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:

- **python3** - passing a .py file to this command with compile and run a Python file.
- **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 +.

```python
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']
```

See "Recommendation Engine (Beginner)" in *Python Projects* folder for final project (sections 1-4).

1.5 Python Objects

1.5.1 Strings and their Methods

We can **slice** a string from a starting index (inclusive) to an ending index (exclusive). We can also have *open-ended selections*, where removing the starting index starts at the beginning and removing the ending index goes to the end of the string.

```
first_name = "Julie"
last_name = "Blevins"

new_account = last_name[:5] # first 5 letters
temp_password = last_name[2:6] # 3rd through 6th letter

def account_generator(first_name, last_name):
  user = first_name[:3] + last_name[:3]
  return user # combines first 3 letters of first name and last name
```

We can use **negative indices** the slice strings backwards (starting at -1 instead of 0).

```python
first_name = "Julie"
last_name = "Blevins"

def password_generator(first_name, last_name):
  password = first_name[-3:] + last_name[-3:]
  return password # combine last 3 letters of first name and last name

temp_password = password_generator(first_name, last_name)
print(temp_password) #lieins
```

Strings are **immutable**, meaning that they cannot be chance, so we must make a new string if we wish to alter any characters in a string.

· Note that we can include special characters that would end our string by using \ (escape character).

```python
first_name = "Bob"
last_name = "Daily"

fixed_first_name = 'R' + first_name[1:] # concatinate 'R' with 'ob'
print(fixed_first_name) # Rob
```

We can use the **in** comparison to see if one string is part of another string (returns a Boolean).

```python
def common_letters(string_one, string_two):
  ans = [ ]
  for x in string_one: # loop through all characters in string one
    if x in string_two and x not in ans: # see if char in string two AND not in ans
      ans.append(x) # if true, add char to list (only unique chars)
  return ans
```

There are many **string methods** that we can use to change our strings. It is important to note that string methods only create NEW strings and do not change the original. Some methods include:

1) We can change the casing of a string with: *.lower( ), .upper( ), .title( )*
2) We can separate a string into a list of sub-strings with *.split( )* and passing the char to split on.
   · note: we can split on newline ('\n') and tab ('\t') characters, not uncommon to see in data.

```python
authors = "Audre Lorde, William Carlos Williams, Gabriela Mistral, Jean Toomer,
An Qi, Walt Whitman, Shel Silverstein, Carmen Boullosa, Kamala Suraiyya"

author_names = authors.split(',') # split on , and create list of names
author_last_names = [x.split()[-1] for x in author_names]
# above will split names on space, index from the end and take only the last name
```

3) We can join string back together using *.join( )* and passing it a list of strings and a given delimiter.
   · note: common to join on ',' to create CSV's, can also join on '\n' or '\t'

```python
reapers_line_one_words = ["Black", "reapers", "with", "the", "sound", "of", "steel"]

reapers_line_one = ' '.join(reapers_line_one_words) # join each word on a space
print(reapers_line_one) # Black reapers with the sound of steel
```

4) We can clean strings of extra spaces or unwanted characters (pass as argument) by using *.strip( )*

```python
love_maybe_lines = ['Always    ', '     in the middle of our bloodiest battles  ',
                    'you lay down your arms', '           like flowering mines    ',
                    '\n' ,'   to conquer me home.    ']

love_maybe_stripped = [x.strip() for x in love_maybe_lines] # strip all list items
love_maybe_full = '\n'.join(love_maybe_stripped) # join each string on newline
```

5) We can replace all instances in a string of the first argument with the second by using *.replace( )*

```
toomer_bio =" Nathan P. Tomer, who adopted the name Jean Tomer early on..."
toomer_fixed = toomer_bio.replace('Tomer', 'Toomer')
print(toomer_fixed) # Nathan P. Toomer, who adopted the name Jean Toomer early on.
```

6) We can locate the index of a string inside of a string by passing the argument to *.find()*
   · note: when searching for multiple characters in a string, it will return the first index value.

```
god_wills_it_line_one = "The very earth will disown you"

disown_placement = god_wills_it_line_one.find('disown') # returns 20
```

7) We can include varaible in a string with {} and passing the variables to *.format()*
   · note: you can pass keywords in {} that can be referenced in .format() in any order (easy to read).

```
def poem_title_card(poet, title):
  return "The poem \"{}\" is written by {}.".format(title, poet)

def poem_description(publishing_date, author, title, original_work):
  poem_desc = "The poem {title} by {author} was originally published in
              {original_work} in {publishing_date}.".format(publishing_date=
              publishing_date, author=author, title=title, original_work=
              original_work)
  return poem_desc

my_beard = poem_description("1974", "Shel Silverstein", "My Beard", "Where the
                            Sidewalk Ends")
print(my_beard) # The poem My Beard by Shel Silverstein was originally published in
                # Where the Sidewalk Ends in 1974.
```

1.5.2 Datetime Module

The **datetime** module allows use to create a python object that represents a point in time.

· We can use *strptime()* to parse a string and extract the date/time from it by passing the original date string as the first argument, and the formatted date string code as the second argument.

· We can use *strftime()* to grab specific parts out of a datetime object and put them into a string. The first argument is the datetime we want to format as a string, the second argument is formatted date string code (above link).

```
from datetime import datetime

birthday = datetime(1997, 2, 15, 4, 25, 12) # 2-15-1997 at 4:25:12 AM (datetime obj)
# access by .year, .month, .day, .hour, .min, .sec
birthday.weekday # returns 1, formatted 0-6 (mon-sun)

datetime.now() # returns current date and time when code is executed
datetime.now() - datetime(2017, 1, 1) # we can subtract datetime objects (no +, *, /)
# returns datetime.timedelta(days, seconds, microseconds)

parsed_date = datetime.strptime('Jan 15, 2018', '%b %d, %Y')
print(parsed_date) # 2018-01-15 00:00:00

date_string = datetime.strftime(datetime.now(), '%b %d, %Y')
print(date_string) # 'Apr 11, 2020'
```

1.5.3 Dictionaries

A **dictionary** is an unordered set of *key: value* pairs that are enclosed by { } and separated by a comma. The values within a list can be strings, numbers, lists, or even another dictionary. However, keys must always be unchangeable/hashable data types like numbers or strings.

- We can add key: value pairs int a dictionary with the syntax: my_dict[new_key] = "new_value"
- We can add multiple key value pairs to a dictionary with the .update( ) method.
- note: entering a key that is already in the list will replace the old value with the new one.

```python
user_ids = {} # empty dictionary
user_ids["proCoder"] = 119238 # add new key:value pair
user_ids.update({'theLooper': 138475, 'stringKing': 85730}) # add multiple key:value

print(user_ids)
# {'proCoder': 119238, 'theLooper': 138475, 'stringKing': 85730}
```

We can use **list comprehension** to combine two lists into a single dictionary with the following syntax: *dictName = {key:value for key, value in zip(list1, list2)}* where list1 are the keys and list2 are the values.

```python
drinks = ["espresso", "chai", "decaf", "drip"]
caffeine = [64, 40, 0, 120]

zipped_drinks = zip(drinks, caffeine)
drinks_to_caffeine = {key:value for key, value in zipped_drinks}
print(drinks_to_caffeine) # {'espresso': 64, 'chai': 40, 'decaf': 0, 'drip': 120}
```

We can **access value's** by calling the dictionary with the key passed to it. Note that if we try to call a key that isn't in our dictionary, we will get a *KeyError* (we can use a try... except method). However, this isn't the best method. We should use the *.get( )* method and pass the key and a value to output if it is not found (default is 'None')

```python
# using drinks_to_caffeine from above
print(drinks_to_caffeine['espresso']) # 64

try:
  print(caffeine_level["matcha"]) # won't print, throws KeyError
except KeyError:
  print("Unknown Caffeine Level") # this will be outputted

print(caffeine_level.get("matcha")) # None
print(caffeine_level.get("chai", 'Does Not Exist')) # 40
```

We can **delete a key** and return its value by using the *.pop( )* method, passing the key and a value to return if it does not exist in the dictionary (similar to .get()).

```python
available_items = {"strength sandwich": 25, "stamina grains": 15, "power stew": 30}
health_points = 20

health_points += available_items.pop("stamina grains", 0)
print(available_items) # {'strength sandwich': 25, 'power stew': 30}
print(health_points) # 35
```

We can get **all keys** in a dictionary by using list( ) function to print out all keys in a dictionary, or the .keys() method to return a dictionary object that contains all the keys in a given dictionary.

```python
user_ids = {"teraCoder": 100019, "pythonGuy": 182921, "samTheJavaMaam": 123112}
num_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22}

print(list(user_ids)) # ['teraCoder', 'pythonGuy', 'samTheJavaMaam']
lessons = num_exercises.keys()
print(lessons) # dict_keys(['functions', 'syntax', 'control flow', 'loops'])
```

We can get **all values** of a dictionary by using the .values( ) method to return a dict_list object.

```
num_exercises = {"functions": 10, "syntax": 13, "control flow": 15, "loops": 22,
                 "lists": 19, "classes": 18, "dictionaries": 18}

total_exercises = 0
for ex in num_exercises.values(): # dict_values([10, 13, 15, 22, 19, 18, 18])
  total_exercises += ex

print(total_exercises) # 115
```

We can get **all items** (both keys and values) with the .items() method, which will return a dict_list object made of tuples consisting of (key, value).

```
pct_women_in_occupation = {"CEO": 28, "Engineering Manager": 9, "Pharmacist": 58,
                           "Physician": 40, "Lawyer": 37, "Aerospace Engineer": 9}

for key,value in pct_women_in_occupation.items():
  print("Women make up {} percent of {}s.".format(value, str(key)))
  # example output: Women make up 28 percent of CEOs.
```

1.5.4 Classes

A **class** is a template for a data type. A **class variable** is a variable that's the same for every instance of the class (and we can access it from an class object we create). We can create **methods** in our classes, with the first argument always being *self* (which refers to the object we create and can access any class variables) and any other variables we want to pass.

```
class Circle:
  pi = 3.14
  def area(self, radius):
    area = self.pi * (radius**2)
    return area

circle = Circle() # create instance of class object
round_room_area = circle.area(5730) # 103095306.0
```

We can create a **constructor** method in Python by using a dunder method (double underscore), which is called every time we create an object from the class (we can also pass multiple parameters to this).

```
class Circle:
  def __init__(self, diameter): # create constructor with 1 parameter
    print("New circle with diameter: {}".format(diameter))

teaching_table = Circle(36) # New circle with diameter: 36
```

The data held by an object is referred to as an **instance variable**. Instance variables aren't shared by all instances of a class, they are variables that are specific to the object they are attached to (also known as instance attributes, they are accessed the same way as class variables).

```
class Store:
  pass

alternative_rocks, isabelles_ices = Store(), Store()

alternative_rocks.store_name = "Alternative Rocks" # instance attribute .store_name
isabelles_ices.store_name = "Isabelle's Ices" # instance attribute .store_name
```

If we want to see **whether or not a class has an attribute**, we can use the following two functions. The *hasattr( )* function gets passed the class and the attribute name as a string, and either returns T/F. The *getattr( )* function takes the same parameters as hasattr( ) but we can pass a third parameter that will be return if it does not find the attribute (default is AttributeError).

```
how_many_s = [{'s': False}, "sassafrass", 18, ["a", "c", "s", "d", "s"]]

for x in how_many_s: # for each element in list
  if hasattr(x, 'count'): # see if it has the attribute 'count' (T/F)
    print(x.count('s')) # if true, print the number of s's
```

We can create **instance variables with self** for each object that we create. Each object can call the class methods, but can have different instance variables defined for them if we pass them through parameters. We can also use the **repr** method to tell Python what we want the string representation of the class to be (good for debugging).

```
class Circle:
  pi = 3.14 # class varaible
  def __init__(self, diameter):
    self.radius = diameter/2 # instance variable (different for each object)
  def circumference(self):
    return 2*(self.pi)*(self.radius)
  def __repr__(self): # string representation everytime we create an object
    return "Circle with radius {}".format(self.radius)

teaching_table = Circle(36)
round_room = Circle(11460)

print(teaching_table) # Circle with radius 18.0
print(teaching_table.circumference()) # 113.04
print(round_room.circumference()) # 35984.4
```

INHERITANCE & POLYMORPHISM

We can have our classes **inherit** from another class (parent to subclass). We can do this by passing the parent class as a parameter to the declaration of the subclass.

```
class Bin: # parent class
  pass

class RecyclingBin(Bin): # subclass
  pass
```

We can define our own **exceptions** by having our classes inherit from the Exception class. We can then use these to throws exceptions in methods of our subclasses.

```
class OutOfStock(Exception): # inherit from Exception class
  pass
class CandleShop:
  name = "Here's a Hot Tip: Buy Drip Candles"
  def __init__(self, stock):
    self.stock = stock

  def buy(self, color):
    if self.stock[color] == 0:
      raise OutOfStock # raise OutOfStock exception
    else:
      self.stock[color] = self.stock[color] - 1

candle_shop = CandleShop({'blue': 6, 'red': 2, 'green': 0})
candle_shop.buy('green') # __main__.OutOfStock error
```

We can **override methods** in our subclasses by creating a new definition in the subclass that is different from the parent class. We can also use the *super()* function to call a method from a parent class and add any other parameters we want to it.

```python
class PotatoSalad: # parent class
  def __init__(self, potatoes, celery, onions):
    self.potatoes = potatoes
    self.celery = celery
    self.onions = onions

class SpecialPotatoSalad(PotatoSalad): # subclass
  def __init__(self, potatoes, celery, onions): # subclass constructor
    super().__init__(potatoes, celery, onions) # call parent class constructor
    self.raisins = 40 # add new instance varaible for subclass
```

Polymorphism is the term used to describe the same syntax doing different actions depending on the type of data. Polymorphism is an abstract concept that covers a lot of ground, but defining class hierarchies that all implement the same interface is a way of introducing polymorphism to our code.

We can define **dunder methods** that define a custom-made class to look/behave like a Python builtin.

```python
class Atom:
  def __init__(self, label):
    self.label = label
  def __add__(self, other): # dunder method that lets us use + to combine objects
    return Molecule([self, other])
class Molecule:
  def __init__(self, atoms):
    if type(atoms) is list:
      self.atoms = atoms

sodium = Atom("Na")
chlorine = Atom("Cl")
salt = sodium + chlorine
```

1.6 Linear Data Structures

1.6.1 Nodes

Nodes are the fundamental building blocks of many computer science data structures. They form the basis for linked lists, stacks, queues, trees, and more. An individual node contains data and links to other nodes (often called **pointers**). The end of the node path is denoted by *null*.

· *Orphaned Node* - inadvertently removing the link to a node and losing any linked nodes data.

```python
class Node:
  def __init__(self, value, link_node=None): # if not pointer passed, then null
    self.value = value
    self.link_node = link_node
  def set_link_node(self, link_node): # set pointer of given node
    self.link_node = link_node
  def get_link_node(self): # get pointer of given node
    return self.link_node
  def get_value(self): # get value of given node
    return self.value

yacko = Node('likes to yak') # New node, no pointer defined
wacko = Node('has a penchant for hoarding snacks') # New node, no pointer defined
dot = Node('enjoys spending time in movie lots') # New node, no pointer defined
yacko.set_link_node(dot) # yacko points to dot
dot.set_link_node(wacko) # dot point to wacko
```

1.6.2 Linked Lists

A **linked list** is comprised of a series of nodes, the head node is the node at the beginning of the list. Each node contains data and a link (or pointer) to the next node in the list. The list is terminated when a node's link is null (this is called the tail node). Linked lists typically contain unidirectional links, but can also sometimes be bidirectional.

```python
class Node:
  def __init__(self, value, next_node=None):
    self.value = value
    self.next_node = next_node

  def get_value(self):
    return self.value

  def get_next_node(self):
    return self.next_node

  def set_next_node(self, next_node):
    self.next_node = next_node

class LinkedList:
  def __init__(self, value=None):
    self.head_node = Node(value)

  def get_head_node(self):
    return self.head_node

  def insert_beginning(self, new_value):
    new_node = Node(new_value) # create new node
    new_node.set_next_node(self.head_node) #  set pointer to current head
    self.head_node = new_node # set new node to head

  def stringify_list(self):
    string_list = ""
    current_node = self.get_head_node()
    while current_node:
      if current_node.get_value() != None:
        string_list += str(current_node.get_value()) + "\n"
      current_node = current_node.get_next_node()
    return string_list

  def remove_node(self, value_to_remove):
    current_node = self.head_node
    if current_node.get_value() == value_to_remove:
      self.head_node = current_node.next_node # remove head if value found there
    else:
      while current_node: # while not None
        if current_node.get_next_node().get_value() == value_to_remove:
          current_node.set_next_node(current_node.next_node.get_next_node())
          current_node = None # set to None if value is removed
        else:
          current_node = current_node.get_next_node() # increment current node
```

1.6.3 Stacks

A **stack** is a data structure which contains an ordered set of data, it provides 3 methods for interaction:

· *Push* - adds data to the “top” of the stack.

· *Pop* - returns and removes data from the “top” of the stack.

· *Peek* - returns data from the “top” of the stack without removing it.

Stacks can be implemented using a *linked list* because it’s more efficient than a list or array, where the top of the stack is the head node of a linked list and the bottom of the stack is the tail node. A constraint that may be placed on a stack is its size (be careful of *stack overflow*).

```python
# using Node class from linked list example above
class Stack:
  def __init__(self, limit=1000): # create a new stack
    self.top_item = None
    self.size = 0
    self.limit = limit

  def push(self, value):
    if self.has_space(): # if not full
      item = Node(value) # create new node
      item.set_next_node(self.top_item) # move current node down
      self.top_item = item # set head to new node
      self.size += 1 # increment size
    else:
      print('No more space remaining')

  def pop(self):
    if not self.is_empty(): # if not empty
      item_to_remove = self.top_item # store node in temp varaible
      self.top_item = item_to_remove.get_next_node() # set head to next node
      self.size -= 1 # decrement size
      return item_to_remove.get_value() # return value of head node
    else:
      print("This stack is totally empty.")

  def peek(self):
    if not self.is_empty(): # if not empty
      return self.top_item.get_value() # return value (but don't pop)
    else:
      print("Nothing to see here!")

  def has_space(self):
    return (self.limit > self.size) # if space left in stack

  def is_empty(self):
    return (self.size == 0) # if stack is empty or not
```

1.6.4 Queues

A **queue** is a data structure which contains an ordered set of data and provide 3 methods for interaction:

1) Enqueue - adds data to the back/end of the queue.
2) Dequeue - provides and removes data from the front/beginning of the queue.
3) Peek - reveals data from the front of the queue without removing it.

Queues can be implemented using a linked list as the underlying data structure. The front of the queue is equivalent to the head node and the back of the queue is equivalent to the tail node. Since both ends of the queue must be accessible, a reference to both the head node and the tail node must be maintained. Be careful of *queue under/overflow* (enqueue on a full queue or dequeue from an empty queue). FIFO.

```
# using Node class from linked list example above
class Queue: # bounded queue class
  def __init__(self, max_size=None):
    self.head = None
    self.tail = None
    self.max_size = max_size
    self.size = 0

  def enqueue(self, value):
    if self.has_space():
      item_to_add = Node(value)
      print("Adding " + str(item_to_add.get_value()) + " to the queue!")
      if self.is_empty():
        self.head = item_to_add
        self.tail = item_to_add
      else:
        self.tail.set_next_node(item_to_add)
        self.tail = item_to_add
      self.size += 1
    else:
    print("Sorry, no more room!")

  def dequeue(self):
    if self.get_size() > 0:
      item_to_remove = self.head
      print("Removing " + str(item_to_remove.get_value()) + " from the queue!")
      if self.get_size() == 1:
        self.head = None
        self.tail = None
      else:
        self.head = self.head.get_next_node()
      self.size -= 1
      return item_to_remove.get_value()
    else:
      print("This queue is totally empty!")

  def peek(self):
    if self.is_empty():
      print("Nothing to see here!")
    else:
      return self.head.get_value()

  def get_size(self):
    return self.size

  def has_space(self):
    if self.max_size == None:
      return True
    else:
      return self.max_size > self.get_size()

  def is_empty(self):
    return self.size == 0
```

1.7 Complex Data Structures

1.7.1 Hash Maps