Exercises

Level 1: Python Syntax 101

1.1: Variables/Conditionals

1) Create a program that outputs ‘Hello World!’ on screen.

2) Do the same as exercise 1), but the caveat here is that each word should appear on a different line.

3) Do the same as exercise 1), but the caveat here is that each word should be separated with a tab**.**

4) Create a program that takes input from the user (using the **input** function), and stores it in a variable.

5) Extend the program from 4) to display the type of the variable that contains the value that the user entered.

6) Extend the program from 5) and create an if statement that does the following:

   a. If the inputted value is a *float* or *int*, then output ‘The inputted value is a number’. Additionally, either output ‘the inputted number is less than 65’, ‘the inputted number is between 64 and 65’, or ‘the inputted number is greater than 64’, depending on the number.

   b. Otherwise, if the inputted value is a string, then output ‘The inputted value is a string’.

   c. Otherwise, if the inputted value is neither a number nor a string, then output ‘The inputted value is neither a number nor a string’.

7) Create a program that takes two inputs from the user (using **input**): The base and height of a triangle. Output the area of the triangle. Be sure to have if statements which check that the input types and values are valid for the sides of a triangle (if not, print an error message to the user).

8) Create a program that takes two inputs from the user (using **raw_input**): The base and height of a triangle. Output should be the area of the triangle. As **raw_input** returns a string in all cases, you’ll need to convert it to a number using **float**. Be sure to have if statements which check that the input values are valid for the sides of a triangle (if not, print an error message to the user).

1.2: Lists/Loops

Looping/Lists

1) Write a program that does the following:
   - **a.** Keeps asking the user for a number, until the user enters the letter *s*.
   - **b.** Once the user finishes entering numbers, calculate and display the average of the numbers. You should do this using a loop.

2) Create a list of ten numbers. Should contain some integers and some decimals. Perform the following operations:
   - **a.** Display the first two numbers from the list (using indexing).
   - **b.** Display the last two numbers.
   - **c.** Display all the numbers besides the last number, using a single print statement.
   - **d.** Display all the numbers besides the first number, using a single print statement.
   - **e.** Display all the numbers besides the first two and last three numbers, using a single print.
   - **f.** Append one number to the end of the list.
   - **g.** Append five numbers to the end of the list, using a single operation.
   - **h.** Insert one number right after the third number in the list.
   - **i.** Modify the fourth-to-last number in the list.
   - **j.** Display the list backwards, using splicing.
   - **k.** Display every second item in the list.
   - **l.** Display every second item in the list, backwards.

3) Create a list of all even integers 1-1000. Write a loop that prints all numbers in the above list, separated by commas.

4) Create a list of all odd integers 1-1000. Write a loop that prints all numbers in the above list, separated by commas.

5) Create an aggregate list from 3) and 4) and print it out, separated by commas.

6) Print out the aggregate list from 5), backwards and separated by commas.

List Comprehensions

7) Write a list comprehension that results in a list of the *squares* of all numbers 0 through 100:
   - **a.** Filter the resulting list, to create another list that only contains numbers greater than 1000.
   - **b.** Filter further, to create another list that only contains even numbers (hint: use the Modulus operator).

8) Create two lists: The first list should be called 'players', and contain at least ten unique names. The second list should be called 'injured_players', and contain a few names of players from the first list. Create a list comprehension which results in a list containing all active (non-injured) players.

9) Create a list of names, and a second list of ages which correspond to a name in the first list:

- **a.** Zip them together and print the result.
- **b.** Using a list comprehension, create a list that contains all the names for which the corresponding age is greater than or equals to 18. (Hint: Use **zip** as necessary. Can you also do this without **zip**? Which is better?).

10) Write a list comprehension that results in a list of all numbers 0 through 10,000,000.

- **a.** Using a loop, filter the resulting list, to create another list that only contains numbers ending with the digit 0.
- **b.** Do the same as **a)** using a list comprehension.

  Use the **time**.**time** function to capture the time taken for each version. Which is quicker? Why?

11) Create a list of lists of any type. Use the double list-comprehension syntax, as described in the lecture, to create a flattened single list.

Note that this can be useful in situations where one has a function that returns a list of items, and calls the function many times, resulting in a large list of lists (which can then be flattened, for simplicity).

1.3: Functions

1) Write a function that can print out the day of the week for a given number. I.e. Sunday is 1, Monday is 2, etc. It should return a tuple of the original number and the corresponding name of the day.

2) Write a function that returns the Fibonacci sequence as a list. The function should take a parameter called N and return the entire sequence of Fibonaccis, from 0-N. You may use either iterative or recursive programming techniques (bonus if you implement both!).

Note that we will generalize the above, to allow it to return an infinite sequence, in Level 3, where we discuss Python generators.

3) Create a function that calculates the mean of a passed-in list.

4) Create a function that calculates the variance of a passed-in list. This function should delegate to the mean function (this means that it calls the mean function instead of containing logic to calculate the mean itself, since mean is one of the steps to calculating variance).

5) The previous exercise did not mention one crucial aspect of calculating the variance: Is it referring to sample or population variance? Population variance has zero *degrees of freedom* whereas sample variance has one *degree of freedom*. To this end, re-implement the **variance** function with an additional parameter called **degOfFreedom**, which is used by the function to determine how to calculate the variance. This parameter should default to 1 (sample variance).

Test the newly implemented variance function by setting **degOfFreedom** to 0 and 1, with and without using it as a keyword argument.

6) Create an alternative **mean** function to use ***args** instead of a taking a list of numbers. It should be invoked as follows (for example):

argsMean(1.3, 4.5, 6.7, 11.2, 100, 987.6)

Test the function with variable numbers of arguments. It's also possible to pass a list or tuple into this version of the function by using the * operator – you should attempt this as well.

7) Write a function that takes name, age as parameters. It should also take ****kwargs**. The function should display the name, age, and any of 'state', 'height', and 'weight' that happen to exist in the kwargs. Call the function with names, ages, and different combinations of keyword arguments (state, height, weight, hairColor, etc.).

8) Extend the program from 8) to display all passed-in keyword arguments, no matter what the key is.

1.4: Built-in Functions

1) You are a lender, holding onto a large number of mortgages. Create code that does the following:
   - **a.** A function that returns an unsorted list of mortgage amounts, in thousands. Numbers should range from 100 to 1,000 and do not need to all be unique.
   - **b.** Filter the result of a) into three lists: Amounts below 200, amounts between 200 and 467, and amounts greater than 467. Call these 'miniMortgages', 'standardMortgages', and 'jumboMortgages' respectively.
   - **c.** Use the *all* function with an *if* statement to verify that the resulting lists of b) indeed contain only numbers within the specified ranges.
   - **d.** Use the *any* function with an *if* statement to verify that the resulting lists of b) indeed contain only numbers within the specified ranges.

2) Find the length of each list in part **b** of the previous exercise. Then, verify that the lengths of all three lists indeed add up to the length of the full list in part **a**.

3) Sum the full list of mortgages, to obtain the total amount owed to your firm.

4) Find the minimum and maximum mortgage amount owed, for each mortgage sub-list.

5) Create any code that demonstrates usage of the **abs** function.

1.5: Dicts and Sets

Sets

1) Port Exercise 1.2.8 to use sets instead of lists. What's the benefit?

2) Create two sets: Set 1 should contain the twenty most common male first names in the United States and Set 2 should contain the twenty most common male first names in Britain (Google it). Perform the following:

- **a.** Find the first names that appear in both sets.
- **b.** Find the first names that appear in the United States set, but not Britain.
- **c.** Find the first names that appear in the Britain set, but not United States.

3) Create a list of unique mortgage amounts.

4) Create a set of mortgage terms, in years (10, 15, 30):

- **a.** Add a 5-year term to the set.
- **b.** Remove the 15-year term from the set.
- **c.** Remove a 45-year term from the set. What happens? How can you prevent that?

Dictionaries

5) Create a simple dictionary that has country name as the key and population as the value (*country:population*). Do this for at least ten countries. Then, do the following:

- **a.** Create code that prompts the user for the name of a country ('0' to exit). Display the population for that country. Repeat until the user enters '0'.
  If the country does not exist in the dict:
  - i. Display a message to the user that the population is unknown and prompt the user to enter the population.
  - ii. Update the dict with the value provided by the user.
- **b.** Display the final dict once the user exits the loop. Display should be in the format:
  Country 1 has population X
  Country 2 has population Y
- **c.** Note that the above display will not necessarily be sorted. Modify the code from part **b** to display sorted by 1) country then 2) population, largest first (Hint: Use the **sorted** function).

6) Create the mode function, building off the frequency function that was demonstrated in the lecture. The function should return a tuple, containing a list of mode values (containing one or more items) and their frequency. Be sure to fully test it.

The below exercises will build off the mortgage function created in **Exercise 1.4.1**.

7) Extend the mortgage function to return a dict of *address:mortgage*. For simplicity, address should be a unique six-character string. For example **{'867E23' : 120}**. Once you have done this, modify the filtering code from 1.4.1 to do the following:

   a. Provide three separate *dicts*, filtered the same way as problem 1).

   b. Modify one value in the *jumboMortgages* dict. Check the original dict; did it remain intact or change? Why?

   c. Extract the lists of amounts from each separate dict. Modify one value in the *miniMortgages* list. Does the *miniMortgages* dict change? How about the original dict? Why?

8) Modify the dict from 7) to be a dict of address keys and the following tuple as value: (amount, rate, term in months). Amount should now be between 100,000 and 1,000,000. Once you have done this, do the following:

   a. Extract a list of tuple values from the dict, and sort the list by amount (descending).

   b. Create a function that calculates the Weighted Average Rate of the mortgage pool. The input parameter should be a list of mortgage tuples (amount,rate,term). Print the rate percentage, rounded to the nearest hundredths.

   c. Create a function that calculates the Weighted Average Maturity (term) of the mortgage pool. The input parameter should be a list of mortgage tuples (amount,rate,term).

   d. Create a new dict (by processing the original dict) with Term as the key and a list of (amount, rate) tuples for each Term key.

1.6: Packages

1) Import the *math* module and demonstrate usage of many of its built-in functions.

2) Create a program with the following structure:

   a. Create a base folder, with the name of your program (can be anything). Note that PyCharm will give you this base folder automatically.

   b. Create a couple subfolders, each of which should contain two subfolders, each of which should contain one subfolder. Create __init__.py for each.

   c. Create one or more Python scripts in each subfolder, at each level. Each script should have a unique function (you may reuse previously-created functions and/or create new functions). Be sure to name the scripts aptly.

   d. Go back and aptly rename each subfolder (based on their content).

   e. Create a main.py, in the base folder, which contains a **main** function. This module should import all the other created modules and demonstrate their functionality. You should use a mix of full-module imports and selective importing.