Data Science for AAE

AAE 718 – Summer 2024

Worksheet 1

Python Basics

1 Reading

1.1 Python Data Science Handbook

- Chapter 1 – Skim to be sure to understand iPython and Jupyter

This worksheet has far more words than normal as the book assumes a basic knowledge of Python.

2 Daily Goals

- Install VSCode, Anaconda and R
- Get Jupyter notebooks working in VSCode and install Python VSCode extension
- Basic Python syntax
  - Lists, Dictionaries, Tuples
  - Conditions (if-elif-else)
  - Loops
  - List comprehension
- Functions
- Strings
- Figure out how to submit homework

3 Submitting Homework

Homework is submitted as a Python file ".py" directly on Canvas. The file should contain only solutions to the homework and any dependecies. For example, if there were two problems one to write a function to compute the square root of a number an another to print a formatted string. You would submit

```
from math import sqrt

def problem_1(x):
    return sqrt(x)

def problem_2(input_string):
    out = f"The input is: {input_string}"
    return out

if __name__ == "__main__":
    #you can test stuff here. It won't run if the file is imported.
    pass
```

The problems will tell you what to name the functions, what inputs should be expected and what output is required. There will occasionally be a provided data file as well, so you can check your answers before submission.

To grade these, I will download the files and test against a large dataset. I'll be looking for correctness and if there are errors. I will also examine your code to ensure things are working well.

It will be easy to copy code from another source or students. I'll know and the penalty will be a 0 for all

parties.

Sometimes you'll also need to upload PDFs. It'll be clear when and I highly recommend using LaTeX to create the PDFs. Markdown is also also acceptable.

Every problem should have a function, if the problem doesn't specify a name just call it `problem_{problem_number}`. Some problems are just writing, in that case have the function

```
def problem_04()
    pass
```

Be sure the problems are in the order assigned. This just makes it easier for me to read.

Do not have print statements your functions.

4 Script file vs Jupyter Notebook

A standard python script file has the extension `.py` and is a raw text file. These files are useful for storing reusable code as they can be imported into other files.

A Jupyter notebook has the extension `.ipynb` and is a JSON file. These are extremely useful to test code or write "one-use code". One-use code is primarily what we write for Data Science, so we use them a lot. Examples are data cleaning (although, this can be in a script and loaded), plotting, and sharing code.

We will use both file types in this class, but homework is always submitted as a script file (.py). Primarily so I can read the files in Canvas and easily import them to another file.

5 VS Code

You should be using VS code for both Jupyter notebooks and script files. I know Jupyter notebooks open in a browser window by default and you may be used to this. Try it my way and you will realize it's better.

To get setup open the *extensions* menu in VSCode, it's on the left and looks like 4 squares (it's the fourth one down in my version). Search for *Python* first and install it. Then search for *Jupyter* and install it.

To open a new Jupyter notebook press control + shift + P[1]. This will open a very important window from which you can do pretty much anything. For us type Jupyter and you'll get the option to open a new Jupyter notebook.

In the upper right of the notebook you should see "Select Kernel" click this and you can find Python. In the future you may have several different Python environments, things with different packages or versions of Python, this lets you switch between those easily.

6 Organization

I highly recommend you stay organized as you will have a lot of files in this course. You should create a folder in your Documents called `AAE_718`, or something. Inside that have folders for each worksheet where you put the relevant files.

I also recommend having another subdirectory called "data" where you put raw data. I've had projects with 20 data files and having these hidden away from my code is much less annoying.

Don't just put all the files for this course in a single folder[2].

[1] It may be command rather than control on Mac, I don't know test it

[2] If I find out everything is on your desktop, I may fail you. I'm being sarcastic, but only a little.

7 Very Basic Python

Open a new Jupyter notebook as described above. I like to test code in a notebook before transferring to a .py file. At it's core, Python is a big calculator, type `2+2` and press Shift+Enter. The Shift+Enter in Jupyter runs the current cell and moves to the next cell. You should see an output of 4 and a second cell. A Jupyter notebook consists of cells of code that can be independently run, but all share memory[3]. Move your mouse over the cells and you'll see buttons to add code or markdown cells. We'll discuss markdown later. If you want a bigger overview of Jupyter here is a reference.

In the next cell run `x=2`. You should get a new cell with nothing displayed. In this new cell run `x` and you'll see a 2. This is because you've stored x as 2. You can now use this in expressions. For example you can evaluate $2x^2 + 3x - 7$ by typing `2*x**2 + 3*x - 7`. A couple of things to notice, you need to be explicit about multiplication and the power is `**` rather than `^`.

8 Data Structures

A data structure is a way to store data[4]. There are three primary data structures in base Python[5]: lists, dictionaries, and tuples. We'll discuss the advantages of each.

8.1 Lists

To create a list use the syntax `L = [1,2,3]`. You can extract elements from the list using `L[0]`, `L[1]`, `L[2]`, or `L[3]`. You should type these in and see the error you get on the third one. Your list only has three items and Python starts counting at 0, so the maximum value is 2. You can view the size of a list using `len(L)`.

You can add items to a list using the syntax `L.append(10)`. If you want to add multiple items to a list you may thing it's `L.append(10,11,12)`, but this will give an error. So you try `L.append([10,11,12])`, this doesn't give an error, but if you look at L now, it's clearly wrong. The correct syntax is `L.extend([10,11,12])`. For a full list of list operations, see this link.

One thing to point out about Jupyter notebooks that can creep up on you, if you've run all the commands from the previous paragraph in different cells, you list L will keep getting bigger an changing. This isn't always desired. My general rule of thumb is that if I know something is going to change, I try to do all the changes in a single cell so that it's reproducible. For example, my cell might look like

```
L = [1,2,3]
L.append(10)
L.extend([10,11,12])
```

You can run this with Shift+Enter from anywhere in the cell, not just the end.

In this example I've only put numbers inside a list, but the values can be anything. You can have a list of lists, or a list of floats, or a list of functions. Literally anything.

You'll end up using lists all the time. They are easy to use and flexible. However, be warned that lists in Python are *SLOW*. We'll have faster data structures next time.

8.2 Dictionaries

Lists are nice, but they are only indexed by numbers. Dictionaries are like a list, but can be indexed by may more things. Here is an example of a dictionary, `D = {"a":1, "b":2}`, notice the {,} rather than [,]. The `"a"` is a string, covered below. You can retrieve information using `D["a"]`.

[3] You'll see

[4] This sounded more profound in my brain

[5] Not really true, but it's enough for us

Dictionaries are a key-value pair. In this example `"a"` is a key and 1 is the value. Much like lists, values can be anything. However, the keys are more restrictive. Keys must be *immutable*, for now take this to mean "lists can't be keys in a dictionary". This won't come up too much in practice, but you should be aware of it. The keys must also be unique in a dictionary, otherwise you'd overwrite your value.

Adding to dictionaries is very easy, `D[7] = 10`. Now 7 is a key with value 10. We can change that to be `D[7] = [1,2,3]`. Now 7 have a list as a value. Here is too much information on dictionaries. You'll use dictionaries all the time too.

8.3 Tuples

To make a tuple use (,), for example `T = (1,2,3)`[6]. Much like lists, you can retrieve values using `T[0]`.

Tuples are very similar to lists, with one big exception. Try running `T[0] = 7` and you get an error. Tuples are *immutable*, which means they can never change. It also means they can be the keys to a dictionary, I use this a lot.

You can convert a list to a tuple using *tuple(L)* or a tuple to a list using *list(T)*. This isn't used as much as you may think.

The values of a tuple can be anything, including tuples. You can also *unpack* tuples. If `T = ("a", 2, "banana")` you can do `a,b,c = T`. Check the values of a, b, and c. You'll see where this is really useful.

9 Loops

The most basic loop in Python[7] is the *for* loop. I'm going to type something strange looking for the general for loop syntax:

```
for i in ITERATOR:
    code using i
```

First, the `ITERATOR` is any object that supports iteration, we'll see examples. Second, the `:` designates the start of the code. Third, the white space is VERY important, you should be using TAB that gets replaced with four spaces. You can change this in VSCode, in the bottom you should see something like "Spaces: 4" or "Tab Size: 4". If you see "Spaces: 4" then you're good. If not, click it, select Indent using Spaces and 4.

Let's do an actual example of a for loop. First, we'll need a function called *range*. If you type `range(10)` it's quite boring, but make it a list *listrange(10)* and you'll see it's the numbers 0 to 9. Notice there are 10 total numbers, it starts at 0 and ends at 9. At this point you should recall how list indexing works and notice it matches.

Now for the example:

```
for i in range(10):
    x = i**2
    print(x)
```

When you run this you should see the numbers 0 to 9 squared. The for loop walks over the iterator, in this case `range(10)`. The first element of `range(10)` is 0, so i starts with the value of 0 and then runs through the for loop code with $i = 0$. At the end (or when the tabs stop), i becomes the next value in the iterator, which is 1. It continues this until it reaches the end of the iterator, in this case 9.

The previous example should give you a clear understand of what a for loop is doing. If not, now is your moment to ask. We'll be using these this often.

[6] Apparently you don't need the parentheses, but I recommend it

[7] And most languages

In practice, we usually won't be using `range` as our iterator, we'll be using something more useful. Here is a pretty complex example,

```
L = ["a", 2, "c", 17]
out = []
for i in L:
    out.append(i*2)
print(out)
```

In this example the iterator is L, which is a list we've defined. The code builds a list, in this example it's just doubling the value of each element of L. You need to understand what this is doing. If you do not, ask.

You can iterate over Tuples just like lists. You won't notice any difference.

Dictionaries are a little trickier. By default, dictionaries iterator over the keys.

```
D = {"a":1,"c":10}

for key in D:
    print(key)
```

The output is just the keys. Typically you'll want the values, this can be accomplished either using

```
D = {"a":1,"c":10}

for key in D:
    print(D[key])
```

Or

```
D = {"a":1,"c":10}

for value in D.values():
    print(value)
```

Or, if you want both the key and value

```
D = {"a":1,"c":10}

for key,value in D.items():
    print(key, value)
```

I use each of these regularly, although I usually default to `D.items()`.

Python also has `while` loops. However, they are less stable as it's easier to enter an infinite loop. I default to for loops.

10 If statements

Sometimes you only want to run a piece of code if some condition is true. Hence the `if` statement.

```
if CONDITION:
    Some Code
elif OTHER CONDITION:
    Other Code
else:
    Final Code
```

First, the `elif` and `else` are optional you can exclude these. You can also have many `elif` statements. Second, just type and don't worry about tabs. Any good editor will figure it out. Seriously try it.

Here is an example with a for loop.

```
for i in range(10):
    if 3<i and i<8:
        print(i)
    elif i<=3:
        print(i**2)
    else:
        print("Sad day")
```

Again, understand the output. If you do not, you need to ask.

There are many different conditional options. Here are a few more

```
if 1==2:
    print("I won't print")

L = [1,2,3]
if 2 in L:
    print("I'll print!")

if 1 == 1 and 2 in L:
    print("I'll also print")
```

If you can form a condition in your brain, you can translate it to code.

11 Comprehension

Comprehensions are easy ways to build lists or dictionaries. They are basically a for loop inside a list.

```
[i**2 for i in range(10)]
```

This should be pretty self-explanatory based on the output. You can also put a conditional in there,

```
[i**2 for i in range(10) if i<4 or 7<i]
```

Figure this out, if it's giving you trouble change `i**2` to `i`.

Dictionaries are similar,

```
{i:i**2 for i in range(10) if i<4 or 7<i}
```

Remember that a dictionary needs both a key and value. This should make clear the previous example too.

12 Brief Functions

Here is a really basic function,

```
def f(x):
    return x**2
```

In this function, x is the input and the output is x^2, which is computed as $x**2$ in Python. You can have multiple inputs, for example

```
def g(x,y):
    return x*y
```

I could use this function in a list comprehension,

```
[f(i) for i in range(10)]
```

Or in a conditional,

```
[i for i in range(10) if f(i)<50]
```

This will give me all the numbers whose square is less than 50. In this example it would probably be better to have `i**2` be explicit (or have call f something better) for clarity.

13 Strings

Occasionally you have a lot of data that you want to display nicely. For example, let's say we have a function,

```
def f(x):
    return 3*x-x**2
```

This is a simple example, but it'll be nice to use. Let's look at this function evaluated at a few points.

```
L = [1,2,3,4]
for i in L:
    print(f(i))
```

The output to this is pretty basic, a line of evaluated points. But we don't know the inputs. There are a few ways to do this.

```
L = [1,2,35,400]
for i in L:
    print(i,f(i))

print('-'*50)

for i in L:
    print(f"i={i}, f(i)={f(i)}")

print('-'*50)

for i in L:
    print(f"i={i:3d}, f(i)={f(i):4f}")
```

There are a few things going on here, try to analyze which you like best. The important thing to notice is the *format* string, designated by leading `f`. The short story is that any $\{x\}$ in the string gets replaced by the variable `x`. Format can also specially format things, as seen in the last example, for the i, it's reserving 3 digits and for the $f(i)$ 4 float decimal digits. This makes tables much easier to read. Try adding 1402 to L, or a float. See what happens.

Format is way more useful than just this.

```
print("Hello {name}".format(name = "Mitch"))

d = {"name":"Mitch"}
print("Hello {name}".format(**d))
```

```
name = "Mitch"
print(f"Hello {name}")
```

The second one, with **d *unpacks* the dictionary which makes it look like the first. The third uses a new *format string*, this knows what variables have been defined and inserts them.

Another useful function is "join". Try the following command,

```
", ".join(['a','b','c','d','eagle'])
```

The output will be

```
'a, b, c, d, eagle'
```

The elements of the list get "joined" with the ', '. The only really important thing to know is that elements of the list must be strings. If they are numbers, you'll need to call 'str(i)' with a list comprehension.

14 Thinking Code

When you start writing code, it's very easy to get locked into only using the computer. You need to not fall into this trap. It's often easier to work on paper first or while you're debugging. Explicitly walk through a loop on paper. Figure out what your outputs *should* be and see if they match your code.

Try to write what you want in human words. Python is designed to match the English language. Talk to the person sitting next to you, describe what you want your code to do to them.

If you are new to programming and intimidated by the language, ignore the language. Solve the problem first, then translate to code.

Below this line is the homework. ChatGPT can solve all these problems without issue. But that isn't really the point of *learning*. If this was your job, it's a slightly different story as your goal is to get things done, not learn how they work[8]. In any course, your goal is to learn. These problems are designed to help you learn. Don't cheat yourself of this opportunity[9].

As a bonus, I'm available to answer your questions and push you in the right direction. I will help you learn faster than if you were doing this on your own.

Problem 1 (5 pt) Write a function called `hello` that takes one input, a string, and returns the string "Hello input_1, my name is your_name". Here is what my function would output.

```
Input: hello("Morty")
Output: "Hello Morty, my name is Mitch."
```

Notice, the string is returned, not printed.

Problem 2 (5 pt) Write a function called `divisible` that returns a list of all numbers 0 to 10,000 that are divisible by the input number.

For example, if the input is 2, the output is $[0, 2, 4, ..., 10000]$.

I recommend using a single list comprehension. I also suggest you google the phrase "Python modulo operator" you'll see %. It'll be useful.

[8] Although you should always be learning.

[9] I won't be able to tell if it's you or ChatGPT, so it really is cheating yourself.

Problem 3 (5 pt) Write a function called `remove_none` with a single dictionary input and returns a dictionary with no values on None.

None is a special object in Python that represents when a function has no output. It's literally `None` in Python.

Here is an example of this function in action:

```
Input: remove_none({"a": 1, "b": "Mitch", "c":None})
Output: {"a":1, "b": "Mitch}
```

Problem 4 (5 pt) Write a function called `length` that finds the length of a list.

Do not use the built-in "len" function.

The one input is a list and the one output is a number.

Problem 5 (5 pt) Write a function called `my_reverse` that reverses a list.

Do not use the built-in "reverse" function.

The one input is a list and the one output is also a list.

Problem 6 (5 pt) Write a function that finds the minimal element of a list.

Do not use the built-in "min" function.

The function should be called `my_min`. The one input is a list and the output is a number.

Problem 7 (10 pt) Write a function called `written_numbers` that returns a dictionary whose keys are integers less than the input and values are strings.

The input is a number, for this problem we can assume the number is positive and less than 1000. So your function should work for any integer between and including 0 and 999.

For example, for an input of 5 you'll return {`0:"zero",1:"one",2:"two",3:"three",4:"four",5:"five"`}. There are ways to be clever about this, 25 and 26 aren't really different.

Problem 8 (10 pt) Write a function called `fizzbuzz`. There should be one input, a positive integer n and output a string. The function will iterate over all integers from 0 to $n-1$ if the integer is a multiple of 3, add fizz to the output, if it's a multiple of 5, add buzz and if it's a multiple of both 3 and 5, print fizzbuzz. Don't forget the newline character "`\n`" at the end of each line.

Here is some sample output if $n = 16$,

```
fizzbuzz
fizz
buzz
fizz
fizz
buzz
fizz
fizzbuzz
```