Lecture 18

File Input

13 Germinal, Year CCXXX

Song of the day: ***La Javanaise*** *by Khatia Buniatishvili (2020).*

Part 1: *Warm-up*

Let's warm up with a quick problem, very similar to the one we did last week.

Let's say we have a list of strings, and each string contained a series of numbers separated by a single blank space:

```
sample_list = [
    '50.98 82.72 89.18 51.57 23.95 69.82',
    '57.7 13.08 1.26 6.15 52.09 42.63 39.46',
    '96.21 43.32 79.45 7.87 10.5 10.92 67.87 21.22',
    '27.27 40.23 79.09 17.56 75.87 80.38 40.98 6.21 44.72 36.45',
    '0.07 29.63 97.73 58.01 97.47 24.07 83.46 99.4',
    '14.03 55.63 31.57 0.01 73.4 91.35 82.06 59.62 2.83 93.04'
]
```

Your goal to create a function, `get_sums()`, that accepts such a list, and returns a list of their sums:

```
sample_list = [
    '50.98 82.72 89.18 51.57 23.95 69.82',
    '57.7 13.08 1.26 6.15 52.09 42.63 39.46',
    '96.21 43.32 79.45 7.87 10.5 10.92 67.87 21.22',
    '27.27 40.23 79.09 17.56 75.87 80.38 40.98 6.21 44.72 36.45',
    '0.07 29.63 97.73 58.01 97.47 24.07 83.46 99.4',
    '14.03 55.63 31.57 0.01 73.4 91.35 82.06 59.62 2.83 93.04'
]

sums = get_sums(sample_list)

print(sums)
```

Output:

```
[368.22, 212.37, 337.36, 448.76, 489.84, 503.54]
```

My first step would be to recognise that we're going to have to repeat the same provess for every single element in this list. Moreover, we're going to have to work on each of the strings in order to make them `float` values, and then add them together. We could do this starting from the outside (working from the outside list toward its contents), or work from the inside in —that is, thinking about the process we're going to have to perform on each of the strings.

I'm going to go with the latter process. Let's write a function that takes a single string of numbers separated by a single space, and returns the sum of those numbers. Let's also forbid the use of the Python built-in function `sum()`:

```
def get_line_sum(string):
    pass
```

Our first is to split the string into a list of number strings. The `split()` function is perfect for this:

```
def get_line_sum(string):
    list = string.split(' ')
```

Since we're barring the use of the `sum()` function, we're going to have to do it manually using a `for`-loop. Remember that we have to cast each number into a `float` before adding it to the total. Finally, we simply return that sum:

```
def get_line_sum(string):
    string = string.split(' ')
    summation = 0

    for element in string:
        element = float(element)
        summation += element

    return summation
```

***Code Block 1**: A function that returns the sum of the numbers contained in a string, separated by blank spaces.*

Awesome. That's simple enough. Now, we work on the outer step: iterating through each string, getting its individual sum, and appending it to a list of sums:

```
def get_sums(lst):
    sums = []

    for string in lst:
        summation = get_line_sum(string)
        sums.append(summation)

    return sums
```

Let's run this, and see if it works as we intended:

```
[368.21999999999997, 212.37000000000003, 337.36, 448.75999999999993, 489.8399999999999, 503.54]
```

Not bad. We are running into that pesky approximation behavior that `float` values occasionally have. Let's add a call to the `round()` function:

```
def get_sums(lst):
    sums = []

    for string in lst:
        summation = get_line_sum(string)
        sums.append(round(summation, 2))
```

```
    return sums

def main():
    sample_list = [
        '50.98 82.72 89.18 51.57 23.95 69.82',
        '57.7 13.08 1.26 6.15 52.09 42.63 39.46',
        '96.21 43.32 79.45 7.87 10.5 10.92 67.87 21.22',
        '27.27 40.23 79.09 17.56 75.87 80.38 40.98 6.21 44.72 36.45',
        '0.07 29.63 97.73 58.01 97.47 24.07 83.46 99.4',
        '14.03 55.63 31.57 0.01 73.4 91.35 82.06 59.62 2.83 93.04'
    ]

    sums = get_sums(sample_list)

    print(sums)

main()
```

Output:

```
[368.22, 212.37, 337.36, 448.76, 489.84, 503.54]
```

Perfect. This is actually a great setup to our next topic: reading and using the contents of files in Python.

Part 2: *Data from Files*

Imagine if the number strings had not existed in a Python list, but rather in a files called **student_grades.txt**:

```
001 50.98,82.72,89.18,51.57,23.95,69.82
002 57.7,13.08,1.26,6.15,52.09,42.63,39.46
003 96.21,43.32,79.45,7.87,10.5,10.92,67.87,21.22
004 27.27,40.23,79.09,17.56,75.87,80.38,40.98,6.21,44.72,36.45
005 0.07,29.63,97.73,58.01,97.47,24.07,83.46,99.4
007 14.03,55.63,31.57,0.01,73.4,91.35,82.06,59.62,2.83,93.04
```

Inside the file, we see the same list of numbers, separated by a comma `,` , with a certain ID at the beginning of each line (e.g. `001` , `002` , etc.).

In reality, most real-world data comes from outside the realm of Python—we simply use Python as a tool to get what we want out of the information. So how to we "import" this data into our program?

Well, the first step, of course, is to open it. We do this using the Python built-in function:

Step 1: *Open the File*

```
file_path = "solutions/student_grades.txt"
file_obj = open(file_path, 'r')
```

So, what's going on here? The `open()` function (as far as this course is concerned) accepts to arguments:

1. The **path of the file** you want to open, in `str` form. Since `student_grades.txt` exists in the same directory (folder) as our current file, we don't have to give the full filepath, but I certainly could do so (

`/Users/sebastianromerocruz/Documents/NYU_Adjunct/2021_Fall/Lectures/Week_11_1/student_grades.txt`) and it would work just the same.
2. The **mode that you want to open the file with**. In order to read the contents of a file, we use `'r'` , which stands for "read mode".

Notice that it's not enough to just make a call to `open()` —we also have to save its returned value into a variable (in this case, the variable `file_obj`).

Step 2: *Operate on The File*

Once we have this file open, we can operate on it in a number of ways. Here are some of the file methods that we can, and will use, in this class:

Method	Example	Description
`read()`	`file_obj.read()`	Reads the entire file as a single string.
`readline()`	`file_obj.readline()`	Returns the next line of the file with all text up to and including the newline character.
`readlines()`	`file_obj.readlines()`	Returns a list of strings, each representing a single line of the file

***Figure 1**: File object methods. See full documentation **here**.*

So how does Python know when a new line starts? For us, it is easy to tell by visually recognising that a line exists below another, but programming languages need specific instructions as to how to recognize these things. It turns out that if you extract a line from `student_grades.txt` and print it, it looks like this:

```
file_path = "solutions/student_grades.txt"
file_obj = open(file_path, 'r')

first_line = file_obj.readline()
print(first_line)
```

Output:

```
'001 50.98,82.72,89.18,51.57,23.95,69.82\n'
```

What do we see at the end of the line? That's right: a `\n` character! The same way we print a new line using, say, the `print()` function, Python reads this "hidden" character and recognizes that this represents a break in the line.

So what if we tried to run this line through our `get_line_sum()` function?

```
file_path = "solutions/student_grades.txt"
file_obj = open(file_path, 'r')

first_line = file_obj.readline()
id = first_line.split()[0]  # before the space
grades = first_line.split()[1]  # after the space

print(get_line_sum(grades))
```

Output:

```
Traceback (most recent call last):
  File "get_sums.py", line 52, in <module>
    main()
  File "get_sums.py", line 44, in main
    print(get_line_sum('001 50.98,82.72,89.18,51.57,23.95,69.82\n'))
  File "get_sums.py", line 12, in get_line_sum
    element = float(element)
ValueError: could not convert string to float: '50.98,82.72,89.18,51.57,23.95,69.82\n'
```

Ah. Looks like we've got an error; specifically, it looks like one of the elements is having trouble being casted into a float. You can probably guess which one it is at this point: the last one, `69.82\n`. Remember that casting ***only*** works when the string contains ***only*** a float or integer. In this case, it contains a newline character, so the casting fails.

It is almost always the case that you will have trailing (or preceding) whitespace characters when reading lines from a file (e.g. `' '`, `'\n'`, `'\t'`), so it'd be great to have a quick way to get rid of it. The string method `strip()` does just the job:

```
string_with_whitespace = "\t    Hello, World!\n\n   \n"
string_without_whitespace = string_with_whitespace.strip()

print(string_without_whitespace)
```

Output:

```
Hello, World!
```

As you can see, `strip()` removes both whitespace the precedes the first non-whitespace character, and whitespaces that trails after the last non-whitespace character. This is the perfect function for our purposes. So let's use it:

```
file_path = "solutions/student_grades.txt"
file_obj = open(file_path, 'r')

first_line = file_obj.readline()
first_line = first_line.strip()

id = first_line.split()[0]  # before the space
grades = first_line.split()[1]  # after the space

print(get_line_sum(grades))
```

Output:

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 12, in get_line_sum
ValueError: could not convert string to float: '50.98,82.72,89.18,51.57,23.95,69.82'
```

Hm. What's going on now? It turns out that now our `get_line_sum()` is in the wrong here. Remember that we're splitting by spaces `' '` in `get_line_sum()`, where our numbers are now split by commas `,`. So let's add another parameter to our `get_line_sum()`, called, separator, that gives us the option of how we want to split our string:

```
def get_line_sum(string, separator=' '):
    string = string.split(separator)
    summation = 0

    for element in string:
        element = float(element)
        summation += element

    return summation
```

Nice. *Now* let's see if it works:

```
file_path = "solutions/student_grades.txt"
file_obj = open(file_path, 'r')

first_line = file_obj.readline()
first_line = first_line.strip()

id = first_line.split()[0]  # before the space
grades = first_line.split()[1]  # after the space

print(get_line_sum(grades))
```

Output:

```
368.21999999999997
```

Theeeeere we go. So, let's say we wanted to write a new function, `get_file_sums()`, that instead of accepting a list of strings, it accepted a filepath string. We'd only need to change a few lines from our `get_sums()` function. Let's try to return a tuple with two elements. The first is going to be the student ID, and the second the sum of grades:

```
def get_file_sums(filepath):
    file_obj = open(filepath, 'r')

    sums = []

    for string in file_obj:
        string = string.split()
        student_id = string[0]
        grades = string[1]

        summation = get_line_sum(grades)
        student_info = (student_id, round(summation, 2))
        sums.append(student_info)

    return sums
```

That's right! It turns out that file objects can be used as sequences that we can iterate over using a `for`-loop, just like we do with ranges, strings, and lists. This makes working with each individual line extremely simple. Now...

Step 3: ***Close the file (Please).***

DON'T RUN THIS YET.

We have to do one last, absolutely critical and indispensible step, and that is to ***close our file***:

```
def get_file_sums(filepath):
    file_obj = open(filepath, 'r')

    sums = []

    for string in file_obj:
        string = string.split()
        student_id = string[0]
        grades = string[1]

        summation = get_line_sum(grades)
        student_info = (student_id, round(summation, 2))
        sums.append(student_info)

    file_obj.close()  # right here!

    return sums
```

It is absolutely imperative that you close your file. If you don't, any other function, process, and/or app that wants to make use of this app will not be able to, causing the whole operation to fail. Please don't forget. Burn it into your brain.

Alright, let's test it now:

```
filepath = 'solutions/student_grades.txt'
file_sums = get_file_sums(filepath)
print(file_sums)
```

Output:

```
[('001', 368.22), ('002', 212.37), ('003', 337.36), ('004', 448.76), ('005', 489.84), ('007', 503.54)]
```

Niiiice.

Part 3: *Command Line and Command Line Arguments (OPTIONAL)*

You know how I've been running my Python programs through PyCharm's Terminal as such?

```
python3 class.py
```

This actually has a more recognizable equivalent. For example, whenever you click on an application on your phone, somewhere in its operating system, a similar command such as the following would be being executed:

```
iphone instagram.app
```

That is, something like:

> Using an `iPhone`, run the `app` called `instagram`.

The Terminal is a form of a **command line**—a non-graphic, text-based user interface. It's difficult to overstate the importance of being able to run your programs using the command line instead of by clicking on an icon in a GUI (graphic user interface) like PyCharm, or your Desktop icons.

For one, we're training to become *programmers*, the very people who *make* these GUIs. Second, the command line allows you to speak to your computer with much more finesse and control than any GUI will ever allow. Operating systems like MacOS make it a point so that its users never have to bother with thinking about the inner workings of their machine. This is both a blessing and a curse; it is a blessing because not everybody is good with computers (I'm certainly not), but it is also a curse because oftentimes we have trouble finding and organizing our files. This is why programmers tend to prefer operating systems such as Linux, which give the user and absurd amount of control over how their environments behave. You eventually learn which one works best for you; a iOS app developer may not ever need to deal with the command line on a day-to-day basis—they have the miracle of XCode instead. A cybersecurity engineering, however, will absolutely want to know how their programs are working deep in the innards of their computers, and the command line is their first stop to do this.