CS630 Data Mining, Fall 2016 Name _____

Python Crash Course Date _____

This worksheet asks you a variety of questions in Python. It assumes you have some familiarity with programming, just not Python. Pay special attention to Problem 1!

Important Note: throughout this assignment, and indeed the entire course, problems may be worded in the style of "find the answer"; I want you to **submit the code that finds that answer!** See "How to submit assignments in this course" in the course expectations sheet for information on what is expected of you regarding formatting.

1. Below is a list of websites. Each of them is a different take on the task of "I'm going to assume you know how to program, now here's how you do it in Python.". Read through at least one of them carefully, and glance through the others, as the remaining problems in this assignment, as well as the rest of the course, will assume you are capable of figuring out whatever you need to answer the question (from one of these sources or from somewhere else).

   - The Python Guru: `http://thepythonguru.com`
   - An Informal Intro to Python: `https://docs.python.org/3/tutorial/introduction.html`
   - Crash into Python: `https://stephensugden.com/crash_into_python/`
   - Dive into Python 3: `http://www.diveintopython3.net/index.html`

   Which ones did you like/dislike, which one did you use the most on this assignment?
   Install Python 3 as we discussed in class, through the anaconda/miniconda distribution:
   `https://www.continuum.io/downloads`
   Side Note: Feel free to find your own reference material online, just beware the version of Python that it uses! In this class we'll use Python 3 and not Python 2; I'm happy to discuss why with anyone interested.

2. Python is an interpretted language.

   a. **What does this mean?**

   b. People often say that Python can be used "as a calculator". Understand why this is, and how to do it. **Can Java be used in the same way? Why/how or why not?** Play around with this enough feature enough so that you could use Python to answer a question like "evaluate $(287,237,188 \times (487^3) + 6) \pmod{17}$" in less than a minute.

   c. Make sure you understand how to type basic Python commands into a `.py` file, and then run this *script* in your favorite way. **Do you use the command line (cmd, Bash, etc) with a text editor (notepad++, Atom, Sublime, Vim, etc.)? Do you use an IDE (PyCharm, Spyder, Emacs, etc.)?** I'm genuinely curious. We'll use a couple of different methods this term, but you'll often be able to use whatever you want.

   d. Read up on the print() function in Python. You'll use it a ton, and it's really capable.

3. Read up on Python white space, comments, and the format for basic "flow control" commands (that is, `if` statements, `for` and `while` loops, and defining functions using `def` and `return`). To test your understanding, look at the comments in `flow_control.py` and **make everything work**. You will know it all works when the code runs without raising exceptions, especially the `AssertionError` from the various `assert` statements in the script. As an answer to this question, **submit your fixed version** of `flow_control.py` with comments describing all fixes you made.

4. **Write a script** that asks a user for a positive integer n, then outputs the *Collatz sequence* for that number n:

   a. If the number $n = 1$, you're done.

   b. If the number n is even, divide it by 2 to get the next number in the sequence: $n \to \frac{n}{2}$.

   c. If the number n is an odd number bigger than 1, multiply by 3 and add 1 to get the next number in the sequence: $n \to 3n + 1$.

   You may assume the user correctly entered a positive integer (Though feel free to implement some checks for that!).

5. Read up on basic string manipulation in Python, for example the effect of `+`, `*`, `in`, `index`, and selecting/*slicing* substrings with `[ ]`.

   a. **Write some Python functions** that perform the following tasks from first principles. What this means is: if there's a function (in the standard library or elsewhere) which does this for you, the code I'm looking for is not simply calling this function. Don't worry, that will not be the rule for the whole term. I believe in using other people's code almost always, but not at the very beginning.

      i. `replace(str,sub,new)`: given three strings, replace all instances of `sub` in `str` with `new`.

      ii. `count(str,sub)`: given two strings, count the number of occurences of `sub` in `str`.

      iii. `longest(str)`: given a string, return a tuple containing the *slice indices* of the (first) longest word in the string. Assume the string is just a sentence or fragment with only ending punctuation. For example: if `s = "Four score and seven years ago,"`, then `longest(s)` will return `(5,10)`, because the longest words have 5 letters, and the first one is `"score"`, which is equal to `s[5:10]` (That's called *slicing* the string).

      iv. `is_balanced(str)`: given a string, return `True` if any parentheses in the string are appropriately nested, and `False` otherwise. For example, `is_balanced("(8x + (46^3)) = 24")` returns `True`, but `is_balanced(")()(")` returns `False`. If you want an added challenge, add the functionality for braces `{}` and brackets `[]` (This is definitely not necessary for you to complete).

   b. Once you're done, go check out the Python standard library documentation on strings. **Can you replace your elementary code completed above with any functions?**

6. One excellent resource when learning a new programming language is Project Euler. PE is a website containing hundreds of coding challenges, from little easy ones to pretty substantial, challenging ones. For this problem:

   - go check out Project Euler, make an account if you want, and read up on lists in Python.
   - **Complete problem 8 on PE**. I am anticipating you use lists for this, so you can just copy the big block of numbers and put them hard-coded into your file (with commas in between), or do something a bit more sophisticated (we'll do it on later problems).

7. Now let's take a look at sorting.

   a. **Write a function** that sorts a list using your favorite algorithm. Assume that the elements of the list can all be compared using `<`.

   b. So far, we haven't had to `import` anything. Take a look at the standard `math` module, and make sure you're comfortable with the following two ways of importing libraries:

   ```
   # method one:
   import math
   math.sqrt(2)

   # method two:
   from math import sqrt
   sqrt(2)
   ```

   c. Mercifully, Python has a built-in sorting algorithm, which can be called in two different ways: `my_list.sort()` and `sorted(my_list)`. **What are the differences between these two methods; why would you use one over the other? Write a script** that times your function against the standard sort on multiple types of data, including but not limited to: "random" numbers generated by slamming your fingers on the keyboard, and random numbers generated by the `random` module (check out `random.randint()`). To time functions, check out `time.time()` or the more sophisticated `timeit` module. Then: read up on the `key` argument to the built-in sort methods, and **explain it using english sentences and a some examples**.

8. Below is a trio of lists, each containing 19 or 20 integers. I've again formatted them to ease copying them into your Python code, if that's how you want to access them.

$$a = [11, 16, 14, 19, 1, 13, 15, 15, 2, 6, 4, 20, 17, 8, 18, 22, 25, 11, 18, -7]$$
$$b = [17, 15, 7, 12, 5, 20, 18, 22, 11, 2, 9, 0, 10, 11, 6, 17, 9, 10, 6]$$
$$c = [6, 16, 1, 6, 14, 5, 5, 15, 6, 11, 8, 15, 10, 3, 15, 10, 5, 14, 17, 13]$$

In this problem, we'll be performing some basic statistics on these lists.

a. Given a list of numbers $a = [a_1, a_2, \ldots, a_n]$, the *mean*, of a is defined by

$$\overline{a} = \frac{1}{n} \sum_{i=1}^{n} a_i.$$

The *median* of a is the element in the center of the ordered list (or the mean of the two central elements if it is an even-length list). The *mode* is the element that occurs most often in the list. Note that the mode need not be unique! **Write a script** that computes the mean, median, and mode of all three lists.

b. Adding lists in Python concatenates them: d = a + b + c. **Compute** the *standard deviation* of `d`:

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(d_i - \overline{d}\right)^2}.$$

9. Read up how to perform input/output of text files in Python. The important function is `open()`, which returns a `TextIOWrapper` object (a "file" object), which has some useful methods like `readlines()`, `write()`, and `close()`.

a. **Write a script** that can open a text file and look through each line of text to discover consecutively doubled words. It should write the results that it discovers to an output file. For example, if line 15 of my text is `"The cat jumped over the the chair."`, then `the` occurs twice consecutively, beginning at character 20. Thus the output file should have a line like

```
15, the, 20
```

or some format that you think makes sense. Some helpful methods are `string.split()` and `string.lower()`.

b. **Modify your script** so that the filename can be specified by command-line argument, so that I can call it by typing `python find_double_words.py my_text_file.txt`

c. **Extend your script** to allow for wrapping around lines and to take multiple files.

10. The lines in `emails.txt` is a collection of lines of the form

    ```
    From: First Last <email> To: First Last <email> Date: Sat, Jan  5 09:14:16 2008
    ```

    Please note: This is a contrived list. I randomly generated realistic-looking emails from a list of the 1000 or so most common first names and last names, and a few domains. If the email address of someone you know happens to appear on this list, please don't fret: it was randomly generated, and not the beginning of a real email.

    a. Read up on Python dictionaries, or dicts. **Write a script** that reads in the lines of the file and splits the important elements into a dictionary (one dictionary for each line).

    b. Using a dictionary, **determine which day of the week is the most commonly occurring**.

    c. **What's the least common email domain?** (That's the portion of the email address that occurs after the `@` symbol.)

    d. **Who sent the most emails?**

11. Take a look at `tictactoe_games.txt`. It contains 200 Tic Tac Toe games, some of which are completed, some of which are not. Note that a game of Tic Tac Toe doesn't need to be filled to be completed. To help you with the tasks below, I have included `TicTacToe.py`, which is a skeleton of the code you will need to write. Note that the script and the class can be written in the same code, but they don't need to be. Be sure to read through both the skeleton and some resources on Python classes to ensure you understand the basics of how to build them.

    a. **Create a Python class** called `TicTacToe` which is constructed from a single game's string. It should have an attribute called `game_id` (corresponding to the id in the file) and can parse the string into another attribute called `board` which is a 2d array containing the board state.

    b. **Add a method** (a function bound to the class) which is able to determine if the game is over, and if so, who is the victor. (Note that games could have ended in a tie.)

    c. **Add a method** which is able to determine whose turn it is, if possible. It should be called only if a game is not complete. (Note that it may not be possible to know the turn from the board alone.)

    d. **Write a script** which reads in `tictactoe_games.txt`, and uses your class to create a new file `tictactoe_games_out.txt`, containing the information determined from your class, formatted as the following examples demonstrate (these examples were not taken from the file):

    ```
    game_id: 016 complete: False Turn: O
    game_id: 021 complete: True Victor: X
    game_id: 132 complete: False Turn: -      if it is not possible to determine the turn
    game_id: 195 complete: True Victor: T     if the game ended in a tie
    ```

12. For this problem, you'll need to use the internet.

    a. In this problem, we'll be using an external library called "Requests", which you should read about over at `http://docs.python-requests.org/en/latest/user/quickstart/` (However, those unfamiliar with HTTP requests will find that a bit daunting. Don't worry! We'll learn about them later in this course). Make sure you at least read the sections "Make a Request" and "Response Content", and browse through the rest. If you installed Python via Anaconda, you already have the Requests library required for this exercise. Whether or not you need to install it, read up on the (command line) command `pip`, and how it's used. The most important two subcommands to understand are `pip install <library_name>` and `pip list`.

    b. **In the Python interpreter**, use a `GET` request to request the content of the PA wikipedia page: `https://en.wikipedia.org/wiki/Phillips_Academy`
    Now that you have it saved to a variable (I'll call it `resp`), let's inspect it. Whenever you want to learn about an object in Python that you're unfamiliar with, you can inspect the object using the keyword `dir`. `dir(resp)` is a "a list of valid attributes for that object". This should give you all methods and local variables for the object. There's a lot going on in there:

    - The double-underscored methods are called *magic* methods, and they are ignorable for now.
    - The methods whose name begins with a single underscore are *private* methods, and they are ignorable too.
    - The rest of the items may be of use to us.

    Construct a list containing only the non-magic, non-private names. What's the remaining list? Which one contains the html code from the website we requested? What's the status code for this request? What is a status code?

    c. **Create a simple "Webcrawler"**: take the html and extract from it all other urls. (If you're not familiar with html, don't worry; we'll spend a bit of time figuring it out this term. Just follow along with these instructions.) One way to find a url is to find a substring that starts with `href="` and then proceeds until it ends with a quotation mark `"`. A real webcrawler would then process those links to determine if they should be followed, then open the relevant ones and repeat. (*Sidenote*: this is not at all the "right" way to do this, we'll learn much better ways soon. I know it's messy, it's supposed to be.)

    d. After that's done, determine the following: are there more links to wikipedia pages or non-wikipedia pages? We'll use the following gross simplifications:

    - If a url begins with a single slash `/` (and *not* two slashes), or a hash `#`, or it contains the word `wiki` in it, then it is a wikipedia page.
    - Otherwise, it is a non-wikipedia page. This is almost certainly false in general, but it for our purposes it's fine.