1
Python Tutorial!

This is just a lil thing I put together for my pals to learn some Python. We'll be going over the following:

- Basic Scripting
- Classes/OOP
- Working with Files
- Using PIP
- Numpy Stuff

Before getting into the programming, let's talk a bit about Python. Python is a swiss army knife programming language–it can be used for object-oriented, functional, or procedural code–, and the Python environment has access to countless modules made throughout the community. There are some modules available by default like `sys`, `os`, and `math`; others are available via `PIP`: the Python package manager. We will get into using `PIP` a bit later.

1.1 Why Python?

Python is a super popular language. It's easy to learn and write, and it's useful in a bunch of different areas, especially given the community behind it and the number of modules available! My guy–and yours–Guido van Rossum made Python, and it's designed to be very readable (more like natural language). With this is the formatting of the language: scope is based around whitespace–more on that later. Similar to Matlab, we can run basic commands in the terminal (REPL) or we can write and run files!

Python ultimately gets used for a bunch of different stuff; you can do maths, you can do web dev, you can do machine learning, you can do system work, you can do most anything with Python. That being said, there are certainly drawbacks to Python. That would *mostly* be speed; not that it's of huge concern–Python is used for tons of huge things in industry and it gets the job done just fine–, but there may be times you want to use Python to prototype a project and then translate to a lower level, faster language. For example, a basic program to find all prime numbers up to some natural number N written in Python vs C for N=1000000 took 187335 ms rather than 32284 ms (that's almost 6 times slower than a similar implementation in C, of course considering that these numbers are coming from my laptop). That being said, very frequently being slow is based off of poor programming rather than poor programming languages. We will also be recreating this program a bit later!

1.2 Do I have Python?

I'm going to assume that you are running macOS or some linux flavor–if you have Windows: don't have Windows! We are going to be using the terminal a fair bit, cause IDEs (Integrated Development Environments) are the devil–remember the good ol' Matlab application that you have to open up and use to write/run Matlab? Gross!! We will be writing Python in a text editor and running it in a terminal emulator. The "G" in GUI stands for "grody", but it's just spelled like "graphical"–the english language, am I right??

> There are a million text editors. If you want to do *everything* in your terminal, feel free to use: Vim, Emacs, or Nano (you should at least know how to use one of these at some point, I personally love Vim). If you're a normal person, you'll probably use a separate application such as: Atom, Sublime, Notepad++, Visual Studio Code, Gedit, or Brackets (I primarily use Visual Studio Code but I crack open Sublime every once in a while). As for a terminal, you have one that works just fine with macOS (called "Terminal") or any sort of Linux distro–maybe the new one for Windows isn't utter garbagio, but I wouldn't count on it.

Open up your terminal! It will have some prompt like this:

```
MyComputer:Path/To/Some/Directory$
```

Or some business like that; the prompt doesn't matter, and I will represent the command line with just a `$`. Go ahead and check whether you have Python like this:

```
$ python --version
```

You should get some reasonable output. I will be going through all of this tutorial stuff for Python 3, cause that's the new stuff. I suppose if your Python version gave some Python 2, you can explicitly check for 3 with:

```
$ python3 --version
```

And so long as that gives reasonable output we are happy and you don't have to install anything!

If you don't have Python 3, I'll strongly recommend you use homebrew which should let you `brew install python` for Python 3!! Also, if you want to be kind with versioning for your programs, you can specify the path to a Python interpreter in a shebang at the top of any of your scripts. It may look like this:

```
#!/usr/bin/env python3
...
print("Wow I'm glad there's a shebang at the top of this file")
...
```

1.3 Command Line Python

As I mentioned, we can run basic Python stuff just in the command line–like we can with Matlab. This is a Read-Eval-Print-Loop, or REPL. I'll go through some basic stuff and output with that real quick, but then we will go back to ignoring it. We initialize the Python shell with

```
$ python
```

or some other executable in your path like:

```
$ python3
```

And then we are greeted with some garbage, and can start writing Python!

```
$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Howdy folks!")
Howdy folks!
>>> 5+4*3
17
>>> 2**5
32
>>> exit()
```

If you don't want to close it with `exit()` you can also use `ctrl-d`.

1.4 Python from a file!

It's time to crack open that fancy new text editor and we will make a cute little script to make sure everything is all nice and cauchy. We will make a new file called `BasicScript.py`, the ".py" being the extension for Python files. This file will look like this:

```
'''
BasicScript.py
    This is just a basic file to print from a list
    By the way, this is in a multiline comment (triple apostrophes)
    We can write in a block here
'''
# This is a one line comment
print("Welcome to my script!")
# This is an array initialized with four strings
listOfWords = ["One", "Fish", "Two", "Fishes"]
# This loops over each element in the array
for word in listOfWords:
    print(word) # This prints each word
```

Let's save make a new folder called `PythonTutorial` on your desktop, and we will save `BasicScript.py` inside of it. Now we can go about running it from your terminal.

1.4.1 An aside on using your terminal

By default, when you open your terminal, it opens from your home directory ("directory" is computer science for "folder"). Let's go over some basic UNIX commands so that we can navigate and feel cool!

- `pwd`: prints current (working) directory
- `ls`: lists the contents of the current directory
- `cd`: changes the current directory
- `cp`: copies a file
- `mv`: moves a file
- `rm`: deletes a file
- `touch`: creates a file
- `mkdir`: creates a directory (folder)
- `cat`: outputs the contents of a file
- `man`: displays a manual page

These are, of course, not all necessary for using Python, mostly just `cd` matters. But we shall quickly go into using some of these. I will prepend the `$` prompt with an indication of the current directory. (Note the tilde is shorthand for your `/home/username` directory commonly referred to as the home directory)

Show you where you are

```
~:$ pwd
/home/cole
```

Show you whats on this folder

```
~:$ ls
Desktop Documents Downloads Music Pictures SomeFolder SomeFile.txt Etc.md
```

Move to the Desktop

```
~:$ cd Desktop
```

See whats on the Desktop

```
~/Desktop:$ ls
FolderOne FolderTwo PythonTutorial SomeOtherFolder SomeFile.png
```

Move to the PythonTutorial folder

```
~/Desktop:$ cd PythonTutorial
```

Run the script we made!

```
~/Desktop/PythonTutorial:$ python3 BasicScript.py
Welcome to my script!
One
Fish
Two
Fishes
```

Look at you go! You wrote and ran your first Python script!!!! Give yourself a frickin pat on the back. Now let's dig a little deeper...

1.5 Python Basics

In Python we have a bunch of different types available by default–we will look at making our own types a little bit later. We have the following:

- boolean
- integer

- float
- complex numbers
- string
- list
- dictionary
- None

And we shall take a quick look at them!

1.5.1 General Variables

In Python, like Matlab, there's nothing special about declaring variables, we just set some variable equal to what it is `VARNAME = VALUE`. This is called *duck typing* where "if it walks like a duck/string and it quacks like a duck/string, then it must be a duck/string."

Let's take a look at making some different variables in a new file called `VariablesTutorial.py`

```python
# boolean
a = True
b = False
# integer
c = 1
d = -23
# float
e = 1.23
d = -20.3
# complex numbers
f = 3+2j
g = -2.3+4.99j
# string
h = "Howdy"
i = 'there'
j = str(d)
k = "use + to concatenate " + str(f) + " a variable"
l = f"or 'f' and brackets to {c} interpolate a variable"
# list
m = [0, 1, 2.44, -3.3]
n = list(range(10))
o = [[1, 3, 2], ["a", "b", 3, 2+1j], list(range(3)), "woah", True]
# dictionary
p = {
    "name": "cole",
    "role": "author",
    "birthday": "3/21/98",
    "numToes": 10,
    "listOfStuff": [1, "a", 3+2j, False]
}
# None
q = None
#print all our variables separated by a newline (sep="\n")
print(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, sep="\n")
```

Now if we run it, we see rather expected output!

```
~/Desktop/PythonTutorial:$ python3 VariablesTutorial.py
True
False
1
-20.3
1.23
(3+2j)
(-2.3+4.99j)
Howdy
there
-20.3
use + to concatenate (3+2j) a variable
or 'f' and brackets to 1 interpolate a variable
[0, 1, 2.44, -3.3]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[[1, 3, 2], ['a', 'b', 3, (2+1j)], [0, 1, 2], 'woah', True]
{'name': 'cole', 'role': 'author', 'birthday': '3/21/98', 'numToes': 10, 'listOfStuff':
    [1, 'a', (3+2j), False]}
None
```

Now let's go a bit deeper with each of these:

1.5.2 Booleans

Booleans are just true/false values, they are `True` or `False` and can be checked in several ways!

```
# Or operator
a = True or False       # True!
# And operator
b = True and False      # False!
# Equality operator
c = True is False       # False!
d = True == False       # False!
e = True != False       # True!
# Negation operator
f = not True is False   # True!
# Exclusive or operator
g = not True ^ False    # False!
# Inequalities
h = 2 < 2               # False!
i = 1 <= 1              # True!
j = 3 > 2               # True!
k = 1 >= 2              # False!
# Set membership
l = 1 in [1, 2, 3]      # True!
m = 4 in [1, 2, 3]      # False!
n = 4 not in [1, 2, 3]  # True!
```

These get used within `conditional statements` to do things only when true! Such as the `if-statement`:

```
if 1 < 2:
    print("1 is less than 2")
else:
    print("1 is not less than 2")
```

or the `while-loop`:

```
i = 0
while i < 10:
    print(i)
    i+=1
```

1.5.3 Numbers

We have integers, floats, and complex numbers. They all work the same way so who cares, let's talk about different operators!

Basic maths

```
a = 1 + 1    # addition: 2
b = 1 - 1    # subtraction: 0
c = 2 * 2    # multiplication: 4
d = 2 ** 4   # exponentiation: 16
e = 4 / 2    # division: 2.0
f = 4 // 5   # floor division: 0
g = 5 % 3    # modulus: 2
h = 3 & 2    # bitwise and: 2
i = 7 | 15   # bitwise or: 15
j = 5 ^ 7    # bitwise xor: 2
k = ~3       # bitwise not: -4
l = 15<<1    # shift left: 30
m = 15>>1    # shift right: 7
```

These assign variables to the values of the operation!

Assignment operators

```
a = b = c = d = e = f = g = h = i = j = k = l = m = 10
a = 12  # reassignment: 12
b += 1   # addition: 11
c -= 1   # subtraction: 9
d *= 2   # multiplication: 20
e **= 3  # exponentiation: 1000
f /= 2   # division: 5.0
g //= 5  # floor division: 2
h %= 3   # modulus: 1
i &= 2   # bitwise and: 2
j |= 15  # bitwise or: 15
k ^= 7   # bitwise xor: 13
```

```
l <<=1   # shift left: 20
m >>=1   # shift right: 5
```

These change a variable based off of an expression. `b+=1` is equivalent to `b=b+1`

Note that we can compare numbers with the prior boolean operators.

1.5.4 Strings

Strings are just some grouping of characters! Rather, they are arrays of characters, and are just a special implementation of a list. As previously noted, we can use single or double quotes, but what if you want to use a double quote within a string made with double quotes? This requires an “escape character.” This is special syntax to signal that the following character is special (kinda like LaTeX tokens!), and we do that with the “’ character. We can do the following

```
$ python3
>>> print("The \"bird\" was \"not a conspiracy\"")
The "bird" was "not a conspiracy"
```

Let’s go over some useful String bits!

```
a = "This is a string"
b = a[0:7]          # Slice: "This is"
c = len(a)          # Length: 16
d = a[3]            # Index: "s"
e = a[-1]           # End index: "g"
f = a.split(" ")    # Splits on character: ['This', 'is', 'a', 'string']
g = "   this has whack whitespace    ".strip()  # Removes leading and trailing whitespace
```

1.5.5 Lists

Lists are the basis of strings! We still have `len()` we still have indexing and slicing, just not really `split()` or `strip()`. That’d be weird.

Let’s think on lists though:

```
a = [] # makes an empty list: []
b = [9, 8, 7] # makes a list with stuff in it: [9, 8, 7]
a.append(1) # add 1 to the list: [1]
a.extend([2, 3, 4]) # add multiple things: [1, 2, 3, 4]
a.append(b) # append a different list: [1, 2, 3, 4, [9, 8, 7]]
c = a.pop() # remove last element of list: a=[1, 2, 3, 4], c=[9, 8, 7]
b.sort()    # sorts list: [7, 8, 9]
a.sort(reverse=True)    # sorts in reverse: [4, 3, 2, 1]
d = a[3]   # gets index from start: 1
e = a[-2]  # gets index from end: 2
```

```
f = a[2:]  # gets slice from index to end: [2, 1]
g = a[:3] # gets slice from start to index: [4, 3, 2]
```

1.5.6 Dictionaries

Dictionaries are a groupof key-value pairs. We can similarly use any data types for keys and values with dictionaries.

We'll look at some basics:

(Note: I use pprint to print things in a more readable fashion, you can install this with `pip install pprint`)

```
import pprint
pp = pprint.PrettyPrinter(indent=3)
a = {} # makes an empty dictionary
pp.pprint(a)
b = {
    "Name": "Cole",
    "Birthday": "3/21/1998",
    "Age": 21,
    "Favorite foods": ["Popeyes", "Louisiana Fast", "Number 7", "Spicy Chicken"]
}
pp.pprint(b)
# Add items
print(len(b), len(a))   #number of key-value pairs: 4, 0
b["Favorite Kernel"] = "Linux"
b.update({"Python version": "3.6.7", "Donut preference": "Glazed"})
pp.pprint(b)
# Delete items
b.pop("Birthday")
del b["Age"]
pp.pprint(b)
b.clear()
pp.pprint(b)
```

1.5.7 None

None is real simple! It is simply something that doesn't exist. We tend to just check if things are None or not. This matches up with `null` of many other languages.

For instance:

```
a = None
if a is not None:
    print(a)
else:
    print("No object 'a' exists")
```

1.6 Practical Example!

I know you're itching to get into some stuff, so let's! We can go through a simple program to output all prime numbers up to a given integer. Let's save it in `Primes.py`

```python
# How high do we go?
n = 1000
# Begin with just 2
primes = [2]
# Loop from 2 to n
for i in range(2, n+1):
    # Assume i is prime
    isPrime = True
    # Search for a contradiction
    for p in primes:
        # if a prime is a divisor, n is not prime
        if (i/p).is_integer():
            isPrime = False
    # otherwise, no divisors exist! add to primes list
    if isPrime:
        primes.append(i)
print(primes)
```

If we go ahead and run this, we will see a list of all primes up to 1000! That's pretty boring though, let's get a little more advanced...

Let's try and abstract some of this into a method called `isPrime` that will tell us if an integer is prime or not.

```python
# How high do we go?
n = 1000
# Begin with just 2
primes = [2]
def isPrime(n):
    for p in primes:
        # if a prime is a divisor, n is not prime
        if (n/p).is_integer():
            return False
    # otherwise, no divisors exist!
    return True
for i in range(2, n+1):
    # Just check for if it is prime or not!
    if isPrime(i):
        primes.append(i)
print(primes)
```

Note that our method must be placed above our main for-loop that calls it! This is because Python is read top to bottom, and if we call `isPrime` before we define it, it's a bad look for us. Let's do something about that. It's generally good practice to write modular code, and we will abstract the main for-loop into a `getPrimes` method, and call it at the bottom.

```python
# Begin with just 2
primes = [2]
def isPrime(n):
    for p in primes:
        # if a prime is a divisor, n is not prime
        if (n/p).is_integer():
            return False
    # otherwise, no divisors exist!
    return True
def getPrimes(n):
    for i in range(2, n+1):
        # Just check for if it is prime or not!
        if isPrime(i):
            primes.append(i)
    print(primes)
if __name__ == "__main__":
    getPrimes(1000)
```

Note the bit with `if __name__ == ‘‘__main__’’:`, the variable `__name__` is automatically set to `‘‘__main__’’` if it is run by itself (kind of... more on these double underscore guys later). If our script is imported by some other module, then `__name__` is set to the name of that module. Using this if-statement allows us to check what is calling our program; somebody might want to use our `getPrimes` method, but doesn't necessarily want it to run for 1000. This condition limits our program to only run `getPrimes(1000)` when the program is run in a standalone fashion. This is quite important, if your TI-83 spat out a bunch of garbage every time you turned it on, you'd be pretty upset.

Now let's get some more tools under our belt. Something like our method `isPrime` might be something we don't much care about keeping around. Python features anonymous functions for that! If we don't want to go through the labour of defining short little functions we can use what is called a `lambda` function.

For a silly example, we may want a function to add two to a number and then square it. This is a quick little guy to write, and maybe we don't care enough to define this function in the usual fashion, and we would rather do it in one line. We can do the following to define and call it!

```python
addTwoAndSquare = lambda a : (a+2)**2
print(addTwoAndSquare(3))   #prints 25
```

Now let's apply this to our prime script. We can make a lambda function that will get all the prime divisors of each integer; then if the list is empty we have a prime number.

```
# Begin with just 2
primes = [2]
def getPrimesLambda(n):
    # Loop from 2 to n
    divisors = lambda a :  [p for p in primes if (i/p).is_integer()]
    for i in range(2, n+1):
        if not divisors(i):
            primes.append(i)
    print(primes)
if __name__ == "__main__":
    # How high do we go?
    getPrimesLambda(1000)
```

Let's say we are all giddy about our prime numbers, and we want to get the prime factorizations of a bunch of integers. We want to break down all the composite factors into products of primes, we have a bunch of primes, so it seems reasonable! Lets do that.

```
composites = {}
def primesComposites(n):
    # Loop from 2 to n
    for i in range(2, n+1):
        divisors = lambda a :  [[p,int(i/p)] for p in primes if (i/p).is_integer()]
        if not divisors(i):
            primes.append(i)
        else:
            composites[str(i)] = divisors(i)
```

First we will adjust our `getPrimesLambda` to make sure we are collecting composite numbers. Of course, if we just want a list of composite numbers, we can take the set difference, you should give that a whirl, by the way. In this, however, we want the composites and all of their prime divisors (and their quotients). We first adjust our lambda function to give us pairs of the prime divisor and the quotient of the integer and the prime divisor (6 should give us 2 and 3). Now, we may add these to some dictionary for composite numbers, the key will be the integer, the value will be a list of all pairs of prime divisors and their quotients. Next we will compute the prime factorization of an integer!

```
def primeFact(n):
    primeFactorization = []
    if n in primes or n == 1:
        primeFactorization.append(n)
    else:
        c = composites[str(n)]
        firstFactor = c[0][0]
        secondFactor = c[0][1]
        primeFactorization.append(firstFactor)
        primeFactorization.extend(primeFact(secondFactor))
    return primeFactorization
```

```
if __name__ == "__main__":
    primesComposites(100)
    numList = [1, 6, 12, 60, 75, 80, 99, 100]
    for num in numList:
        print(num, ":", primeFact(num))
```

This is a recursive method–a method that calls itself–that will return a list representing the prime factorization of `n`. If `n` is prime (or 1), the prime factorization is just itself. Otherwise, we can hash into the composites dictionary and get the lowest factor (the first entry). This takes the form of `[prime, prime or composite]` and, we are writing a function to return the prime factorization of a prime or composite number! We ought to add the prime in the first index, and then we can recurse on the second index that may be prime or composite! It will continue to call `primeFact` until we reach the base case of `n` being prime. Then, once we have that, we return it.

Perhaps we collected all primes up to some large number, and it took *forever*. We don't want to wait that long ever again, what do we do? How can we hold on to our computer's hard work. Let's save it! `JSON` stands for JavaScript Object Notation, and it is a form of a serializable object: we can encode it and decode it! It resembles a Python dictionary. Let's say we want to write our hard work to some json file via a method `writePrimes`. First we want to think on how we would like to store it; I think it'd be wise to keep our primes separate from our composites, so we will put them in some dictionary called `data`. Then, we must open a file to write to (don't worry, it'll create it if it doesn't exist). We can do this with a block, where we open the file, note that we are writing to it with `‘‘w’’` and calling it something like `f`. Then we just call `json.dump()` and give it our new `data` dictionary and the file reference. That's all!!

```
import json
def writePrimes():
    data = {
        "primes": primes,
        "composites": composites
    }
    with open("primes.json", "w") as f:
        json.dump(data, f)
```

Of course, just writing isn't Cauchy, we want to be able to retrieve this information. This is a similar process, however we are storing the retrieved data into our variables. We will be setting `primes` and `composites` to the contents of the file, and thus have to note the scope of them by declaring that we are using the global variables. Then after reading from the file with `json.load(f)` we may parse the result in the usual fasion of accessing a Python dictionary.

```python
def readPrimes():
    global primes
    global composites
    with open("primes.json", "r") as f:
        data = json.load(f)
    primes = data["primes"]
    composites = data["composites"]
```

We did it! Now we can do something like this to store all the primes and composites up to 100000:

```python
if __name__ == "__main__":
    primesComposites(100000)
    writePrimes()
```

And then this to load them and look at some factorizations:

```python
if __name__ == "__main__":
    for _ in range(0, 10):
        num = random.randint(1, max(primes))
        print(num, ":", primeFact(num))
```

Note: the _ in the for-loop is a dummy variable and is not used! This underscore tells Python that we don't care about it. It stores the last value from an expression–like `ans` on a calculator.

Ok, we have done a LOT here. Our main has changed a bunch, and that's probably not so good. Let's abstract some more, and we will change what we do based off of user input!!! Time for... processing the command-line!

First, let's think about what functionality we want at the end of this. I'm thinking we should be able to

- Say if a passed number is prime
- Give an input/output file
- Print out all primes in the file or up to a number
- Print prime factorization of a passed number

- Print prime factorizations of a tuple
- Print a help message

So we should set flags for those! I'll note those through the output of our help flag:

```
To run:
        primes.py (options) NUMBER
Options:
        -h || --help                        Prints this help message
        -v || --verbose                     Gives additional output
        -r=<FILE> || --read=<FILE>          Reads in primes list from specified file (
            JSON)
        -w=<FILE> || --write=<FILE>         Writes primes list to specified file (JSON)
        -c || --composite                   Collects composite numbers also
        -f || --factorization               Prints prime factorization of NUMBER
        -b=num,num... || --batch=num,num... Prints prime factorization of num,num,num
            ...
```

Alright, now that we've thought on what we want to be able to do, let's do it! And I'm sure you've guessed it, there'll just be a lot of conditionals and checking for these flags. We'll start our main by making a bunch of variables corresponding to the presence of the flags and the attached bits if applicable:

```python
if __name__ == "__main__":
    verbose = False # print more stuff!
    write = False   # write to file
    read = False    # read from file
    factorization = False   # print prime factorization
    getComposites = False   # collect composite numbers
    readFile = None     # filename to read from
    writeFile = None    # filename to write to
    batchFactorizationList = None   # tuple to factor
    maxOfBatch = 2 # max int of batch
    numPrimes = None # number passed
```

Now we are going to add some helper methods, they'll be a lil tricky, don't worry, I'll explain:

```python
# checkArgs: is the argument present
checkArgs = lambda toCheck: any([any(arg.startswith(s) for arg in sys.argv) for s in
    toCheck])
# removeArgs: sublist without the argument
removeArgs = lambda toRemove: [arg for arg in sys.argv if not any([arg.startswith(s) for
    s in toRemove])]
# getString: extract string with arg
getString = lambda toRemove: str(set(sys.argv).difference(removeArgs(toRemove)).pop()).
    split('=', 1)[1]
```

Ok SO: `checkArgs` takes in a list of strings, and returns true if any of the command line arguments start with any of the strings from the list. `removeArgs` works similarly, but it reconstructs the arguments passed while filtering out any arguments beginning with strings from a list. `getString` is for reading, writing, and batch factorization, where we want to get the argument that we are going to remove in `removeArgs`. I'll break them down into a bit more detail, in case it's confusing:

`checkArgs`: `any()` returns true if any of the parameters are evaluate to true. So we have an outer `any` that takes in a list made from another `any` within list comprehension. Moreover, we look at every string `s` in `toCheck`, and if any arguments in `sys.argv` begin with a member of `toCheck` then we evaluate to true!

`removeArgs`: Similarly, we will use list comprehension and `any` to make our result. We want each of the arguments from `sys.argv` if they don't begin with any string from our blacklist, `toRemove`.

`getString`: This is used because we may pass in something like `--read=primeslist.json` and we want to be able to extract `primeslist.json` or whatnot. So, we can get the argument from the set difference of `sys.argv` and the output of `toRemove` with a passed filter. We then pop the set to remove the element, we cast it as a string, and we split the string on '='. This gives us a list of two elements, the argument up to the equals sign, and the part after it, we can get the latter by accessing the array from `split()` at index 1. Just in case we have some file name with an equals sign, we set the maximum splits to be 1 as well.

Hopefully that's a decent enough explanation, now let's get into using them! Here we have the getting of options:

```
if checkArgs(["-h", "--help"]) or len(sys.argv) is 1:
    print("To run:\n\tprimes.py (options) NUMBER")
    print("""Options:
    -h || --help                          Prints this help message
    -v || --verbose                       Gives additional output
    -r=<FILE> || --read=<FILE>            Reads in primes list from specified file (JSON)
    -w=<FILE> || --write=<FILE>           Writes primes list to specified file (JSON)
    -c || --composite                     Collects composite numbers also
    -f || --factorization                 Prints prime factorization of NUMBER
    -b=num,num... || --batch=num,num...   Prints prime factorization of num,num,num..."""
        )
    sys.exit()
if checkArgs(["-v", "--verbose"]):
    sys.argv = removeArgs(["-v", "--verbose"])
    verbose = True
if checkArgs(["-r", "--read"]):
    readFile = getString(["-r", "--read"])
    sys.argv = removeArgs(["-r", "--read"])
    readPrimes(readFile)
    read = True
    if verbose:
        print("Read file:", readFile, "max num:", primes[-1])
if checkArgs(["-w", "--write"]):
    writeFile = getString(["-w", "--write"])
    sys.argv = removeArgs(["-w", "--write"])
    if verbose:
        print("Write file:", writeFile)
if checkArgs(["-f", "--factorization"]):
    sys.argv = removeArgs(["-f", "--factorization"])
    factorization = True
if checkArgs(["-c", "--composite"]):
    sys.argv = removeArgs(["-c", "--composite"])
    getComposites = True
if checkArgs(["-b", "--batch"]):
    batchFactorizationList = getString(["-b", "--batch"])
    batchFactorizationList = [int(n) for n in batchFactorizationList.split(",")]
    maxOfBatch = max(batchFactorizationList)
    sys.argv = removeArgs(["-b", "--batch"])
```

This is rather formulaic: if the argument exists, process and remove it! Some small notes though: we want to print the help output if the flags are passed or if there aren't enough arguments to do anything else, in reading we want to show the biggest prime in the file if we are verbose, and for batch prime factorization we make an array of ints by splitting on commas and casting to ints and then we make note of the maximum. Now, let's do the rest!

```
if len(sys.argv) > 1:
    numPrimes = int(sys.argv[1])
    if getComposites or factorization or batchFactorizationList:
        primesComposites(max(maxOfBatch,numPrimes))
    else:
        getPrimesLambda(numPrimes)
    if verbose:
        print([prime for prime in primes if prime <= numPrimes])
    if factorization:
        print(numPrimes, ":", primeFact(numPrimes))
elif verbose and read:
    print(primes)
if batchFactorizationList:
    if not numPrimes:
        primesComposites(maxOfBatch)
    print("Factorization Batch:")
    for num in batchFactorizationList:
        print("-->  ",num, ":", primeFact(num))
if numPrimes:
    print(numPrimes, "is prime" if isPrime(numPrimes) else "is not prime")
if writeFile is not None:
    writePrimes(writeFile)
```

We start by getting the number if it is present (it'll be index 1 since we remove all options while processing them!). Then, we choose the method for getting primes: we collect composites if we pass the flag or if we have to do prime factorization, otherwise we proceed with just primes. If we are verbose, print out all primes up to the number passed. If we care about prime factorization, print that out too! If we did not pass in a number, but we are reading from a file, we will print out all the primes in the file. Then, we deal with batch factorization: if we didn't pass a number then we have to collect the primes/composites here instead! Then we want to end by printing out whether the passed number is prime or not, and then writing to the file if applicable.

Cool!! We finished our first project!!! I'll put in the completed file right here, followed by example calls to it:

```python
#!/usr/bin/python3
import sys
import json
import random
primes = [2]
composites = {}
def primesComposites(n):
    # Loop from 2 to n (range is not inclusive)
    for i in range(2, n+1):
        divisors = lambda a :  [[p,int(i/p)] for p in primes if (i/p).is_integer()]
        # add to primes if no divisors, otherwise add to composites
        if not divisors(i):
            primes.append(i)
        else:
            composites[str(i)] = divisors(i)
def primeFact(n):
    primeFactorization = [] # start with empty list
    if isPrime(n) or n == 1:
        # add to list if prime or 1
        primeFactorization.append(n)
    else:
        # otherwise, recurse on the composite num
        c = composites[str(n)]
        firstFactor = c[0][0]
        secondFactor = c[0][1]
        primeFactorization.append(firstFactor)
        # add prime factorization of the composite num
        primeFactorization.extend(primeFact(secondFactor))
    return primeFactorization
def isPrime(n):
    for p in primes:
        if p == n: # if we called using a prime
            return True
        if p > n: # if no divisors up to n
            return False
        if (n/p).is_integer(): # if a prime is a divisor, n is not prime
            return False
    return True # otherwise, no divisors exist!
def getPrimesLambda(n):
    # Loop from 2 to n, add prime if it has no divisors
    divisors = lambda a : [p for p in primes if (i/p).is_integer()]
    for i in range(2, n+1):
        if not divisors(i):
            primes.append(i)
def readPrimes(filename):
    # use the global scoped vars
    global primes
    global composites
    with open(filename, "r") as f:
        data = json.load(f)
        primes = data["primes"]
        composites = data["composites"]
def writePrimes(filename):
    # construct data dict to write
    data = {
        "primes": primes,
        "composites": composites
    }
    with open(filename, "w") as f:
        json.dump(data, f)
if __name__ == "__main__":
    verbose = False # print more stuff!
    write = False   # write to file
    read = False    # read from file
    factorization = False   # print prime factorization
    getComposites = False   # collect composite numbers
    readFile = None     # filename to read from
    writeFile = None    # filename to write to
    batchFactorizationList = None   # tuple to factor
    maxOfBatch = 2 # max int of batch
    numPrimes = None # number passed
    # checkArgs: is the argument present
    checkArgs = lambda toCheck: any([any(arg.startswith(s) for arg in sys.argv) for s in
        toCheck])
    # removeArgs: sublist without the argument
    removeArgs = lambda toRemove: [arg for arg in sys.argv if not any([arg.startswith(s)
        for s in toRemove])]
    # getString: extract string with arg
    getString = lambda toRemove: str(set(sys.argv).difference(removeArgs(toRemove)).pop()
        ).split('=', 1)[1]
    if checkArgs(["-h", "--help"]) or len(sys.argv) is 1:
        print("To run:\n\tprimes.py (options) NUMBER")
        print("""Options:
        -h || --help                        Prints this help message
        -v || --verbose                     Gives additional output
        -r=<FILE> || --read=<FILE>          Reads in primes list from specified file (
            JSON)
        -w=<FILE> || --write=<FILE>         Writes primes list to specified file (JSON)
        -c || --composite                   Collects composite numbers also
        -f || --factorization               Prints prime factorization of NUMBER
        -b=num,num... || --batch=num,num... Prints prime factorization of num,num,num
            ...""")
        sys.exit()
    if checkArgs(["-v", "--verbose"]):
        sys.argv = removeArgs(["-v", "--verbose"])
        verbose = True
    if checkArgs(["-r", "--read"]):
        # get file attached to arg
        readFile = getString(["-r", "--read"])
        sys.argv = removeArgs(["-r", "--read"])
        # read in file, print max prime if verbose
        readPrimes(readFile)
        read = True
        if verbose:
            print("Read file:", readFile, "max num:", primes[-1])
    if checkArgs(["-w", "--write"]):
        # get file attached to arg
        writeFile = getString(["-w", "--write"])
        sys.argv = removeArgs(["-w", "--write"])
        if verbose:
            print("Write file:", writeFile)
    if checkArgs(["-f", "--factorization"]):
        sys.argv = removeArgs(["-f", "--factorization"])
        factorization = True
    if checkArgs(["-c", "--composite"]):
        sys.argv = removeArgs(["-c", "--composite"])
        getComposites = True
    if checkArgs(["-b", "--batch"]):
        # get tuple attached to arg, put into int list, save max
        batchFactorizationList = getString(["-b", "--batch"])
        batchFactorizationList = [int(n) for n in batchFactorizationList.split(",")]
        maxOfBatch = max(batchFactorizationList)
        sys.argv = removeArgs(["-b", "--batch"])
    if len(sys.argv) > 1:
        # get number passed in
        numPrimes = int(sys.argv[1])
        # collect composites if necessary
        if getComposites or factorization or batchFactorizationList:
            primesComposites(max(maxOfBatch,numPrimes))
        else:
            getPrimesLambda(numPrimes)
        # print all primes up to number passed if verbose
        if verbose:
            print([prime for prime in primes if prime <= numPrimes])
        # print factorization
        if factorization:
            print(numPrimes, ":", primeFact(numPrimes))
    # if no number passed, print all primes of file
    elif verbose and read:
        print(primes)
    if batchFactorizationList:
        # if no number passed, get composites to max
        if not numPrimes:
            primesComposites(maxOfBatch)
        print("Factorization Batch:")
        for num in batchFactorizationList:
            print("-->  ",num, ":", primeFact(num))
    # print whether num passed is prime or not
    if numPrimes:
        print(numPrimes, "is prime" if isPrime(numPrimes) else "is not prime")
    # save to file
    if writeFile is not None:
        writePrimes(writeFile)
```

Note that the order of the arguments doesn't matter, that's the point of removing the arguments– also note that that's not necessarily best practice, we can always expect users to be able to adhere to a certain order.

```
python Primes.py 100
python Primes.py 100 -v
python Primes.py 100 --verbose -c -w=primes.json
python Primes.py -b=13,14,15,16 -v --read=primes.json 60 -f
```

Ok, so now that we are kickin it with Python as a scripting tool, let's get into some object oriented programming in our next section...

1.7 Object Oriented Programming

Ok, I know, I'm no Kerri-Ann Norton, no Bob McGrail, no Keith O'Hara, no Sven Anderson, but I've tutored for at least 3/4 of them!!! This won't be a super in depth OOP section, but I want to get some of the basics across. This will include:

- Attributes and methods
- Getters and setters
- Encapsulation
- Inheritance
- Polymorphism
- Operator overloading/magic methods

What is OOP? Well, it's just a style of programming where we create things that have information and can do stuff. The "things" are **objects**, the "information" an object has is **attributes**, and the "stuff" it can do is **methods**. More specifically, **objects** are *instances* of **classes**. Classes are like recipes/models for the thing that we are representing, and there is an important distinction between the class and the object. An object is an *instance* of a class (I know I said it earlier, but this is important!!). There may be attributes and methods at either the class or the instance level/scope. We have **class variables** and **class methods** which are defined only once and are at the class level. And, we have **instance variables** and **instance methods** which are at the object level, thus each object will hold a unique reference to them. Moreover, class methods can only access class variables, and instance methods can access both instance and class variables.

We define a class with the `class` keyword followed by the (capitalized by convention) name. The constructor takes the form `def __init__(self,[param1,param2...]):` where `self` is the only required parameter (a reference to the instance). Similarly all instance methods will take the form `def someMethod(self,[other,params]):`, and instance variables take the form `self.variableName`. For class methods and variables, we simply scrap the `self` and put them in the class scope.

Let's start by asking, "Why might I care about OOP?". Well, you *might*. There's certainly an anti-OOP wave at the moment, so let's talk about that. In favor of OOP, we often have reusability in a useful fashion, abstract data types (ADTs) can be tremendously useful fot... abstracting data, modeling objects, code structure, and of course encapsulation/inheritence/polymorphism. Against OOP is less so against OOP and more so against OOP as an end-all-be-all programming paradigm: creating and carrying around objects can be expensive and slow, things not using object-oriented niceties need not follow the object-oriented approach, making a model for everything isn't necessary.

Hopefully that didn't spook ya! OOP is certainly nifty and a useful tool to have under your belt, so let's get crackin. We're gonna hit a bunch of bases and make a bunch of different stuff. We'll start with something super simple: triangles! (Note, if you don't have `matplotlib` installed, install it with `pip install matplotlib`)

1.7.1 Basic Class Example: Triangle

```python
import math
# These are needed to draw
import matplotlib.pyplot as plt
import matplotlib.lines as lines
class Triangle:
    greeting = "Hi!! I'm a triangle!"
    dist = lambda a,b : math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)
    def getAngle(a, b, c):
        taninv = lambda a,b : math.atan2(b[1]-a[1], b[0]-a[0])
        theta = math.degrees(taninv(b, c) - taninv(b, a))
        return theta + 360 if theta < 0 else theta
    def __init__(self, pointA, pointB, pointC):
        print(Triangle.greeting, self.greeting) # instances have direct access to class
            variables
        self.pointA = pointA
        self.pointB = pointB
        self.pointC = pointC
        self.angleA = Triangle.getAngle(self.pointC, self.pointA, self.pointB)
        self.angleB = Triangle.getAngle(self.pointA, self.pointB, self.pointC)
        self.angleC = Triangle.getAngle(self.pointB, self.pointC, self.pointA)
        self.sideA = Triangle.dist(self.pointB, self.pointC)
        self.sideB = Triangle.dist(self.pointA, self.pointC)
        self.sideC = Triangle.dist(self.pointA, self.pointB)
    def printAngles(self, rad=False):
        theta = lambda a : str(a) + " deg" if not rad else str(math.radians(a)) + " rad"
        print("Angle A:", theta(self.angleA))
        print("Angle B:", theta(self.angleB))
        print("Angle C:", theta(self.angleC))
    def printSides(self):
        print("Side A:",self.sideA)
        print("Side B:",self.sideB)
        print("Side C:",self.sideC)
    def area(self, precision=5):
        s = (self.sideA + self.sideB + self.sideC)/2
        result = math.sqrt(s*(s-self.sideA)*(s-self.sideB)*(s-self.sideC))
        result = round(result, precision)
        return result if result != round(result) else round(result)
    def perimeter(self):
        return self.sideA + self.sideB + self.sideC
    def show(self):
        # x and y points
        x = [self.pointA[0], self.pointB[0], self.pointC[0], self.pointA[0]]
        y = [self.pointA[1], self.pointB[1], self.pointC[1], self.pointA[1]]
        plt.plot(x, y) # plot the points
        ax=plt.gca() # get the axes (get current axes)
        ax.set_xlim(min(x)-1, max(x)+1) # set bounds based on max/min
        ax.set_ylim(min(y)-1, max(y)+1)
        plt.title("Triangle") # title of graph
        plt.show()  # show the plot!
if __name__ == "__main__":
    tri = Triangle([0,0], [0, 2], [2,2])
    print("Perimeter:",tri.perimeter())
    print("Area:",tri.area())
    print("Tri Side A:", tri.sideA)
    tri.printAngles()
    tri.printSides()
    tri.show()
```

Ok, that was a lot... let's talk about it. We have the following:

Class attributes:

- `greeting`: Just a string to say howdy
- `dist`: A lambda function stored in a variable that gets the distance between two points (points being lists of length 2)

Class methods:

- `getAngle`: Gets the angle from three passed points

Instance variables/attributes:

- `pointA`: first point
- `pointB`: second point

- `pointC`: third point
- `angleA`: angle at pointA
- `angleB`: angle at pointB
- `angleC`: angle at pointC
- `sideA`: side across angleA
- `sideB`: side across angleB
- `sideC`: side across angleC

Instance methods:

- `__init__`: takes in three points and returns an instance of `Triangle`
- `printAngles`: helper method to print all angles (defaults to degrees)
- `printSides`: helper method to print all sides
- `area`: computes area using Heron's formula (defaults to 5 decimal point rounding)
- `perimeter`: computes the perimeter by summing side length
- `show`: displays the triangle using matplotlib

Most of this business is fairly simple, and we have seen how to do most of it, so I'll make a few notes and then talk a bit about matplotlib. We access class attributes and methods with `CLASSNAME.attribue` and `CLASSNAME.method()`, and we access instance attributes and methods with `self.attribute` and `self.method()`. Class attributes may be referenced within an instance with `self.attribute`. Parameters may be given default values, such as our `printAngles(self, rad=False)` where we default to not use radians, but we may override this when calling the method. The instance is what is returned by the constructor, and the constructor `def __init__(self):` is called with the classname `tri = Triangle([0,0], [0, 2], [2,2])`–here `tri` is the instance.

I'm no matplotlib expert, and I recommend looking more into it on your own, but I'll walk you through what's happening here. We define all of our points in two lists: one for the x-coordinates and one for the y-coordinates. Then, we add them to the pyplot with `plt.plot`. We set the axes to be plus and minus the max and min of each axis so that lines can't occur on the axes (remove the +/- 1 and keep the same triangle, it should be hide the lines on the axes). Changing bounds on the axes isn't totally necessary, but matplotlib depends on your system theme and those are never to be trusted–this lets us play it safe. Then we just give it an unnecessary title and call `plt.show()` to display the plot.

1.7.2 Encapsulation / Getters example: Banks

One of those things that the object-oriented folks rave about is encapsulation, but just what is that? Encapsulation is just some way to limit the accessibility of attributes and methods within a class. There are three access levels that we have got: public, protected, private. Public, the default in Python `thisVar = ‘‘public’’`, allows for arbitrary access. Protected, denoted by a leading underscore `_thisVar = ‘‘protected’’`, gives access within the class and subclasses. Private, denoted by *two* leading underscores `__thisVar = ‘‘private’’`, gives access only to the class. That being said, these are mostly just convention, and Python doesn't support true private variables, and there can be workarounds to access and modify "private" variables; I'll put in an example of that a bit later. I'll give the classic money spiel though:

Suppose you just got picked up by Big Bank Hank for a freelance programming job. You are to make a banking program (vague, I know, Henry Lee Jackson is new to the banking world in this example), and it had better be genius. Imp the Dimp isn't playing around, he's got bodyguards and two big cars... that definitely ain't the whack! So, now you're feeling the pressure. It's building up, and you're on the grind trying to get things right, but it's wearing on you... this is what you come up with:

```
import random
class Person:
    numActions = 10
    def __init__(self, name, pocketMoney=20):
        self.name = name
        self.salary = random.randint(100,500)
        self.pocketMoney = pocketMoney
        self.bankAccount = BankAccount(1250)
        self.actions = 0
        print("Hi! I'm", self.name)
    def work(self):
        print(self.name, "went to work")
        if random.random() > .9:
            print("WOO!! I got a raise!!!")
            self.salary *= 1.5
        self.pocketMoney += self.salary
        self.actions+=1
    def gamble(self):
        print(self.name, "hit the tables")
        r = random.random()
        if r > .999:
            print("We won big, baby!")
            self.pocketMoney += 1000000
        elif r < .01:
            self.pocketMoney *= -1
        else:
            self.pocketMoney -= self.salary/2
        self.actions+=1
    def deposit(self, amount):
        self.bankAccount.deposit(amount)
        self.pocketMoney -= amount
    def withdraw(self, amount):
        self.bankAccount.withdraw(amount)
        self.pocketMoney += amount
    def printPerson(self):
        print("Name:", self.name)
        print("Salary:", self.salary)
        print("Pocket Money:", self.pocketMoney)
        print("Actions left:", str(Person.numActions - self.actions))
        self.bankAccount.printAccount()
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount
    def getBalance(self):
        return self.balance
    def printAccount(self):
        print("Bank balance:", self.balance)
if __name__ == "__main__":
    print("Welcome to Hank's bank sim!")
    print("Work with 'j'")
    print("Gamble with 'g'")
    print("Print stats with 'p'")
    print("Deposit with 'd'")
    print("Withdraw with 'w'\n")
    hank = Person("Hank")
    while hank.actions < Person.numActions:
        hank.printPerson()
        instruction = input("\nWhat to do today?\n >")
        processed = instruction.strip().lower()
        if processed == "j":
            hank.work()
        elif processed == "g":
            hank.gamble()
        elif processed == "p":
            hank.printPerson()
        elif processed == "d":
            print("Max deposit:", hank.pocketMoney)
            amount = int(input("Deposit how much?\n >").strip().lower())
            if amount > hank.pocketMoney:
                amount = hank.pocketMoney
            hank.deposit(amount)
        elif processed == "w":
            print("Max withdraw:", hank.bankAccount.balance)
            amount = int(input("Withdraw how much?\n >").strip().lower())
            if amount > hank.bankAccount.balance:
                amount = hank.bankAccount.balance
            hank.withdraw(amount)
        else:
            try:
                # This works! "hank.bankAccount.deposit(100000)"
                eval(instruction)
            except Exception as e:
                print("Nice try, tough guy...")
```

Ok, yeah, if you've ever been in a CS class, you've heard this a thousand times... make your instance variables private! We don't want the Casanova Fly to be able to set his balance to whatever he wants without it coming from an actual deposit. So, let's toss in our dunderscores (double underscores) and our getters and fix it. I'll leave it as an exercise for you to patch up the rest, but I'll help get the ball rolling:

```
def __init__(self, name, pocketMoney=20):
    self.__name = name
    self.__salary = random.randint(100,500)
    self.__pocketMoney = pocketMoney
    self.__bankAccount = BankAccount(1250)
    self.__actions = 0
    print("Hi! I'm", self.__name)
def getName(self):
    return self.__name
def getActions(self):
    return self.__actions
def getBalance(self):
    return self.__bankAccount.getBalance()
def getPocketMoney(self):
    return self.__pocketMoney
...
elif processed == "d":
    print("Max deposit:", hank.getPocketMoney())
    amount = int(input("Deposit how much?\n >").strip().lower())
    if amount > hank.getPocketMoney():
        amount = hank.getPocketMoney()
    hank.deposit(amount)
```

But, just as we noted earlier, Python doesn't have true private variables, and Big Bank Hank can almost just as easily fraudulently deposit like this: ``hank._Person__bankAccount.deposit(100000)''. So, was that exercise useless? Of course not! We learned from it, now we know about accessibility modifiers, and we know we can use getters to access private variables in a specified way (i.e. `Person.getBalance()`). We also got to see nested classes with the `Person` having a `BankAccount` variable. We processed user input: `input()` waits for a user inputted string. We saw `eval()` which executes a command passed as a string. We saw `try-except` blocks–more on that in one moment. And most of all, we got some practice with classes.

Let's talk about the `try-except` block real quick though. Often we are running some program, and we run into some sort of error. This may be our fault (often), or it might be something more-or-less unrelated to us (less often). Either way, it'd sure be nice to be able to check for errors and then deal with them on our own accord. For instance, we may be writing some `Dog` class, and we may want to raise an exception (throw an error) if someone calls the `Dog()` constructor leading to `height=''10ft''`. I guess God didn't intend for 10 foot dogs, but the lord did intend for a 10 foot dog to not ruin my program with runtime errors! We can use a `try-except` block to try and create a 10ft Dog, and then if it doesn't work, we can do something else. Here we go:

```
class Dog:
    def __init__(self, height):
        if height >= 10:
            raise ValueError("\"Dog's can't be that tall, you fool\"")
        else:
            self.height = height
try:
  clifford = Dog(10)
except Exception as e:
    print(e)
    print("What a shame")
```

Well, clearly I've had a lot of fun writing this section. But, hopefully we can see the use of encapsulation. Sure, there are funky things to get around it sometimes, but not having true private variables can also be a thing of beauty: what if you are using someone elses module, and there's an issue related to a private field? You can actually deal with it! Python relies on a wonderful community that will adheres to these practices less so for security, and more so for readability: we can note intent via access level. Exception handling is also very important (not just for OOP!!). This is a tiny, contrived example. But this should give a nice gateway into the ways of `try-except`! We're going to use a bit of it in our next example. Clocks!

1.7.3 Encapsulation / Getters and Setters example: Clock

Encapsulation is important, and hopefully we can see why from our bank example. But, if we aren't supposed to access private/protected instance variables directly, how should we access them? We touched on it with the previous example after adding our dunderscores: getters and setters! Getters and setters are methods used to interface with private/protected fields, and they allow us to restrict usage. For instance, we had the `deposit` and `withdraw` methods in the `Person` class that equally affected `Person.__pocketMoney` and `Person.__bankAccount.__balance`. We can, of course make this more rigorous within the method, and only allow the `Person` to deposit as much pocket money as they have, and only withdraw as much is in their account. But, I don't care for dealing with money, and diversity of examples goes a long way. So let's look at some getters and setters for a `Clock` class. We're also going to look at using `property()` to create properties of a class.

Let's think about what we want to do for our `Clock` class. We just want to have an hour and a minute. We want to default to military time and store our hour as a number from 0-23 and our minute from 0-59. We'll have one method return a string for military time and one that returns a string converted to AM/PM. Now let's think on the getters and setters for hour and minute. The getters should be straightforward: we'll just return the value. The setters should only allow for valid arguments (0-23 for hours and 0-59 for minutes); we will raise an exception if we try to set times out of bounds. But, we may want the functionality of *adding* time: so we will make `addMinute` and `addHour` methods that jump forward or backward in time. Alright, now we know what we need to do, so let's get cracking!

```python
class Clock:
    def __init__(self, hour=0, minute=0):
        self._hour = hour
        self._minute = minute
    def time(self):
        hour = self._hour
        minute = self._minute
        if minute < 10: minute = "0"+str(minute)
        else: minute = str(minute)
        return f"{hour}:{minute}"
    def toAmPm(self):
        hour = self._hour
        minute = self._minute
        isAM = hour in range(0,12)
        if minute < 10: minute = "0"+str(minute)
        else: minute = str(minute)
        if hour == 0: hour = 12
        elif hour > 12: hour -= 12
        return f"{hour}:{minute} " + ("AM" if isAM else "PM")
    def getMinute(self):
        return self._minute
    def setMinute(self, value):
        if value < 0 or value > 59:
            raise ValueError("Minute out of bounds")
        else: self._minute = value
    minute = property(getMinute,setMinute)
    def addMinute(self, value):
        if value < 0:
            sumMin = self._minute +value
            hourDiff = -1*(sumMin // 60)
            minRem = sumMin % 60
            self.hour = self.hour - hourDiff
            self._minute = minRem
        else:
            sumMin = self._minute+value
            hourDiff = sumMin // 60
            minRem = sumMin % 60
            self.hour = self.hour + hourDiff
            self._minute = minRem
    @property
    def hour(self):
        return self._hour
    @hour.setter
    def hour(self, value):
        if value < 0 or value > 23:
            raise ValueError("Hour out of bounds")
        else: self._hour = value
    def addHour(self, value):
        self._hour = (self._hour + value) % 24
if __name__ == "__main__":
    clock = Clock(19,1)
    print(clock.toAmPm(), clock.time())
    # our getters
    print("Get hour:", clock.hour, "Get minute:", clock.minute)
    # our setters
    clock.hour = 5
    clock.minute = 25
    print(clock.time())
    clock.addMinute(123)
    print(clock.time())
    clock.addMinute(-63)
    print(clock.time())
    try:
        clock.hour = 30
    except ValueError as err:
        print("Error!",err)
```

Again, this should be a rather straightforward class exercise. We know how clocks work–yes, we even know military time. I'll instead focus on what's new. We have `getMinute` and `setMinute` where we are either returning or manipulating the value of `_minute`. In `setMinute`, if the new value is out of bounds, we raise a value error, so that the user will know what went wrong, otherwise we simply change the value. We do the same for our `_hour` attribute. The more funky part is with `minute = property(getMinute,setMinute)`. What this does is makes `minute` (as opposed to `_minute`) a property, and it has the value returned by `getMinute` and modifying it (via `=`) will go through `setMinute` with the value that we are changing it to. This allows us to use `minute` just as we would any other public instance variable, but it will adhere to the constraints of `getMinute` and `setMinute`. We do a similar thing for `hour` but we are using the shorthand property decorator. We place `@property` above our `hour()` method to signify that `hour` is to be treated like a public instance variable and `hour()` is the getter: we note the setter with a method `hour(value)` with the decorator `@hour.setter` above it. We use these getters and setters in both our main (commented above), and in `addMinute` where the hour is being both get and set within `self.hour = self.hour +/- hourDiff`.

That should about wrap up encapsulation and getters/setters for now, but while we are still in the neighborhood of operator overloading with `property()`, let's go a little deeper. We just saw how `=` can be made to invoke a method (our hour and minute getters). Can we overload something like the `+` operator? Of course we can!! It's time for our next section.

1.7.4 Operator Overloading / Magic Methods: Fractions

So, we have been using Python for a bit now, and we love what it has to offer. I mean, a default complex numbers class? Amazing!! But, where's the support for our rationals? You'd think that we'd have rational numbers far before complex numbers, but I'm not seeing them anywhere. Well, I suppose it's ok, we've known about fractions since elementary school, and we can make a `Fraction` class *all by ourselves.*

You know how we write constructors in Python: `def __init__(self,arg1,arg2...)`. Well, there are a whole bunch of other methods like that, ones for comparison, assignment, creation, deletion, arithmetic, conversion, and then some! If we want to look at them sans *Google-it-University* we can call `help(int)` to see some of the magic methods that the `int` class uses. We'll use some of them in our `Fraction` class! We want to be able to add, subtract, multiply, divide, exponentiate, compare, negate, floor, ceiling, and get int, float, and string representations. Here's our basic `Fraction` class:

```
import math
import sys
# OPERATOR OVERLOADING
class Fraction:
    def __init__(self, num, den = 1):
        if den == 0:
            raise ValueError("Denominator can't be 0")
            sys.exit()
        elif den < 0:
            # don't allow for negative denominators
            den *= -1
            num *= -1
        self.num = num
        self.den = den
        self.reduceFrac()
    def gcd(self, m, n):
        # euclids method for finding gcd of M and N
        r = m % n
        while r != 0:
            m = n
            n = r
            r = m % n
        return n
    def reduceFrac(self):
        gcd = self.gcd(self.num, self.den)
        self.num /= gcd
        self.den /= gcd
        if self.den < 0:
            self.den *= -1
            self.num *= -1
        self.num = int(self.num)
        self.den = int(self.den)
    def castFrac(self, other):
        if not isinstance(other, Fraction):
            try:
                other = Fraction(int(other))
            except Exception as e:
                print(e)
                sys.exit()
        return other
    # Overload <
    def __lt__(self, other):
        other = self.castFrac(other)
        self.reduceFrac()
        other.reduceFrac()
        return (self.num * other.den) < (other.num * self.den)
    # Overload >
    def __gt__(self, other):
        other = self.castFrac(other)
        self.reduceFrac()
        other.reduceFrac()
        return (self.num * other.den) > (other.num * self.den)
    # Overload ==
    def __eq__(self, other):
        other = self.castFrac(other)
        self.reduceFrac()
        other.reduceFrac()
        return (self.num * other.den) == (other.num * self.den)
    # Overload +
    def __add__(self, other):
        other = self.castFrac(other)
        den = self.den * other.den
        num = (self.num * other.den) + (self.den * other.num)
        return Fraction(num, den)
    # Overload -
    def __sub__(self, other):
        other = self.castFrac(other)
        den = self.den * other.den
        num = (self.num * other.den) - (self.den * other.num)
        return Fraction(num, den)
    # Overload *
    def __mul__(self, other):
        other = self.castFrac(other)
        den = self.den * other.den
        num = self.num * other.num
        return Fraction(num, den)
    # Overload /
    def __truediv__(self, other):
        other = self.castFrac(other)
        otherInv = Fraction(other.den, other.num)
        return self * otherInv
    # Overload **
    def __pow__(self, exp):
        return Fraction(self.num**exp, self.den**exp)
    # Overload str()
    def __str__(self):
        return str(self.num) + "/" + str(self.den)
    # Overload -Fraction
    def __neg__(self):
        return Fraction(-1 * self.num, self.den)
    # Overload abs(Fraction)
    def __abs__(self):
        return Fraction(abs(self.num), self.den)
    # Overload math.floor(Fraction)
    def __floor__(self):
        return self.num//self.den
    # Overload math.ceil(Fraction)
    def __ceil__(self):
        return math.ceil(self.num/self.den)
    # Overload int(Fraction)
    def __int__(self):
        return int(float(self))
    # Overload float(Fraction)
    def __float__(self):
        return self.num/self.den
if __name__ == "__main__":
    a = Fraction(2, 4)
    b = Fraction(1, 3)
    print(a, "+", b, "=", a+b)
    print(a, "-", b, "=", a-b)
    print(a, "*", b, "=", a*b)
    print(a, "/", b, "=", a/b)
    print(a, "<", b, "=", a<b)
    print(a, ">", b, "=", a>b)
    print(a, "=", b, "=", a==b)
    a+=b
    print(a)
    print(a, "^ 2 =", a**2)
    print("negative",a, "=", -a)
    print("int(",a*3, ") =", int(a*3))
    print("float(",a, ") =", float(a))
    print("Floor:",math.ceil(a), "Ceil:",math.floor(a))
    print("|",a, "* -11 | =", abs(a*-11))
    print(math.floor(a*"asd"))
```

Again, this doesn't require much explanation. We are simply doing an exercise in operator overloading. I've made note of what each method is overloading, and placed examples within the main. Hopefully that should more or less put a wrap on magic methods and operator overloading. There's some really cool stuff you can do with this though, and I urge you to take a look into it!

Now that we are closing our encapsulation and operator adventures, it's time to get back into the object oriented programming basics. It's time to venture towards inheritance and polymorphism!

1.7.5 Inheritance and Polymorphism

Inheritance is another one of those key concepts in object oriented programming. Its essence is the ability to define a class from another class to achieve some form of hierarchy of classes with certain shared attributes and methods. Any given class may extend/inherit from another class–its super/parent class. When one class extends some parent class, it inherits all public and protected methods of that parent class, however we may want to tweak some functionality within the subclass. This comes in the form of *method overriding* or *polymorphism*. Let's talk about that sentence though...

Method overriding is not the same as method overloading. Method overloading is compile-time polymorphism: more than one method will share the same name with different parameters. Python isn't traditional when it comes to method overloading, but we've seen it before; we can't have multiple methods in a class by the same name (at least, not without decorators), but we can set default parameters such as in our `Fraction` constructor, `def __init__(self, num, den=1)`, which allows for calling with or without a passed denominator. Method overriding, on the other hand, is run-time polymorphism and it is linked to inheritance. Method overriding is for creating a tailored implementation of a method inherited from a parent class.

Now, we've gotten into the polymorphism talk. It's a fancy term for something quite familiar: the ability for multiple things to use the same interface with catered results. It's the ability for something to take several forms. Perhaps we have two classes: `Triangle` and `Rectangle` they may both have a method called `area()`, the triangle's returning $\frac{1}{2} \cdot base \cdot height$ and the rectangle's giving $width \cdot height$. Boom, there you have it: polymorphism.

There is the topic of abstract classes. Abstract classes are just a base class from which to inherit from, they cannot be instantiated, but they can be subclassed. But they aren't natively supported in Python, and we have to use some module (`abc` from the standard library) to go about this.

That's about it for this whole explanation, let's get into an example. This one will be real short and sweet. We'll just do the classic animal example. We'll start with some `Animal` class, and we will make a few subclasses until we get the hang of it. I'm thinking that the hierarchy should be as follows: `Animal` is the abstract base class (hence `abc`), then `Bird` and `Mammal` will extend `Animal`. And just for kicks (or cause we want to showcase extension of subclasses) we will have `Squirrel` and `Dolphin` extend `Mammal`.

```python
from abc import ABC, abstractmethod
# Base Class
class Animal(ABC):
    def __init__(self):
        pass
    def domain(self):
        pass
    def move(self):
        pass
    def funFact(self):
        pass
    def printStuff(self):
        print("Domain:",self.domain())
        print("Movement Method:",self.move())
        print("Fun Fact:",self.funFact())
# child class
class Bird(Animal):
    def __init__(self):
        print("Real bird stuff")
    def domain(self):
        return "The sky"
    def move(self):
        return "Flight"
    def funFact(self):
        return "The Eagles won the super bowl"
# child class
class Mammal(Animal):
    def __init__(self):
        print("Mammal gang")
    def domain(self):
        return "The land"
    def move(self):
        return "Walkin and shit"
    def funFact(self):
        return "The bible says God is a mammal"
# grandchild class
class Squirrel(Mammal):
    def funFact(self):
        return "God is a squirrel"
# grandchild class
class Dolphin(Mammal):
    def domain(self):
        return super().domain()+", but more so, water"
if __name__ == "__main__":
    bird = Bird()
    mammal = Mammal()
    squirrel = Squirrel()
    dolphin = Dolphin()
    print("\nBird stuff:")
    bird.printStuff()
    print("\nMammal stuff:")
    mammal.printStuff()
    print("\nSquirrel stuff:")
    squirrel.printStuff()
    print("\nDolphin stuff:")
    dolphin.printStuff()
```

Hopefully this makes sense! We have `Animal` that is more or less useless, but spells out what something that extends it will have. Then `Mammal` and `Bird` both implement the unimplemented methods of `Animal` in a way that reflects them. Then, we have `Squirrel` that overrides the `funFact()` of `Mammal` to say something else. And finally, we have `Dolphin` which overrides the `domain()` of `Mammal`, but also invokes that `domain()` through `super().domain()`! Inheritance and polymorphism can be incredible tools in keeping clean, expressive, non-repetitive code, and they're certainly worth getting into.

Here marks our wrap on object oriented programming. We made a bunch of classes, had some fun with encapsulation, and got a bit into inheritance and polymorphism. Along the way we also saw plotting with `matplotlib`, processing user input with `input()`, operator overloading and magic methods, getters and setters, and some basic exception handling. That was quite the journey, you owe yourself a pat on the back!

Now that we have a good deal of fundamentals under our belt, we are going to get into some more practical business. Next up we are going to look at dealing with files (reading and writing).

1.8 File Manipulation

Dealing with files is a thing of beauty with Python. Things are quick to set up and use, and we can do a whole bunch of stuff! We have basic reading and writing with files, we saw working with JSON in our prime numbers example, we have a whole bunch of utilities through modules like `shutil` and `os`, and then some!

Python has some great systems capabilities, the more basic of which some in terms of making, moving, copying, and deleting files and directories. I'm sure this is a practice you are familiar with, but there are certainly reasons to stray from GUIs (grody/graphical user interfaces) when it comes to file manipulation. Whether it's for speed, automation, or looking *cool*, we can always hone our systems skills. So, let's get into that a bit:

```
import os
import shutil
from datetime import datetime
def printDir(path='.', numTabs=0, recursive=False):
    '''
        This guy is chill to print out all files
        in a directory (including subdirs!)
    '''
    directory = os.scandir(path)
    tabs = '|'
    for i in range(0, numTabs+1):
        tabs += '\t'
    for entry in directory:
        if entry.is_file():
            print(tabs, "File:",entry.name)
        elif entry.is_dir():
            print(tabs, "Directory:", entry.name)
            if recursive:
                printDir(path + "/" + entry.name, numTabs+1)
def printInfo():
    '''
        This guy is chill to print OS information
        regarding a file in the current directory!
        We also get some dealings with time (the devil)
    '''
    toMDY = lambda time : datetime.utcfromtimestamp(time).strftime('%b %d, %Y - %H:%M:%S'
        )
    directory = os.scandir('.')
    spacer = " -"
    for entry in directory:
        print(entry.name)
        info = entry.stat()
        print(spacer, "File type and permissions:",info.st_mode)
        print(spacer, "Inode number:",info.st_ino)
        print(spacer, "Device ID:",info.st_dev)
        print(spacer, "Number of hard links:",info.st_nlink)
        print(spacer, "UID:",info.st_uid)
        print(spacer, "GID:",info.st_gid)
        print(spacer, "Size (bytes):",info.st_size)
        print(spacer, "Last access time (s):",toMDY(info.st_atime))
        print(spacer, "Last modify time (s):",toMDY(info.st_mtime))
        print(spacer, "Last metadata change time (s):",toMDY(info.st_ctime))
        break
def makeDirsAndFiles():
    '''
        This guy makes a bunch of directories and files
    '''
    try:
        os.mkdir('./MyNewDirectory')
    except FileExistsError as e:
        print("Directory exists!")
    os.makedirs('./MyNewDirectory/My/New/Directory', exist_ok=True)
    for i in range(0,4):
        for extension in ['.py', '.txt']:
            fileName = "MyNewDirectory/My/New/Directory/"+str(i)+extension
            with open (fileName, 'w') as f:
                f.write("Oh hi there :)")
def copyAndMove():
    '''
        This guy copies stuff and moves stuff
    '''
    basedir = "./MyNewDirectory/My/New/Directory/"
    try:
        shutil.copytree(basedir, basedir+"backup")
    except FileExistsError as e:
        print("Already exists :)")
    os.rename(basedir+"backup", basedir+"backup2")
    os.rename(basedir+"backup2/0.py", basedir+"backup2/cool.py")
    shutil.move(basedir+"backup2/", basedir+"renamedIt/")
    shutil.move(basedir+"renamedIt/", "./MyNewDirectory/My/New")
def deleteStuff():
    '''
        This guy deletes stuff we just made
    '''
    basedir = "./MyNewDirectory/My/New/renamedIt/"
    directory = os.scandir(basedir)
    for entry in directory:
        os.remove(basedir + entry.name)
    os.rmdir(basedir)
    basedir = "./MyNewDirectory/My/New/Directory/"
    directory = os.scandir(basedir)
    for entry in directory:
        os.remove(basedir + entry.name)
    basedir = basedir[:basedir.rfind('/')]
    while basedir.rfind('/') != -1:
        os.rmdir(basedir)
        basedir = basedir[:basedir.rfind('/')]
if __name__ == "__main__":
    printDir(recursive=False)
    printInfo()
    makeDirsAndFiles()
    copyAndMove()
    deleteStuff()
```

If we run this program as is, it should do pretty much nothing! That's probably for the better in this instance–I can't think of much reason that we would want a bunch of numbered files in strange folders that say "Oh hi there :)"–but I guess it's cute. Let's chat about what is happening here though: `printDir` will print out the contents of a directory (and subdirectories, if you want!), `printInfo` spits out a bunch of operating system jargon and the timestamps are converted to human readable time, then `makeDirsAndFiles`, `copyAndMove`, and `deleteStuff` make a bunch of directories and files, copy and move them, and then delete them all! A bit more detail is warranted, so let's dive in:

`printDir` takes in some string representing a path to a directory, and uses `os.scandir` to gather the files within that directory. Then, we check if the file is a standard file or if it is a subdirectory; in either case we print its name (prepended by tabs proportionate to its depth), and if it is a subdirectory we may recurse on it. `printInfo` is a little method to show some OS functionality, we get a file and print a bunch of file attributes–those of which that are unix timestamps are converted to be human readable (look up unix timestamps and the 2038 problem, it's good table talk). `makeDirsAndFiles` makes directories in two ways: one via `os.mkdir` that makes only one directory, and one via `os.makedirs`–which is more or less equivalent to our familiar `mkdir -p` with our shells–and then we have some nested loop that creates a bunch of files with `open` and `write`. `copyAndMove` isn't the most elegant–we are hardly doing any exception handling, so I emplore you to look at doing that as an exercise–but, we are looking at copying and moving things: first we have `shutil.copytree` that recursively copies all files within a subdirectory to some new directory, we have `os.rename` that can rename a standard file or directory, and we have `shutil.move` that lets us move a directory to a new directory and move a directory to within an existing directory. Finally, we have `deleteStuff` that goes through the files and folders that we just made and deletes them: we use `os.scandir` paired with `os.remove` to get and remove the files in the directories that we wrote in or copied/moved to and we used `os.rmdir` to remove the empty folders (there is `shutil.rmtree` to remove non-empty directories, but that's *spooky*); there's one neat thing here too, we use `rfind` to break down the directory string. Python has some right-operators, such that we can specify (or things may be checked under the hood) whether we mean `2+3` as `2+3` or as `3+2`: for example our `Fraction` class allows for addition of integers, but the `int` class doesn't directly support adding `Fractions`, so there we have binary operators to the rescue! `rfind` then allows for us to look for '/' from the right-most index to break down our empty directories.

That about wraps up our jig on basic file manipulation, and next up we are going to dive a bit into working with CSV (comma separated values). Which is a lovely table format to work with, and we can import and export CSV to things like Google Sheets!

1.8.1 Working with CSV

We love CSV! It's a wonderful form of data. It's just tabular data in a text file, and we are already familiar with it through things like Excel/Sheets. So let's get started.

CSV is quite popular (as is TSV for tab separated values, but a little less so), and thus we have a myriad of ways to deal with it. We're going to look at three popular ways of reading and writing CSV: with no modules, with the `csv` module, and with `pandas`. The `csv` module comes via the standard library, so we should be all set for that, but `pandas` may need installation (`pip install pandas`). Pandas may be the most used of these practices, as it's a very popular module for data analysis–I highly recommend taking a further look into it! I hope that's an apt introduction, let's get into the code:

```python
import matplotlib.pyplot as plt
import csv
import pandas
import re
data = {}
filename = "SalesJan2009.csv"
writefile = "test.csv"
writeLabels = ["Name", "Department", "Favorite Food", "ID"]
writeList = [
    ["Cole", "Math and Computer Science", "Popeyes", 11],
    ["Emma", "Math and Econ", "Might be Popeyes", 26],
    ["Veronika", "Math and Econ", "Dolphinately Popeyes", 7]
]
writeDict = [
    {"Name": "Cole2", "Department": "Math and Computer Science", "Favorite Food": "
        Popeyes", "ID": 117},
    {"Name": "Emma2", "Department": "Math and Econ", "Favorite Food": "Might be Popeyes",
         "ID": 268},
    {"Name": "Veronika2", "Department": "Math and Econ", "Favorite Food": "Dolphinately
        Popeyes", "ID": 73}
]
def noModuleRead():
    '''
        This reads a csv file into a dictionary
        without any modules: note the removal of
        quotes and commas
    '''
    data = {}
    with open(filename, "r") as f:
        csvLines = f.readlines()
        for label in csvLines[0].strip().split(','):
            data[label] = []
        for line in csvLines[1:]:
            # get rid of quotation marks/in-field commas
            offset = 0
            while line.find('"') != -1:
                firstInd = line.find('"', offset)
                secondInd = line.find('"', firstInd+1)
                offset = secondInd+1
                if secondInd != -1:
                    middle = line[firstInd:secondInd].replace(',', '').replace('"', '').
                        strip()
                    diff = len(line[firstInd:secondInd]) - len(middle)
                    offset += diff
                    line = line[:firstInd] + middle + line[secondInd+1:]
            # add entry to dictionary at proper field
            for entry, label in zip(line.split(','), data.keys()):
                data[label].append(entry.strip())
    return data
def noModuleWrite():
    '''
        This writes sample csv data to a file
        without any modules
    '''
    with open(writefile, "w") as f:
        f.write(','.join(writeLabels) + "\n")
        # writing from lists
        for line in writeList:
            f.write(','.join([str(entry) for entry in line]) + "\n")
        # writing from dicts
        for line in writeDict:
            f.write(','.join([str(line[label]) for label in writeLabels]) + "\n")
def csvModuleRead():
    '''
        This reads a csv file into a dictionary
        with the csv module
    '''
    data = {}
    with open(filename, "r") as f:
        csvLines = csv.DictReader(f)
        labels = False
        for line in csvLines:
            # initialize key-value pairs in dict
            if not labels:
                labels = True
                for label in line:
                    data[label] = [line[label]]
            else:
                # add new entries
                for label in data.keys():
                    data[label].append(line[label].strip())
    return data
def csvWrite():
    '''
        This writes sample csv data to a file
        with the csv module
    '''
    with open(writefile, "w") as f:
        # writing from lists
        csvWriter = csv.writer(f)
        csvWriter.writerow(writeLabels)
        csvWriter.writerows(writeList)
        # writing from dicts
        csvWriter = csv.DictWriter(f, fieldnames=writeLabels)
        csvWriter.writerows([entry for entry in writeDict])
def pandasRead():
    '''
        This reads a csv file into a dictionary
        with the pandas module
    '''
    data = {}
    df = pandas.read_csv(filename)
    data.update(df)
    return data
def pandasWrite():
    '''
        This writes sample csv data to a file
        with the pandas module
    '''
    # writing from list
    df = pandas.DataFrame(writeList, columns=writeLabels)
    # writing from dicts
    df = df.append(pandas.DataFrame(writeDict, columns=writeLabels), ignore_index=True)
    df.to_csv(writefile, index=False)
def printEntry(index):
    '''
        prints the entry at a passed index
    '''
    print(f"\nPrinting entry {index}:")
    for label in data.keys():
        print(label, ":", data[label][index])
def getAvgPriceForCard():
    '''
        This returns a dictionary where each key corresponds
        to a card type, and we have a list for the value
        where the first index is average price and the second
        is the number of data points (non-numeric chars removed)
    '''
    paymentToCard = {}
    # use regex to get rid of in-field commas
    intPrice = lambda price : int(re.sub('[^0-9]+', '', str(price)))
    for price, card in zip(data["Price"], data["Payment_Type"]):
        if card not in paymentToCard:
            paymentToCard[card] = [intPrice(price), 1]
        else:
            # sum prices per card
            paymentToCard[card][0] += intPrice(price)
            paymentToCard[card][1] += 1
    # divide each sum by number of points
    for card in paymentToCard:
        paymentToCard[card][0] /= paymentToCard[card][1]
    return paymentToCard
def plotStuffMPL():
    '''
        plots average payment by card (matplotlib)
    '''
    cards = avgPayment.keys()
    avg = [ avgPayment[card][0] for card in avgPayment ]
    padding = (max(avg) - min(avg)) / 2
    plt.ylim(min(avg) - padding, max(avg) + padding)
    plt.bar(cards, avg)
    plt.ylabel('Average Payment')
    plt.ylabel('Card')
    plt.title('Average Payment by Card')
    plt.show()
```

```
def plotStuffPD():
    '''
        plots average payment by card (matplotlib and pandas)
    '''
    df = pandas.DataFrame(avgPayment)
    row = df.iloc[0]
    row.plot.bar()
    plt.ylabel('Average Payment')
    plt.ylabel('Card')
    plt.title('Average Payment by Card')
    plt.show()
if __name__ == "__main__":
    # Testing our three methods of reading!
    data = noModuleRead()
    printEntry(3)
    data = csvModuleRead()
    printEntry(3)
    data = pandasRead()
    printEntry(3)
    # Use our data to plot something!
    avgPayment = getAvgPriceForCard()
    plotStuffMPL()
    plotStuffPD()
    # Write some CSV to a file
    noModuleWrite()
    csvWrite()
    pandasWrite()
```

We're going to dive in to what's going on here. First off, we define some different things we might want to write to our `test.csv` file: we want to be able to write from lists and from dictionaries, so we filled one of each with sample data.

Then we get into reading csv: note we reset `data` to be an empty dictionary at the beginning of each read method so that we aren't just extending the same dictionary. In `noModuleRead` we read the file line-by-line, splitting each line on commas. We separate the first line (of labels) and create the keys of our dictionary from this. We then look over the rest of the file (the actual data) and remove all in-field commas and quotation marks before splitting on commas; to update the entry in the dictionary, we zip the current split line with the keys of our dictionary to ease the mapping onto our dictionary. In `csvModuleRead` we go through the lines of our file using a `DictReader`; we must distinguish the first line as we have to initialize key-value pairs in data, then we may just append the line at each label to our dictionary at that label. Then, pandas has the simplest mehod: in `pandasRead` we may just use `pandas.read_csv` to read into a `pandas.DataFrame` which plays nicely with dictionaries.

Writing is a similar story, we are just looking at how to write lists vs dictionaries. In `noModuleWrite` we open a file for writing as usual, then we write all of our data by using `','.join(list) + ''\n''`: this creates a string where each element in the list is separated by a comma, and we end with a new line. Then we loop over our list and write each list entry ensuring that they are strings, followed by looping over our dictionary doing the same (we must access the dictionary though!). For `csvWrite` we are use a `csv.writer` to write our lists and a `csv.DictWriter` to write our dictionary. We open our file, and either use `csv.writer.writerow` or `csv.writer.writerows` depending on the plurality of the passed parameters. Then we use our `DictWriter` to be able to specify the `fieldnames` for our dictionary. In `pandasWrite` we will be using `pandas.DataFrame.to_csv` to write, which of course takes in a dataframe. We will create one with our list, noting that the columns are our labels, then we append another dataframe that we make using our dictionary: note, when we are appending, we will use `ignore_index=True` to continue indexing as usual (this is unnecessary here as we set `index=False`, but it's worth seeing).

Now, let's wrap things up. We may want to check to see what's going on after we've parsed our file, so we wrote up a little method `printEntry` which takes an index and prints the corresponding entry. And then, we'd like to make some chart from our data: perhaps we would like to look at the average priced purchase for each card. We will first aggregate this data within `getAvgPriceForCard`: we simply loop over all the entries "price" and "card" columns and keep a sum and number of prices for each card and divide by the number when we are done. Now we can plot this data in our brilliantly named `plotStuffMPL` with `matplotlib`. We will use our newfound card-averages dictionary–card name on the x-axis and average price on the y-axis. We set the y-axis bounds based on the min and max values offset by some padding (for your viewing pleasure!) determined by the average of the max and the min–this is unimportant. We use `plt.bar` and give it our x and y values to make the chart, and we display it with `plt.show` as usual. We may, alternatively, plot with `pandas` and `matplotlib` which we do with `plotStuffPD`, all the funny business with padding axes is absent in this method, however.

This should be a reasonable introduction to working with data in Python, but of course there is so much more! I strongly recommend diving into Pandas if this is the realm you are going to be in, there's some wonderful bits with working with missing data, and there's even more in store for you if you want to do things like linear regression via `sklearn` or whatnot. We will get into that a bit later, as `scipy` and `pandas` *both* rely on `numpy` and we have hardly mentioned that thus far.

1.8.2 Numpy

Numpy is the most prevalent mathematics module, and it largely revolves around the n-dimensional array (ndarray). This may be used as a vector, matrix, or tensor, and it is heavily optimized (whatever that means in Python, am I right??). But in all seriousness, ndarrays don't carry pointers to all of their elements' data (the way that standard lists do), and the metadata kept around for them is mostly in shape and data-type. The restriction on having a single type allows for far more efficient accessing and manipulation (don't worry about it if that doesn't make a lot of sense), and we can–of course–specify the data-type, allowing for as small as 8-bit integers. So, let's take a look at some base functionality of `numpy`!

```
import numpy as np
# making ndarrays in numpy
a = np.array([1, 2, 3]) # 3x1 ndarray
b = np.array([1.1, 2.2, 3.3]) # 3x1 ndarray with floats
c = np.array([[1, 2, 3], [4, 5, 6]]) # 2x3 ndarray
d = np.zeros([3, 5]) # 3x5 ndarray (all 0)
e = np.ones([2, 3, 5]) # 2x3x5 ndarray (all 1)
f = np.empty([3, 4]) # 3x4 ndarray (random from memory)
g = np.arange(3, 10) # evenly spaced 1d ndarray from 3-10 (stepsize = 1)
h = np.arange(10, 20, 2.5) # evenly spaced 1d ndarray from 3-10 (stepsize = 2.5)
j = np.linspace(2, 8, 12) # evenly spaced 1d ndarray from 2-12 with 12 steps
i = np.ones_like(j) # creates ndarray of ones in the shape of j
k = np.arange(6) # evenly spaced 1d ndarray from 0-6
l = k.reshape(2, 3) # reshape k to a 2x3 ndarray
m = np.arange(24).reshape(2, 3, 4) # creates a 2x3x4 ndarray
n = np.random.random([2, 3, 4]) # 2, 3, 4 ndarray (random from 0-1)
o = np.random.random([2, 3, 4]).reshape(-1, 8) # reshapes ndarray to _x8 (must be
    possible)
numpyArrs = [a, b, c, d, e, f, g, h, i, j, k, l, m, n, o]
for index, arr in enumerate(numpyArrs):
    print(chr(ord('a') + index).upper()+", shape:", np.shape(arr))
    print(arr, "\n")
# make an ndarray from a function
func = lambda a, b, c : a + 2*b + 3*c # doesn't have to be a lambda function
a = np.fromfunction(func, (2, 3, 2))
print(a)
for entry in a.flat: print(entry)
for subarr in a: print(subarr)
# matrix business: (All arithmetic operations are elementwise)
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
c = np.arange(16).reshape(4,4)
print(a * b) # elementwise multiplication
print(a**2) # elementwise exponentiation
print(a+2) # elementwise addition
a*=b # set a to a*b
print(a)
# various numerical niceties
print(a.min(), a.max(), a.sum(), a.mean())
print(c, c.sum(axis=0), c.cumsum(axis=1), sep="\n")
print(c.min(axis=0), c.max(axis=1), sep="\n")
# universal functions: also operate elementwise
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.exp(a), np.sqrt(a), np.add(a, b), np.sin(a), sep="\n")
a[0, 1] = -33
a[1, :] = 100
print(a)
# shallow copy vs deep copy
a = np.arange(12).reshape(3, 4)
b = a[:,0]
b[:] = -100
print(a)    # shallow copy
a = np.arange(12).reshape(3, 4)
b = a[:,0].copy()
b[:] = -100
print(a)    # deep copy
# linalg submodule!
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
c = np.array([0.8, -3])
print(np.dot(a, b), a.dot(b), a @ b, np.matmul(a, b), sep="\n") # matrix multiplication (
    a @ b shortcut requires python 3.5 or higher)
print(np.transpose(a), a.transpose(), a.T, sep="\n") # matrix transpose
print(np.eye(3)) # identity matrix
print(np.linalg.inv(a)) # inverse of a
print(np.linalg.norm(a)) # norm of a
print(np.linalg.cond(a)) # condition number of a
print(np.linalg.det(a)) # determinant of a
print(np.linalg.matrix_rank(a)) # rank of a
print(np.trace(a)) # sum over diagonal of a
print(np.linalg.solve(a, c)) # solving linear equations
```

There's not much to say here. Everything that's done is more-or-less a one-off command. I'll try and explain a bit though! First off we have a bunch of ways to create ndarrays. We have the standard constructor that takes in a list of values (which may be more lists or whatnot). There are `zeros`, `ones`, `empty`, and `random.random` that fill the array (with passed shape) with zeros, ones, bits from memory, or random numbers. We also have things such as `ones_like` that takes the shape of a passed ndarray. There is `arange` which gives an ndarray beginning from some value (default is 0), ending at a passed parameter, and taking steps of a passed size (default is

1). This is similar to `linspace` which evenly spaces values from a passed beginning to end in a specified number of steps. Then, we can use `reshape` to morph our ndarray to fit a different shape (so long as it works with our size). Then we look at making an ndarray by passing a shape through a function wrap up our ndarray creation section.

Real quick sidenote on how I am printing the first arrays: I enumerate a list of all the ndarrays which allows us to unpack the array and the index, then we use the index to offset a char beginning at 'a'. Just in case you hadn't seen that before.

All of the standard arithmetic operations work elementwise on ndarrays (a b will not be their dot product). We have some utilities like `min()`, `max()`, `sum()`, `mean()`, and `cumsum()`, all of which we can specify an axis for. We have "universal functions" that operate elementwise as well, like the exponential function, e^x, square root, and sin. We index with comma separated indices in a list, and we may use slices as normal.

We have the difference between shallow and deep copying, such that shallow copying (the more intuitive assignment) does not create a unique object, but deep copying via `ndarray.copy` will.

Then, we close things with linear algebra niceties. We have a bunch of ways to take the dot product and a few to take the transpose. Them, we have matrix operations for the inverse, norm, condition number, determinant, rank, trace (sum over diagonal), and solving linear equations.

I have just one more example for this section. We're going to look at another graphing example, and it's going to use things from a bunch of modules! We'll use `datetime`,`numpy`, `matplotlib`, `pandas`, `scipy`, and `sklearn` to look at some Australian rain data. We have gotten the csv data from kaggle, which is a popular data set site. We will read in the data, take the columns we care about, normalize the data, and plot it against a normal distribution.

```python
# https://www.kaggle.com/jsphyg/weather-dataset-rattle-package
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy import stats
from sklearn import preprocessing
# read csv into dataframe
df = pd.read_csv("weatherAUS.csv", date_parser=lambda x: datetime.strptime(x, "%Y-%m-%d")
    )
df = df[np.isfinite(df['Rainfall'])]
df = df[np.isfinite(df['Pressure9am'])]
df = df[np.isfinite(df['Pressure3pm'])]
df.fillna(0, inplace=True)
print(df.head())
data = {
    "Rainfall": df['Rainfall'],
    "MeanPressure": (df['Pressure9am']+df['Pressure3pm'])/2
}
df = pd.DataFrame(data)
# normalize data
normalizedData = pd.DataFrame(preprocessing.MinMaxScaler().fit_transform(df))
# set up plot
plt.scatter(normalizedData[1], normalizedData[0], alpha=0.1)
plt.ylabel("Rainfall")
plt.xlabel("Average Pressure (9am-3pm)")
plt.title("Average Pressure vs Rainfall")
# plot multivariate distribution
cov = np.cov(normalizedData[1], normalizedData[0])
mean = np.array([np.mean(normalizedData[1]),  np.mean(normalizedData[0])])
x, y = np.random.multivariate_normal(mean, cov, 5000).T
plt.plot(x, y, 'x', color="red", alpha=0.1)
# plot univariate distributions for each var
# x axis
scale = 5
mu = np.mean(normalizedData[1])
stddev = np.sqrt(np.var(normalizedData[1]))
x = np.linspace(mu - scale*stddev, mu + scale*stddev, 100)
plt.plot(x, stats.norm.pdf(x, mu, stddev), color="green")
# y axis
mu = np.mean(normalizedData[0])
stddev = np.sqrt(np.var(normalizedData[0]))
x = np.linspace(mu - scale*stddev, mu + scale*stddev, 100)
plt.plot(x, stats.norm.pdf(x, mu, stddev), color="orange")
plt.show()
```

We read in csv with `pandas` and set a `date_parser` to convert the strings for dates into datetime objects (we don't use this here, but you might want to!). We do some quick processing so that we only consider rows where "Rainfall", "Pressure9am", and "Pressure3pm" all exist. Then we set all other NaN values to 0 (we probably don't want that in most cases, but this is just for you to see how you would do that). We use `df.head()` to print out the first 5 rows, as we just want to look at what's going on. Then, we do some quick processing to average the "Pressure9am" and "Pressure3pm" data and take our dataframe to be the rainfall and mean pressure. Then, to get all values between 0 and 1 we use the `MinMaxScaler` from `sklearn.preprocessing`. Now, we just make a scatterplot from our normalized data, and we set a small alpha value to get a bit of a sense of data density. Then, we get some stats information with `numpy` to plot multivatiate and univariate distributions (I guess?). We need the covariance matrix for our data, as well as our means and standard deviations for each variable. Then we use `np.random.multivariate_normal` with our covariance, means, and some amount of sample points to plot against our data. We finish off by plotting our univariate distributions that use our means and standard deviations for each variable, and plot `stats.norm.pdf` against our values! Forgive me if any of that was wrong, I don't know anything about statistics...

Anyways, let's move past this, I hope you can see how we can use this though. Let's move on to web scraping.

1.9 Web Scraping

We're gonna do some web scraping together!!! We just saw some stuff on Kaggle, so maybe we are all excited and we'd like to work on making our own dataset. I, personally, would like to compile a bunch of lyrics by music genre (for research purposes). So, first, we want to browse for options for scraping: what looks structured in the most conducive way for us to work with? We can look at a bunch of lyric sites and take a look at what we want from them, then try and think about how we would have to set up crawlers for each of them. I ended up picking "metrolyrics," as they had a nice alphabetized table of artists (that included genre), and there was a nice flow from the table to an artist page to their songs. So then, we start to set up our scraper. We will use `BeautifulSoup` from `bs4`, which is a standard for web scraping (sometimes we may use things like `Selenium` if we need to simulate a browsing session). But, in general, the web is a wonderful data collection playscape, and Python is often a nice way to go about scraping–scraping with JS and `cheerio` is pretty nifty too!

Now that we have landed on metrolyrics, let's plan our crawler. The artist tables are paginated in a very human-readable form: theres a character representing the first letter of the artist followed by a number for result page. We can loop over those easily, and at each step we can collect the names, genres, and urls for each artist. Then we can go to the artist page and collect song names and urls. Then finish by going to the song pages and collecting the lyrics! Then, we want to write things to files. It may take a while to run, so let's break it up into a bunch of files as we go, both for memory purposes and for our ability to stop the program without losing much time. Then, given the nature of our writing strategy (one dictionary per artist at a time, appended to the file), we will want to clean our JSON to make it valid. Then we can take our files and combine them to one more cohesive dataset. Let's take a look at the code:

```
import urllib3
from bs4 import BeautifulSoup
import requests
import json
import os
import re, string
import sys
import pprint
class LyricScraper:
    def __init__(self):
        self.bases = list("1abcdefghijklmnopqrstuvwxyz")
    def scrapeMainArtists(self, numPages=1, letterInd=0):
        baseUrl = "http://www.metrolyrics.com/artists-"+self.bases[letterInd]+"-"
        artistList = []
        for i in range(0,numPages):
            url = baseUrl + str(i) + ".html"
            page = requests.get(url)
            soup = BeautifulSoup(page.content, "html.parser")
            artistTable = soup.find("tbody")
            allRows = artistTable.find_all("tr")
            for row in allRows:
                allEntries = row.find_all("td")[0:2]
                artistName = allEntries[0].text.replace("Lyrics", "").strip()
                artistGenre = allEntries[1].text.strip()
                if artistGenre is "":
                    artistGenre = "Unknown"
                artistUrl = allEntries[0].find("a", href=True)['href']
                artistList.append({
                    "name": artistName,
                    "genre": artistGenre,
                    "url": artistUrl,
                })
        return artistList
    def scrapeArtist(self, artist):
        songlist = []
        url = artist['url']
        page = requests.get(url)
        soup = BeautifulSoup(page.content, "html.parser")
        artistTable = soup.find(class_="songs-table compact").find("tbody")
        allRows = artistTable.find_all("tr")
        for row in allRows:
            songEntry = row.find_all("td")[1]
            songName = songEntry.text.replace("Lyrics", "").strip()
            songUrl =  songEntry.find("a", href=True)['href']
            songlist.append({
                "songname": songName,
                "songurl": songUrl
            })
        artist["songlist"] = songlist
        return artist
    def scrapeSong(self, artist):
        for song in artist["songlist"]:
            lyrics = ""
            page = requests.get(song["songurl"])
            soup = BeautifulSoup(page.content, "html.parser")
            allLyrics = soup.find_all(class_="verse")
            translator = str.maketrans('', '', string.punctuation)
            for lyric in allLyrics:
                lyrics += lyric.text.lower().translate(translator).replace('\n',' ')
            song["lyrics"] = lyrics
        return artist
    @staticmethod
    def scrapeSingleSong(self, url):
        lyricsDict = {
            "url": url,
            "lyrics": ""
        }
        page = requests.get(url)
        soup = BeautifulSoup(page.content, "html.parser")
        allLyrics = soup.find_all(class_="verse")
        translator = str.maketrans('', '', string.punctuation)
        for lyric in allLyrics:
            lyricsDict['lyrics'] += lyric.text.lower().translate(translator).replace('\n'
                ,' ')
        return lyricsDict['lyrics']
    def run(self, numPages=1):
        for i in range(0, len(self.bases)):
            artists = self.scrapeMainArtists(numPages, i)
            for artist in artists:
                artist = self.scrapeSong(self.scrapeArtist(artist))
                self.write(artist, f"songscraping/songscraping-{self.bases[i]}.json",
                    True)
        self.fixJson()
        self.makeDset()
    def write(self, artist, filename, append=False):
        with open(filename, 'a' if append else 'w') as outfile:
            json.dump(artist, outfile, sort_keys = True, indent = 4)
    def fixJson(self):
        directory = os.scandir('./songscraping')
        for filename in directory:
            if filename.is_file():
                if filename.name.endswith('json'):
                    filestr = ""
                    with open('./songscraping/'+filename.name, "r") as f:
                        filestr = f.read().replace("}{", "},{")
                    with open('./songscraping/'+filename.name, "w") as f:
                        f.write("[\n" + filestr + "\n]")
    def makeDset(self):
        directory = os.scandir('./songscraping')
        genreLyrics = {}
        '''
            {
                "rock": {
                    "lyrics": "askdjhkalsdkj",
                    "artists": ["one", "two",...]
                }
            }
        '''
        for filename in directory:
            if filename.is_file():
                if filename.name.endswith('json') and filename.name.startswith("
                    songscraping"):
                    filestr = ""
                    with open('./songscraping/'+filename.name, "r") as f:
                        currentList = json.load(f)
                        for artist in currentList:
                            print(artist["name"], artist["genre"])
                            artistname = artist["name"].lower().strip()
                            genre = artist["genre"].lower().strip()
                            if genre not in genreLyrics:
                                genreLyrics[genre] = {"lyrics":"", "artists":[]}
                            if artistname not in genreLyrics[genre]["artists"]:
                                    genreLyrics[genre]["artists"].append(artistname)
                                for song in artist["songlist"]:
                                    genreLyrics[genre]["lyrics"]+=song["lyrics"]+"\n"
            with open('./songscraping/dset.json', "w") as f:
                json.dump(genreLyrics, f, indent=2)
if __name__ == "__main__":
    scraper = LyricScraper()
    scraper.run()
```

Woo! That was fun. If you ran it, it might be a while–plenty of time to understand it! We made a class for it, just for the purpose of abstraction. Then, we did just as we planned! It may be because the "planning" was edited, but that's neither here nor there. We have a list for the alphabet (and "1") for accessing the artist table pages on metrolyrics that is made in the constructor. Next we shall trace through `LyricScraper.run()` to take a look at what we are doing.

We go through our alphabet (and the number 1) in the loop from `0` to `len(self.bases)`, we get the artists corresponding to the current character, then loop over those. We scrape the songs for each artist, then write that artist dictionary to a file. After we are done writing all the artists, we go through the files and fix the invalid JSON that we wrote. Note: we wrote invalid JSON on purpose so that we can stop the program whenever, but we have to keep this in mind and fix it! Then after we fix the JSON, we can make some sort of more polished dataset.

I'll be a bit more verbose regarding the use of BS4, but there's not a whole lot of complexity here. With `scrapeMainArtists`, we get the url by inserting in our current character to the base url. Then we loop over the number of pages that we want to visit. At each step, we "visit" the page by making a `get` request to the url (more on that later too, but feel free to look up REST apis). Then we search the content for the `tbody` tag (table body, if you're not sharp on your html) and get all its rows (`tr` is table row). Then for each row, we get the cells that we care about (the first two `td`–table cell–elements), then we just do some string processing and toss things into a dictionary! Note that for the url, we get the href for the `a` tag under the first cell.

The other methods take a similar form. We do, however, do some cleaning in scraping the songs where we remove special characters and make everything lowercase! Also, we have a static method, `scrapeSingleSong` which may be used without an instance of `LyricScraper`. You may just call `LyricScraper.scrapeSingleSong(‘‘someurl.forfun’’)`!

1.10 Bye for now (and only for now)

That just about wraps things up for now. There’ll be more to come eventually, but I’m tired and this takes a lot of time. It is really fun to work on though! So I’m looking forward to getting back into it when I’ve got more time on my hands.