1 Programming in Python

1.1 Git Bash & Workflow

We will use BASH as our command line interface. Note that BASH is the default shell on Max OS X (so we just use the terminal). On Windows, we will use Git Bash as our shell. We can do the following basics commands:

· **ls** - lists the files and folders (also known as directories) inside the current directory.
· **pwd** - this prints the working directory that you are currently in.
· **cd** - allows us to change directories, takes argument of desired directory (.. moves previous directory).
· **mkdir** - this makes a new directory in the current one, takes argument of new directory name.
· **touch** - this creates a new file in the working directory, takes argument of new file name.
· **echo** - this lets us add text to a specified file, for example: echo "Testing" >> test.txt
· **cat** - this lets us print the contents of a specified file to the terminal, for example: cat test.txt

A *filesystem* organizes the computer's files and directories into a tree structure.
Note: Using the 'up arrow' on the keyboard will allow you to cycle through previous commands.

We can use Git to keep track of changes made to a project over time. A Git project can be thought of having the following workflow:

1) *Working Directory* - where you do all the work (creating, editing, deleting, organizing).
2) *Staging Area* - where you list changes made to working directory (ready to commit).
3) *Repository* - where Git stores changes as different version of the project.

We can use the following commands in our Git project:

· **git init** - this will initialize an empty Git repository in your current work directory.
· **git status** - this will show status of changes (changes to be committed and untracked files).
· **git add** - this will add a file to staging area, pass a parameter of the filename.
· **git diff** - shows us the lines added since our last 'git add', pass filename parameter (marked by +).
· **git commit -m " "** - permanently stores changes from staging area (pass message in " ").
· **git log** - this lets you refer back to earlier versions of a project (store chronologically).

1.2 Lists

We can use **zip()** to create pairs from multiple lists. However, it returns the location in memory and must be converted back to a **list()** in order to print it. We can add a single element to a list using **.append()**, which will place at the end of the list. We can add multiple lists together by using + to concatenate them.

```
last_semester_gradebook = [("politics", 80), ("latin", 96), ("dance", 97),
("architecture", 65)]

subjects = ["physics", "calculus", "poetry", "history"]
grades = [98, 97, 85, 88]
subjects.append("computer science")
grades.append(100)
gradebook = list(zip(subjects,grades)) # combine and cast as a list
gradebook.append(("visual arts", 93)) # append a tuple
print(gradebook)

full_gradebook = gradebook + last_semester_gradebook
print(full_gradebook)
```

We can create an array of integers for a given size by **range()**, which generates starting at a point (0 by default) to the (input value - 1). However, you must convert it to a list since it returns on object.

```
my_list = range(9) # values 0 to 8
my_list_2 = range(5, 15, 3) # start at 5, end at 14, increment by 3
print(list(my_list_2)) # [5, 8, 11, 14]
```

We can select a section of a list by using syntax array[start:stop], called **slicing**.

```
suitcase = ['shirt', 'shirt', 'pants', 'pants', 'pajamas', 'books']
start = suitcase[:3] # same as suitcase[0:3]
end = suitcase[-2:] # gets last 2 elements of suitcase
```

We can count how many times an element appears in a list with **.count()**

```
votes = ['Jake', 'Jake', 'Laurie', 'Laurie', 'Laurie', 'Jake']
jake_votes = votes.count('Jake')
print(jake_votes)
```

We can sort a list alphabetically or numerically with **.sort()** - only alters a list, doesn't return a value
We can use **sorted()** to also sort a list, but it will not affect the original list (returns sorted copy)

```
games = ['Portal', 'Minecraft', 'Pacman', 'Tetris', 'The Sims', 'Pokemon']

games_sorted = sorted(games)
print(games) # in same order as above
print(games_sorted) # new list of sorted games

games.sort()
print(games) # now the games list is also sorted
```

Tuples are immutable (can't change any values after creating) and are denoted with ( )
We use tuples to store data that belongs together and don't need order or size to change

```
my_info = ('Derek', 22, 'Student')
name, age, occupation = my_info # will assign each value to a varaible

one_element_tuple = (4,) # NOTE: we need the , after 4 otherwise it wont be a tuple
one_element_tuple_2 = (4) # same as one_element_tuple_2 = 4
```

1.3 Loops

We can use **for** loops to iterate through each item in a list, with the following general formula
- We can use range() to execute a for loop from start (0 by default) to stop (n-1)
- We can use *break* to exit a for loop when a certain value is found
- We can use *continue* to move to the next index in a list if a condition is found

If we have a list made of multiple lists, we use **nested** loops to iterate through them

```
sales_data = [[12, 17, 22], [2, 10, 3], [5, 12, 13]]
scoops_sold = 0

for location in sales_data: # for each list in list
for sales in location: # for each element in inner list
scoops_sold += sales

print(scoops_sold)
```

We can use **list comprehension** to efficiently iterate through a list instead of a for loop
We can also use this to alter values in a list and create a new list

```
heights = [161, 164, 156, 144, 158, 170, 163, 163, 157] # in cm's

can_ride_coaster = [cm for cm in heights if cm > 161]
print(can_ride_coaster) # [164, 170, 163, 163]

celsius = [0, 10, 15, 32, -5, 27, 3] # degrees in C

fahrenheit = [f_temp * (9/5) + 32 for f_temp in celsius] # convert C to F degrees
print(fahrenheit) # [32.0, 50.0, 59.0, 89.6, 23.0, 80.6, 37.4]
```

1.4 List Comprehension / Lambda Functions

We can iterate through lists within lists with the following syntax

```
nested_lists = [[4, 8], [15, 16], [23, 42]]

product = [(val1 * val2) for (val1, val2) in nested_lists]
print(product) # [32, 240, 966]

greater_than = [ (val1 > val2) for (val1, val2) in nested_lists]
print(greater_than) # [False, False, False]
```

We can iterate through two lists in one list comprehension by using the zip() function.

```
x_values_1 = [2*index for index in range(5)] # [0.0, 2.0, 4.0, 6.0, 8.0]
x_values_2 = [2*index + 0.8 for index in range(5)] # [0.8, 2.8, 4.8, 6.8, 8.8]

x_values_midpoints = [(x1 + x2)/2.0 for (x1, x2) in zip(x_values_1, x_values_2)]
# [0.4, 2.4, 4.4, 6.4, 8.4]

names = ["Jon", "Arya", "Ned"]
ages = [14, 9, 35]

users = ["Name: " + n + ", Age: " + str(a) for (n,a) in zip(names,ages)]
print(users) # ['Name: Jon, Age: 14', 'Name: Arya, Age: 9', 'Name: Ned, Age: 35']
```

1.5 Python Objects

1.5.1 Strings