David Brady

February 15th, 2021

Foundations of Programming: Python

Assignment 05

**Dictionaries and Best Practices**

**Overview**

Today we are introducing not only new techniques that we will utilize to create deeper functionality within user manipulated programs; including Dictionaries and a deeper exploration into the functionality of Lists- but we will additionally work towards becoming more capable and effective programmers by learning to get started quickly with Templates, catch user created errors and fix them on the spot with Try-Exceptions, learn to work using the design principle known as Separation of Concerns, upload our project to Github for review and more. We will take a second to overview some of the benefits of each of these ideas as we run through the tools needed to complete our Module Assignment.

**Reviewing Lists and Featuring Dictionaries**

Lists as we have seen are an easy, mutable way to hold a collection of objects. They have built in functions and hold and present data in memory. Dictionaries have *“Key”: “Value*” “pairs” and use **curly braces{}** to contain their information. So instead of storing information as a sequence of indexes, we store it in pairs- looking up a key word gets its definition or “value” with a dictionary.

One of the most common things to do with a dictionary is to search for its values. The simplest method of accessing a value is to search for its ”key” as follows:

Geek = {“404”: “Broke”}

Geek = [“404”]

When searching a dictionary we use similar techniques as we would when accessing an index, but instead of the index location, we search the dictionaries “key”- returning the keys “value”. No position numbers for dictionaries! Keep in mind a value cannot be used to retrieve a key in this way.

**Reading Data from Files into Lists and Dictionaries**

Lists are simple, but they contain many useful functions for developers- and because of this we often load text file data into them. It is valuable at this junction to speak a bit about split() and strip() in some detail.

Split() is used to *separate elements* of a new list based on comas found in the text file.

As in Split(“,”)

Strip() functions will remove unwanted carriage returns in script.

We load the file first into a variable, then we create a list to hold, edit and present the data within. We may first split data into a row and use a for loop and split function to separate inner variables.

Dictionary keys will work like columns in a spread sheet or data base, thinking of a dictionary as a row of data can be helpful. Dictionaries have built in functions to facilitate this, i.e; items(), values() and keys(). These will help but they will not do the work for us!

We can create lists of dictionaries as well:

list\_table = [dict\_row, dictrow2]

When uploading a file into a dictionary first we load the file into a variable. Declaring our dictionary and lists beforehand can be helpful in this section of our code. Next we open our file and select our access mode. We then create a for loop to cycle through rows in our file. We can split our Key: Value pairs using split(“,”) to split them by comas and pull our dictionary variable into our for loop and assign our files information into rows using a row variable. We then call our dictionary variable and insert indexes from our file into keys- we can do this by naming keys and inserting file data via indexes. We can append the finished dictionary to a list and close our file from here.

**Quick review of Access Modes and File Methods**

When working with files we append “access\_modes” to our built-in functions. These will reflect and implement our intentions with a file, and though we have gone over them previously, it will not take long to review our options:

Open(file\_address, “**access\_mode”**)

Access\_Modes include:

“r”: Opens a file for read only.

“r+”: Opens a file for reading and writing.

“w”: Opens a file for writing only.

“w+”: Opens a file for writing and reading.

“a”: Opens a file for appending.

“a+”: Opens a file for appending and reading.

Reading a file may appear like so:

F = open(\_file\_,”r”)

text = f.read(10)

print(text)

f.close()

File methods could also use some light review and include open(), read(), write() and close().

Read([size]) reads an entire file and returns its contents in the form of a string. In the absence of a size argument, it reads until EOF (end of file). Write() writes content of a string to file and has no return value, while close() is used to close a file. A closed file cannot be read or written upon after it has been closed.

**Readability, Efficiency and Best Practices**

When attempting to improve as a programmer overall, it is easy to pick a direction and walk when first starting out- but sometimes it can be difficult with any new skill to see what is going to be useful down the road and what isn’t. Recognizing the value in version control, Separation of Concerns, Templates and Try-Exceptions will go a long way towards investing in skills that will pay off down the road, as all of these are language agnostic best practices.

SoC or **Separation of Concerns** is a syntax agnostic concept design principle. It separates our program into organized groups of designated code that solves unique issues in our program. The concern is technically a “set of information that affects the code of a computer program”. Data, Processing, Presentation (I/O). This is a common best practice organizational layout for Separation of Concerns. Successfully practicing this will yield more readable and aesthetically satisfying code. Extremely cool! Especially considering practicing this now in one language will yield great dividends in all languages and projects in our future.

It is worth mentioning that we have not worked with **Functions** this week, but they can still be a big part of organizing our code and we still have access to some uses for them and explanations in our notes and chapter. Making code compact can make it more readable and more aesthetically pleasing. Paying attention to how many lines we accumulate rewriting statements, we will begin to see within large programs that we could save a lot of time, effort and space by writing a set of programming statements into a function name that we can quickly and easily call at any time we need them. This is the purpose of functions. PEP8 recommends we use a lower case/underscore notation to write functions(my\_functions) but in general it is most important to remember we can create functions in our processing sections and utilize them in our presentation, or however it is most appropriate.

Creating a custom **Template** is common across multiple languages and IDE’s, even fields of study outside coding- templates can be such a fast way to get started no matter how much experience you have. It is a common practice to save them and use them for multiple projects, they work to hold up consistency in your work and writing patterns as a sort of prompt. An easy first template to organize for yourself may be to create a template that includes your module headers and your separations of concerns. It will improve your professionalism, development speed and consistency (readability) if implemented successfully.

When a user runs into an error using your program, often times the error message may go over their head, even if it’s a simple problem they could resolving by changing their user input. A **Try-Except** clause fits nicely in the conditionals within our program, and the result they yield helps everyone. By adding the “except” clause to our conditionals, we may plan as programmers for specific errors from our users input and write messages that they can read without understanding the inner workings of our program.

Ideally this will save a programmer time by directly telling a user how and why their error occurred, and saves the user time by telling them immediately- in a way they can understand- what we need to communicate to them in order for them to resolve their issue immediately!

**Summary**

We learned a LOT this week about the philosophy of becoming better programmers through exploring what developers know to be best practices. Our ability to produce code should be improved in every way, which means even if our assignments get larger and more complicated- we should have all the tools we need to expand and complete our tasks even faster than we have previously- and yield results that are easier for others to read, and for ourselves to return to later and read, or edit.