterates through every line in the file and strips the line of the newlines. If you're looking at an even numbered line, you add to your tuple of names. If you're looking at an odd numbered line, you add to your tuple of phone numbers. In both cases, notice that you're adding a singleton tuple, so you need to put an extra comma in the parentheses. Lastly, the function returns a tuple of tuples so that you can hand off the information parsed from the file.

Listing 23.2. Read names and phone numbers from a file

```
def read_file(file):
    """
    file, a file object
    Starting from the first line, it reads every 2 li
    and stores them in a tuple.
    Starting from the second line, it reads every 2 l
    and stores them in a tuple.
    Returns a tuple of the two tuples.
    """
    first_every_2 = ()
    second_every_2 = ()
    line_count = 0
    for line in file:
        stripped_line = line.replace("\n", "")
        if line_count%2 == 0:
            first_every_2 += (stripped_line,)
        elif line_count%2 == 1:
            second_every_2 += (stripped_line,)
        line_count += 1
    return (first_every_2, second_every_2)
```

- ***1*** **docstring**
- ***2*** **Empty tuples for names and phone numbers**
- ***3*** **Counter for line number**
- ***4*** **Loops through every line**
- ***5*** **Removes newline character**
- ***6*** **Odd-numbered lines**
- ***7*** **Adds to the names tuple**
- ***8*** **Even-numbered lines**
- ***9*** **Adds to the phone tuple**
- ***10*** **Increments line number**
- ***11*** **Returns a tuple of tuples**

23.2. SANITIZING USER INPUTS

You never specified the format of phone numbers, so the user can have a file that contains phone numbers in any format; users might have dashes, parentheses, spaces, or any other weird characters. Before you can analyze the numbers, you have to get them into a consistent form. This means removing all the special characters and leaving the digits all together.

This seems like a good job for a function. The function `sanitize` does this by using the `replace` method you learned about to replace all the special characters with the empty string "". The following listing shows a possible implementation. You iterate through each string and replace unnecessary characters that might be found in phone numbers. After removing dashes, spaces, and parentheses, you put the cleaned-up phone number (as a string) into the new tuple that you return.

Listing 23.3. Remove spaces, dashes, and parentheses from phone numbers

```
def sanitize(some_tuple):
    """
    phones, a tuple of strings
    Removes all spaces, dashes, and open/closed paren
    in each string
    Returns a tuple with cleaned up string elements
    """
    clean_string = ()                                1
    for st in some_tuple:
        st = st.replace(" ", "")
        st = st.replace("-", "")                     2
        st = st.replace("(", "")                     2
        st = st.replace(")", "")                     2
        clean_string += (st,)                        3
    return clean_string                              4
```

- ***1* Replaces unnecessary characters with empty string**
- ***2* Empty tuple**
- ***3* Adds cleaned number to new tuple**
- ***4* Returns new tuple**

23.3. TESTING AND DEBUGGING WHAT YOU HAVE SO FAR

The remainder of the larger task is to do analysis on this data. Before moving on to this, it's a good idea to do a little testing (and debugging, if necessary) to make sure the two functions you wrote work well together.

At this point, you have two functions that do a couple of interesting tasks. Recall that functions don't run until they're called somewhere in a larger program. Now you'll write code that integrates these functions together.

23.3.1. File objects

When you're working with files, you have to create file *objects*. As with other objects you've seen so far, Python knows how to work with these file objects to do specialized operations. For example, in the `read_file` function you wrote, you were able to write `for line in file` to iterate over each line in a specific file object.

23.3.2. Writing a text file with names and phone numbers

In Spyder, create a new file. Type in a few lines of data in the format that `read_file` expects. Start with a name; on the next line, put in a phone number, then another name, then another phone number, and so on. For example,

```
Bob
000 123-4567
Mom Bob
(890) 098-7654
Dad Bob
321-098-0000
```

Now save the file as friends.txt or any other name you want. Make sure you save the file in the same folder as the Python program you're writing. This file will be read by your program, so it's a plaintext file.

23.3.3. Opening files for reading

You create file objects by opening a filename. You use a function named `open`, which takes in a string with the filename you want to read. The file must be located in the same folder as your .py program file.

Listing 23.4 shows how to open a file, run the functions you wrote, and check that the functions return the correct thing. You use the `open` function to open a file named friends.txt. This creates a file object, which is the parameter to the function `read_file()`. `read_file()` returns a tuple of tuples. You store the return in two tuples: one for the names and one for the phones.

that the output is as you expect it to be.

Listing 23.4. Read names and phone numbers from a file

```
friends_file = open('friends.txt')
(names, phones) = read_file(friends_file)

print(names)
print(phones)
clean_phones = sanitize(phones)

print(clean_phones)
friends_file.close()
```

- ***1* Opens the file**
- ***2* Calls function**
- ***3* Outputs printed to the user**
- ***4* Sees whether your function worked**
- ***5* Outputs printed to the user**
- ***6* Closes the file**

It's good practice to test functions one by one before moving on to write more. Occasionally, you should test to make sure that the data being passed around from the output of one function to the input of another function works well together.

23.4. REUSING FUNCTIONS

The great thing about functions is that they're reusable. You don't have to write them to work for a specific kind of data; for example, if you write a program that adds two numbers, you can call the function to add two numbers representing temperatures, or ages, or weights.

You already wrote a function to read in data. You used the function to read in and store names and phone numbers into two tuples. You can reuse that function to read in a different set of data organized in the same format.

Because the user is going to give you phone numbers, suppose you have a file that contains area codes and the states to which they belong. The lines in this file are going to be in the same format as the file containing people names and their phone numbers:

```
Area code 1
State 1
Area code 2
State 2
<and so on>
```

The first few lines of a file called map_areacodes_states.txt are as follows:

```
201
New Jersey
202
Washington D.C.
203
Connecticut
204
<and so on>
```

With this file, you can call the same function, `read_data`, and store the returned value:

```
map_file = open('map_areacodes_states.txt')
(areacodes, places) = read_file(map_file)
```

23.5. ANALYZING THE INFORMATION

Now it's time to put everything together. You gathered all your data and stored it into variables. The data you now have is as follows:

- Names of people
- Phone numbers corresponding to each name
- Area codes
- States corresponding to the area codes

23.5.1. The specification

Write a function named `analyze_friends` that takes in your four tuples: the first is the names of friends, the second is their phone numbers, the third is all the area codes, and the fourth is all the places corresponding to the area codes.

```
Ana
801-456-789
Ben
609 4567890
Cory
(206)-345-2619
Danny
6095648765
```

Then the function will print this:

```
You have 4 friends!
They live in ('Utah', 'New Jersey', 'Washington')
```

Notice that even though you have four friends, two live in the same state, so you'll print only the unique states. The following is the docstring of the function you'll write:

```
def analyze_friends(names, phones, all_areacodes, all
    """
    names, a tuple of friend names
    phones, a tuple of phone numbers without special
    all_areacodes, a tuple of strings for the area co
    all_places, a tuple of strings for the US states
    Prints out how many friends you have and every un
    state that is represented by their phone numbers.
    """
```

23.5.2. Helper functions

The task of analyzing the information is complicated enough that you should write helper functions. *Helper functions* are functions that help another function achieve its task.

Unique area codes

The first helper function you'll write is `get_unique_area_codes`. It doesn't take in any parameters and returns a tuple of only the unique area codes, in any order. In other words, it doesn't duplicate area codes in a tuple of area codes.

Listing 23.5 shows the function. This function will be nested in the `analyze_friends` function. Because it's nested, this function knows of all parameters given to `analyze_friends`. This includes the `phones` tuple, meaning that you don't have to pass in this tuple as a parameter again to `get_unique_area_codes`.

The function iterates through every number in phones and looks only at the first three digits (the area code). It keeps track of all the area codes it has seen so far and adds it to the unique area codes tuple only if it isn't already in there.

Listing 23.5. Helper function to keep only unique area codes

```
def get_unique_area_codes():
    """

    Returns a tuple of all unique area codes in phone
    """

    area_codes = ()                                  1
    for ph in phones:                                2
        if ph[0:3] not in area_codes:                3
            area_codes += (ph[0:3],)                 4

    return area_codes
```

- ***1*** **Tuple to contain unique area codes**
- ***2*** **Goes through every area code, variable phones is a parameter to analyze_friends**
- ***3*** **Checks that area code isn't there**
- ***4*** **Concatenates tuple of unique codes with a singleton tuple**

Mapping area codes to states

Two of the inputs to `analyze_friends` are tuples containing area codes and states. Now you want to use these tuples to map each unique area code to its state. You can write another function that does this; call it `get_states`. The function takes in a tuple of area codes and returns a tuple of states corresponding to each area code. This function is also nested inside `analyze_friends`, so it will know of all the parameters given to `analyze_friends`.

area code tuple where the given area code is. You use the `index` method on tuples to get this value. Recall that the area code tuple and the states tuples match up (that's how we created them when we read them in from the file). You use the index you get from the area code tuple to look up the state at that same position in the state tuple.

A good programmer anticipates any possible problems with inputs from the user and tries to deal with them gracefully. For example, sometimes the user might enter a bogus area code. You anticipate that by writing code to the effect of something like "if you give me a bad area code, I will associate a state with it named BAD AREACODE."

Listing 23.6. Helper function to look up states from unique area codes

```
def get_states(some_areacodes):
    """

    some_areacodes, a tuple of area codes
    Returns a tuple of the states associated with tho
    """

    states = ()
    for ac in some_areacodes:
        if ac not in all_areacodes:                    1
            states += ("BAD AREACODE",)
        else:
            index = all_areacodes.index(ac)            2
            states += (all_places[index],)             3
    return states
```

- *1* **User gave you a bogus value; variable all_areacodes is a parameter to analyze_friends**
- *2* **Finds the position of the area code in tuple**
- *3* **Uses position to look up the state**

And that's it for the helper functions nested within the `analyze_friends` function. Now you can use them so that the code inside the `analyze_friends` function is simple and readable, as shown in the following listing. You just call the helper functions and print the information they return.

Listing 23.7. Body of the `analyze_friends` function

```
def analyze_friends(names, phones, all_areacodes, all
    """

    names, a tuple of friend names
    phones, a tuple of phone numbers without special
    all_areacodes, a tuple of strings for the area co
    all_places, a tuple of strings for the US states
    Prints out how many friends you have and every un
    state that is represented by their phone numbers.
    """

    def get_unique_area_codes():
        """
        Returns a tuple of all unique area codes in p
        """
        area_codes = ()
        for ph in phones:
            if ph[0:3] not in area_codes:
                area_codes += (ph[0:3],)

        return area_codes

    def get_states(some_areacodes):
        """
        some_area_codes, a tuple of area codes
        Returns a tuple of the states associated with
        """
        states = ()
        for ac in some_areacodes:
            if ac not in all_areacodes:
                states += ("BAD AREACODE",)
            else:
                index = all_areacodes.index(ac)
                states += (all_places[index],)
        return states

    num_friends = len(names)
    unique_area_codes = get_unique_area_codes()
    unique_states = get_states(unique_area_codes)

    print("You have", num_friends, "friends!")
    print("They live in", unique_states)
```

- ***1*** **Number of friends**
- ***2*** **Keeps only unique area codes**
- ***3*** **Gets states corresponding to the unique area codes**
- ***4*** **Prints number of friends**
- ***5*** **Prints the unique states**
- ***6*** **Nothing to return**

The final step of the program is to read the two files, call the function to analyze the data, and close the files. The following listing shows this.

Listing 23.8. Commands to read files, analyze content, and close files

```
friends_file = open('friends.txt')
(names, phones) = read_file(friends_file)
areacodes_file = open('map_areacodes_states.txt')
(areacodes, states) = read_file(areacodes_file)

clean_phones = sanitize(phones)
analyze_friends(names, clean_phones, areacodes, state

friends_file.close()
areacodes_file.close()
```

- ***1*** **Opens files in the same directory as the program**
- ***2*** **Uses the same function to read two different data sets**
- ***3*** **Normalizes the phone data**
- ***4*** **Calls the function that does most of the work**
- ***5*** **Closes the files**

SUMMARY

In this lesson, my objective was to teach you how to take on the problem of analyzing your friends' data. You wrote a few functions that specialized in doing certain tasks. One function read data from a file. You used that function twice: once to read in names and phone numbers, and another time to read in area codes and states. Another function cleaned up data by removing unnecessary characters from phone numbers. A final function analyzed the data you collected from the files. This function comprised two helper functions: one to return unique area codes from a set of area codes, and one to convert the unique area codes to their respective states. Here are the major takeaways:

- You can open files in Python to work with their contents (to read lines as strings).
- Functions are useful for organizing code. You can reuse any function you wrote with different inputs.
- You should test your functions often. Write a function and immediately test it. When you have a couple of functions, make sure they work well together.
- You can nest functions inside other functions if the nested functions are relevant to only a specific task, as opposed to the program as a whole.