Python Data Structures

Strings, Lists, Tuples, Sets, Dictionaries Python 3.8.8

Sequences: String, List, Tuple

Definitions:

- mutable: liable to change

indexing - access any item in the sequence using its index. Indexing starts with 0 for the first element.

```
# string - sequence of Unicode characters. UTF-8 8-bit values
x = 'frog'
print (x[3])

# list - mutable object
x = ['pig', 'cow', 'horse']
print (x[1])

# tuple - immutable object
x = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print (x[0])
```

```
g
cow
Kevin
```

slicing - slice out substrings, sublists, subtuples using indexes. [start : end+1 : step]

```
x = 'computer'
print(x[1:4])
print(x[1:6:2])
print(x[3:])
print(x[:5])
print(x[-1])
print(x[-3:])
print(x[:-2])
```

```
omp
opt
puter
compu
r
```

```
ter
comput
```

adding / concatenating - combine 2 sequences of the same type by using +

```python
# string
x = 'horse' + 'shoe'
print(x)

# list
y = ['pig', 'cow'] + ['horse']
print(y)

# tuple
z = ('Kevin', 'Niklas', 'Jenny') + ('Craig',)
print(z)
```

```
horseshoe
['pig', 'cow', 'horse']
('Kevin', 'Niklas', 'Jenny', 'Craig')
```

multiplying - mulitply a sequence using *.

```python
# string
x = 'bug' * 3
print(x)

# list
y = [8, 5] * 3
print(y)

# tuple
z = (2, 4) * 3
print(z)
```

```
bugbugbug
[8, 5, 8, 5, 8, 5]
(2, 4, 2, 4, 2, 4)
```

checking membership - test whether an item is or is not in a sequence.

```python
# string
x = 'bug'
print('u' in x)

# list
y = ['pig', 'cow', 'horse']
print('cow' not in y)

# tuple
```

```python
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print('Niklas' in z)
```

```
True
False
True
```

iterating - iterating through the items in a sequence.

```python
# item
x = [7, 8, 3]
for item in x:
    print(item)

# index & item
y = [7, 8, 3]
for index, item in enumerate(y):
    print(index, item)
```

```
7
8
3
0 7
1 8
2 3
```

number of items - count the number of items in a sequence.

```python
# string
x = 'bug'
print(len(x))

# list
y = ['pig', 'cow', 'horse']
print(len(y))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(len(z))
```

```
3
3
4
```

minimum - find the minimum item in a sequence lexicographically. Alpha or numeric types, but cannot mix types.

```python
# string
x = 'bug'
print(min(x))

# list
```

```
y = ['pig', 'cow', 'horse']
print(min(y))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(min(z))
```

```
b
cow
Craig
```

maximum - find the maximum item in a sequence lexicographically. Alpha or number types, but cannot mix types.

```
# string
x = 'bug'
print(max(x))

# list
y = ['pig', 'cow', 'horse']
print(max(y))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(max(z))
```

```
u
pig
Niklas
```

sum - find the sum of items in a sequence. Entire sequence must be numeric.

```
# string -> error
# x = [5, 7, 'bug']
# print(sum(x))   # generates an error

# list
y = [2, 5, 8, 12]
print(sum(y))
print(sum(y[-2:]))

# tuple
z = (50, 4, 7, 19)
print(sum(z))
```

```
27
20
80
```

sorting - returns a new list of items in sorted order. Does not change the orginal list.

```python
# string
x = 'bug'
print(sorted(x))

# list
y = ['pig', 'cow', 'horse']
print(sorted(y))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(sorted(z))
```

```
['b', 'g', 'u']
['cow', 'horse', 'pig']
['Craig', 'Jenny', 'Kevin', 'Niklas']
```

sorting - sort by second letter. Add a key parameter and a lambda function to return the second character. (the word *key* here is a defined paramter name, *k* is an arbitrary variable name).

```python
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(sorted(z, key=lambda k: k[1]))
```

```
['Kevin', 'Jenny', 'Niklas', 'Craig']
```

count(item> - returns count of an item.

```python
# string
x = 'hippo'
print(x.count('p'))

# list
y = ['pig', 'cow', 'horse', 'cow']
print(y.count('cow'))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(z.count('Kevin'))
```

```
2
2
1
```

index(item) - returns the index of the first occurence of an item.

```python
# string
x = 'hippo'
print(x.index('p'))

# list
y = ['pig', 'cow', 'horse', 'cow']
```

```
print(y.index('cow'))

# tuple
z = ('Kevin', 'Niklas', 'Jenny', 'Craig')
print(z.index('Kevin'))
```

```
2
1
0
```

unpacking - unpack the n items of a sequence into n variables.

```
x = ['pig', 'cow', 'horse']
a, b, c = x
print(a, b, c)
```

```
pig cow horse
```

Lists

- General purpose
- Most widely used data structure
- Grow and shrink size as needed
- Sequence type
- Sortable

constructors - creating a new list

```
x = list()
y = ['a', 25, 'dog', 8.43]
tuple1 = (10, 20)
z = list(tuple1)

# list comprehension
a = [m for m in range(8)]
print(a)
b = [i**2 for i in range(10) if i>4]
print(b)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
[25, 36, 49, 64, 81]
```

delete - delete a list or an item in a list.

```
x = [5, 3, 8, 6]
del(x[1])
print(x)
del(x)      # list x no longer exists
```

```
[5, 8, 6]
```

append - append an item to a list.

```
x = [5, 3, 8, 6]
x.append(7)
print(x)
```

```
[5, 3, 8, 6, 7]
```

extend - append a sequence to a list.

```
x = [5, 3, 8, 6]
y = [12, 13]
x.extend(y)
print(x)
```

```
[5, 3, 8, 6, 12, 13]
```

insert - insert an item at a given index.

```
x = [5, 3, 8, 6]
x.insert(1, 7)
print(x)
x.insert(1, ['a', 'm'])
print(x)
```

```
[5, 7, 3, 8, 6]
[5, ['a', 'm'], 7, 3, 8, 6]
```

pop - pops last item off list and returns item.

```
x = [5, 3, 8, 6]
x.pop()      # pop off the 6
print(x)
print(x.pop())
```

```
[5, 3, 8]
8
```

remove - remove first instance of an item

```
x = [5, 3, 8, 6, 3]
x.remove(3)
print(x)
```

```
[5, 8, 6, 3]
```

reverse - reverse the order of the list. It is an in-place sort, meaning it changes the original list.

```
x = [5, 3, 8, 6]
x.reverse()
print(x)
```

```
[6, 8, 3, 5]
```

sort - sort the list in place. Note: sorted(x) returns a new sorted list without changing the original list x. x.sort() puts the items of x in sorted order (sorts in place)

```
x = [5, 3, 8, 6]
x.sort()
print(x)
```

```
[3, 5, 6, 8]
```

reverse sort - sort items descending. Use reverse=*True* parameter to the sort function.

```
x = [5, 3, 8, 6]
x.sort(reverse=True)
print(x)
```

```
[8, 6, 5, 3]
```

Tuples

- Immutable (can't add/change)
- Useful for fixed data
- Faster than Lists
- Sequence type

constructors - creating new tuples.

```
x = ()
x = (1, 2, 3)
x = 1, 2, 3
x = 2,  #the comma tells Python it's a tuple
print(x, type(x))

list1 = [2, 4, 6]
x = tuple(list1)
print(x, type(x))
```

```
(2,) <class 'tuple'>
(2, 4, 6) <class 'tuple'>
```

tuples are immutable, but member objects may be mutable.

```
x = (1, 2, 3)
# del(x[1])        # fails
# x[1] = 8         # fails
print(x)

y = ([1, 2], 3)    # a tuple where the first item is a list
del(y[0][1])       # delete the 2
print(y)           # the list withen the tuple is mutable
```

```
y += (4,)          # concatenating two tuples works
print(y)
```

```
(1, 2, 3)
([1], 3)
([1], 3, 4)
```

Sets

- Store non-duplicate items
- Very fasst access vs Lists
- Math Set ops(union, intersect)
- Sets are Unordered

constructors - creating new sets.

```
x = {3, 5, 3, 5}
print(x)

y = set()
print(y)

list1 = [2, 3, 4]
z = set(list1)
print(z)
```

```
{3, 5}
set()
{2, 3, 4}
```

set operations

```
x = {3, 8, 5}
print(x)
x.add(7)
print(x)

x.remove(3)
print(x)

# get length of set x
print(len(x))

# check membership in x
print(5 in x)

# pop random item from set x
print(x.pop(), x)
```

```
# delete all items from set x
x.clear()
print(x)
```

```
{8, 3, 5}
{8, 3, 5, 7}
{8, 5, 7}
3
True
8 {5, 7}
set()
```

Mathematical set operations intersection (AND): set1 & set2 union (OR): set1 | set1 symmetric difference (XOR): set1 ^ set2 difference (in set1 but not set2): set1 - set2 subset (set2 contains set1): set1 <= set2 superset (set1 contains set2): set1 >= set2

```
s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1 & s2)
print(s1 | s2)
print(s1 ^ s2)
print(s1 - s2)
print(s1 <= s2)
print(s1 >= s2)
```

```
{3}
{1, 2, 3, 4, 5}
{1, 2, 4, 5}
{1, 2}
False
False
```

Dictionaries (dict)

- Key/Value pairs
- Associative array, like Java HashMap
- Dicts are Unordered

```
x = {'pork':25.3, 'beef':33.8, 'chicken':22.7}
print(x)
x = dict([('pork', 25.3), ('beef', 33.8), ('chicken', 22.7)])
print(x)
x = dict(pork=25.3, beef=33.8, chicken=22.7)
print(x)
```

```
{'pork': 25.3, 'beef': 33.8, 'chicken': 22.7}
{'pork': 25.3, 'beef': 33.8, 'chicken': 22.7}
{'pork': 25.3, 'beef': 33.8, 'chicken': 22.7}
```

dict operations

```python
x['shrimp'] = 38.2      # add or update
print(x)

# delete an item
del(x['shrimp'])
print(x)

# get length of dict x
print(len(x))

# delete all items for dict x
x.clear()
print(x)

# delete dict x
del(x)
```

```
{'pork': 25.3, 'beef': 33.8, 'chicken': 22.7, 'shrimp': 38.2}
{'pork': 25.3, 'beef': 33.8, 'chicken': 22.7}
3
{}
```

accessing keys and values in a dict

```python
y = {'pork':25.3, 'beef':33.8, 'chicken':22.7}
print(y.keys())
print(y.values())
print(y.items())       # key-value pairs

# check membership in y_keys (only look in keys, not values)
print('beef' in y)

# check membership in y_values
print('clams' in y.values())
```

```
dict_keys(['pork', 'beef', 'chicken'])
dict_values([25.3, 33.8, 22.7])
dict_items([('pork', 25.3), ('beef', 33.8), ('chicken', 22.7)])
True
False
```

iterating a dict - note, items are in random order.

```python
for key in y:
    print(key, y[key])

for k, v in y.items():
    print(k, v)
```

```
pork 25.3
beef 33.8
```

```
chicken 22.7
pork 25.3
beef 33.8
chicken 22.7
```

Python List Comprehensions

basic format: new_list = [transform sequence [filter]]

```
import random
```

get values withen a range

```
under_10 = [x for x in range(10)]
print('under_10: ' + str(under_10))
```

```
under_10: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

get squared values

```
squares = [x**2 for x in under_10]
print('squares: ' + str(squares))
```

```
squares: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

get odd numbers using mod

```
odds = [x for x in range(10) if x%2 == 1]
print('odds: ' + str(odds))
```

```
odds: [1, 3, 5, 7, 9]
```

get multiples of 10

```
ten_x = [x * 10 for x in range(10)]
print('ten_x: ' + str(ten_x))
```

```
ten_x: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

get all numbers from a string

```
s = 'I love 2 go t0 the store 7 times a w3ek.'
nums = [x for x in s if x.isnumeric()]
print('nums: ' + ''.join(nums))
```

```
nums: 2073
```

get index of a list item

```
names = ['Cosmo', 'Pedro', 'Anu', 'Ray']
idx = [k for k, v in enumerate(names) if v == 'Anu']
print('index = ' + str(idx[0]))
```

```
index = 2
```

delete an item from a list

```
letters = [x for x in 'ABCDEF']
random.shuffle(letters)
letrs = [a for a in letters if a != 'C']
print(letters, letrs)
```

```
['E', 'F', 'B', 'C', 'D', 'A'] ['E', 'F', 'B', 'D', 'A']
```

if-else condition in a comprehension must come before iteration

```
nums = [5, 3, 10, 18, 6, 7]
new_list = [x if x%2 == 0 else 10*x for x in nums]
print('new_list: ' + str(new_list))
```

```
new_list: [50, 30, 10, 18, 6, 70]
```

Stacks, Queues & Heaps

Stack using Python List

Stack is a LIFO data structure -- last-in, first-out. Use append() to push an item onto a stack. Use pop() to remove an item.

```
my_stack = list()
my_stack.append(4)
my_stack.append(7)
my_stack.append(12)
my_stack.append(19)
print(my_stack)
```

```
[4, 7, 12, 19]
```

```
print(my_stack.pop())
print(my_stack.pop())
print(my_stack)
```

```
19
12
[4, 7]
```

Stack using List with a Wrapper Class

We create a Stack class and a full set of Stack methods. But the underlying data structure is really a Python List. For pop and peek methods we first check whether the stack is empty, to avoid exceptions.

```
class Stack():
    def __init__(self):
        self.stack = list()
    def push(self, item):
```

```
        self.stack.append(item)
    def pop(self):
        if len(self.stack) > 0:
            return self.stack.pop()
        else:
            return None
    def peek(self):
        if len(self.stack) > 0:
            return self.stack[len(self.stack)-1]
        else:
            return None
    def __str__(self):
        return str(self.stack)
```

Test Code for Stack Wrapper Class

```
my_stack = Stack()
my_stack.push(1)
my_stack.push(3)
print(my_stack)
print(my_stack.pop())
print(my_stack.peek())
print(my_stack.pop())
print(my_stack.pop())
```

```
[1, 3]
3
1
1
None
```

Queue using Python Deque

Queue is a FIFO data structure -- first-in, first-out. Deque is a double-ended queue, but we can use it for our queue. We use append() to enqueue an item, and popleft() to dequeue an item.

```
from collections import deque
my_queue = deque()
my_queue.append(5)
my_queue.append(10)
print(my_queue)
print(my_queue.popleft())
```

```
deque([5, 10])
5
```

Fun Exercise:

Write a wrapper class for the Queue class, similar to what we did for Stack deque. Try adding enqueue, dequeue, and get_size methods.

Python Maxheap

A MaxHeap always bubbles the highest value to th etop, so it can be removed instantly. Public functions: push, peek, pop Private functions: **swap, _flotUp, _bubbleDown, _str.**

```
class MaxHeap:
    def __init__(self, items=[]):
        super().__init__()
        self.heap = [0]
        for item in items:
            self.heap.append(item)
            self.__floatUp(len(self.heap) - 1)

    def push(self, data):
        self.heap.append(data)
        self.__floatUp(len(self.heap) - 1)

    def peek(self):
        if self.heap[1]:
            return self.heap[1]
        else:
            return False

    def pop(self):
        if len(self.heap) > 2:
            self.__swap(1, len(self.heap) - 1)
            max = self.heap.pop()
            self.__bubbleDown(1)
        elif len(self.heap) == 2:
            max = self.heap.pop()
        else:
            max = False
        return max

    def __swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def __floatUp(self, index):
        parent = index//2
        if index <= 1:
            return
        elif self.heap[index] > self.heap[parent]:
            self.__swap(index, parent)
            self.__floatUp(parent)
```

```python
    def __bubbleDown(self, index):
        left = index * 2
        right = index * 2 + 1
        largest = index
        if len(self.heap) > left and self.heap[largest] < 
self.heap[left]:
                largest = left
        if len(self.heap) > right and self.heap[largest] < 
self.heap[right]:
            largest = right
        if largest != index:
            self.__swap(index, largest)
            self.__bubbleDown(largest)

    def __str__(self):
        return str(self.heap)
```

MaxHeap Test Code

```python
m = MaxHeap([96, 3, 21])
m.push(10)
print(m)
print(m.pop())
print(m.peek())
```

```
[0, 96, 10, 21, 3]
96
21
```

Python Linked Lists

Node Class

Node class has a constructor that sets the data passed in, and optionally can optionally set the next_node and prev_node. It also has a str method to give a string representation for printing. Note that prev_node is only used for Doubly Linked Lists.

```python
class Node:

    def __init__ (self, d, n=None, p=None):
        self.data = d
        self.next_node = n
        self.prev_node = p

    def __str__ (self):
        return ('(' + str(self.data) + ')')
```

LinkedList Class

A LinkedList object has two attributes: a root node that defaults to None, and size that defaults to 0.

Add method receives a piece of data, creates a new Node, setting the root node as the next node and changes the LL's pointer to the new node, and increments size.

Find iterates through the nodes until it finds the data passed in. If it finds the data is will return it, otherwise returns None.

Remove needs pointers to this_node and prev_node. If it finds the data, it needs to check if it is in the root node (prev_node is None) before deciding how to bypass the deleted node.

Print_list iterates the list and prints each node.

```
class LinkedList:

    def __init__(self, r = None):
        self.root = r
        self.size = 0

    def add(self, d):
        new_node = Node(d, self.root)
        self.root = new_node
        self.size += 1

    def find(self, d):
        this_node = self.root
        while this_node is not None:
            if this_node.data == d:
                return d
            else:
                this_node = this_node.next_node
        return None

    def remove(self, d):
        this_node = self.root
        prev_node = None

        while this_node is not None:
            if this_node.data == d:
                if prev_node is not None:  # data is in non-root
                    prev_node.next_node = this_node.next_node
                else:  # data is in root node
                    self.root = this_node.next_node
                self.size -= 1
                return True  # data removed
            else:
                prev_node = this_node
                this_node = this_node.next_node
```

```
            return False  # data not found

    def print_list(self):
        this_node = self.root
        while this_node is not None:
            print(this_node, end='->')
            this_node = this_node.next_node
        print('None')
```

Linked List Test Code

This test code adds nodes to the LinkedList, Prints the list, prints the size, removes an item, and finds an item.

```
myList = LinkedList()
myList.add(5)
myList.add(8)
myList.add(12)
myList.print_list()

print("size="+str(myList.size))
myList.remove(8)
print("size="+str(myList.size))
print(myList.find(5))
print(myList.root)
```

```
(12)->(8)->(5)->None
size=3
size=3
5
(12)
```

Circular Linked List

Includes attributes root and size Includes methods add, find, remove, and print_list.

```
class CircularLinkedList:

    def __init__(self, r = None):
        self.root = r
        self. size = 0

    def add (self, d):
        if self.size == 0:
            self.root = Node(d)
            self.root.next_node = self.root
        else:
            new_node = Node (d, self.root.next_node)
            self.root.next_node = new_node
```

```python
            self.size += 1

    def find (self, d):
        this_node = self.root
        while True:
            if this_node.data == d:
                return d
            elif this_node.next_node == self.root:
                return False
            this_node = this_node.next_node

    def remove (self, d):
        this_node = self.root
        prev_node = None

        while True:
            if this_node.data == d:  # found
                if prev_node is not None:
                    prev_node.next_node = this_node.next_node
                else:
                    while this_node.next_node != self.root:
                        this_node = this_node.next_node
                    this_node.next_node = self.root.next_node
                    self.root = self.root.next_node
                self.size -= 1
                return True  # data removed
            elif this_node.next_node == self.root:
                return False # data not found
            prev_node = this_node
            this_node = this_node.next_node

    def print_list (self):
        if self.root is None:
            return
        this_node = self.root
        print (this_node, end='->')
        while this_node.next_node != self.root:
            this_node = this_node.next_node
            print (this_node, end='->')
        print()
```

Circular Linked List Test Code

```python
cll = CircularLinkedList()
for i in [5, 7, 3, 8, 9]:
    cll.add(i)

print("size="+str(cll.size))
print(cll.find(8))
```

```
print(cll.find(12))

my_node = cll.root
print (my_node, end='->')
for i in range(8):
    my_node = my_node.next_node
    print (my_node, end='->')
print()
```

```
size=5
8
False
(5)->(9)->(8)->(3)->(7)->(5)->(9)->(8)->(3)->
```

```
cll.print_list()
cll.remove(8)
print(cll.remove(15))
print("size="+str(cll.size))
cll.remove(5)  # delete root node
cll.print_list()
```

```
(5)->(9)->(8)->(3)->(7)->
False
size=4
(9)->(3)->(7)->
```

Doubly Linked List

```
class DoublyLinkedList:

    def __init__ (self, r = None):
        self.root = r
        self.last = r
        self.size = 0

    def add (self, d):
        if self.size == 0:
            self.root = Node(d)
            self.last = self.root
        else:
            new_node = Node(d, self.root)
            self.root.prev_node = new_node
            self.root = new_node
        self.size += 1

    def find (self, d):
        this_node = self.root
        while this_node is not None:
            if this_node.data == d:
                return d
            elif this_node.next_node == None:
                return False
```

```
            else:
                this_node = this_node.next_node

    def remove (self, d):
        this_node = self.root
        while this_node is not None:
            if this_node.data == d:
                if this_node.prev_node is not None:
                    if this_node.next_node is not None: # delete a
middle node
                        this_node.prev_node.next_node =
this_node.next_node
                        this_node.next_node.prev_node =
this_node.prev_node
                    else: # delete last node
                        this_node.prev_node.next_node = None
                        self.last = this_node.prev_node
                else: # delete root node
                    self.root = this_node.next_node
                    this_node.next_node.prev_node = self.root
                self.size -= 1
                return True # data removed
            else:
                this_node = this_node.next_node
        return False # data not found

    def print_list (self):
        if self.root is None:
            return
        this_node = self.root
        print (this_node, end='->')
        while this_node.next_node is not None:
            this_node = this_node.next_node
            print (this_node, end='->')
        print()
```

Doubly Linked List Test Code

```
dll = DoublyLinkedList()
for i in [5, 9, 3, 8, 9]:
    dll.add(i)

print("size="+str(dll.size))
dll.print_list()
dll.remove(8)
print("size="+str(dll.size))

size=5
(9)->(8)->(3)->(9)->(5)->
size=4
```

```
print(dll.remove(15))
print(dll.find(15))
dll.add(21)
dll.add(22)
dll.remove(5)
dll.print_list()
print(dll.last.prev_node)
```

```
False
False
(22)->(21)->(9)->(3)->(9)->
(3)
```

Binary Search Tree

Constructor sets three attributes: data, left subtree and right subtree. **Insert** inserts a new subtree into the proper location. **Find** finds a value. If value not found, return False. **Get_size** returns the number of nodes in the tree (excluding None nodes). **Preorder** prints a perorder traversal of the tree. **Inorder** print an indorder traversal of the tree.

```
class Tree:
    def __init__(self, data, left=None, right=None):
        self.data = data
        self.left = left
        self.right = right
    def insert(self, data):
        if self.data == data:
            return False # duplicate value
        elif self.data > data:
            if self.left is not None:
                return self.left.insert(data)
            else:
                self.left = Tree(data)
                return True
        else:
            if self.right is not None:
                return self.right.insert(data)
            else:
                self.right = Tree(data)
                return True
    def find(self, data):
        if self.data == data:
            return data
        elif self.data > data:
            if self.left is None:
                return False
            else:
                return self.left.find(data)
        elif self.data < data:
            if self.right is None:
```

```
                return False
            else:
                return self.right.find(data)
    def get_size(self):
        if self.left is not None and self.right is not None:
            return 1 + self.left.get_size() + self.right.get_size()
        elif self.left:
            return 1 + self.left.get_size()
        elif self.right:
            return 1 + self.right.get_size()
        else:
            return 1
    def preorder(self):
        if self is not None:
            print (self.data, end=' ')
            if self.left is not None:
                self.left.preorder()
            if self.right:
                self.right.preorder()
    def inorder(self):
        if self is not None:
            if self.left is not None:
                self.left.inorder()
            print (self.data, end=' ')
            if self.right is not None:
                self.right.inorder()
```

Test Code

We create a new tree, insert one value, insert a whole list of values, find all values for 1 to 15 (False for 0, 5 and 8 shows that those values ar enot in the tree), print the size of the tree, print preorder and postorder traversals.

```
tree = Tree(7)
tree.insert(9)
for i in [15, 10, 2, 12, 3, 1, 13, 6, 11, 4, 14, 9]:
    tree.insert(i)
for i in range(16):
    print(tree.find(i), end=' ')
print('\n', tree.get_size())

tree.preorder()
print()
tree.inorder()
print()
```

```
False 1 2 3 4 False 6 7 False 9 10 11 12 13 14 15
 13
```

```
7 2 1 3 6 4 9 15 10 12 11 13 14
1 2 3 4 6 7 9 10 11 12 13 14 15
```

Vertex Class

The Vertex class has a constructor that sets the name of the vertex (in our example, just a letter), and creates a new empty set to store neighbors.

The add_neighbor method adds the name of a neighboring vertex to the neighbors set. This set automatically eliminates duplicates.

```
class Vertex:
    def __init__(self, n):
        self.name = n
        self.neighbors = set()

    def add_neighbor(self, v):
        self.neighbors.add(v)
```

Graph Class

The Graph class uses a dictionary to store vertices in the format, vertex_name:vertex_object.

Adding a new vertex to the graph, we first check if the object passed in is a vertex obect, then we check if it already exists in the graph. If both checks pass, then we add the vertex to the graph's vertices dictionary.

When adding an edge, we receive two vertex names, we first check if both vertex names are valid, then we add each to the others's neighbors set.

To print the graph, we iterate through the vertices, and print each vertex name (the key) followed by its sorted neighbors list.

```
class Graph:
    vertices = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex) and vertex.name not in 
self.vertices:
            self.vertices[vertex.name] = vertex
            return True
        else:
            return False

    def add_edge(self, u, v):
        if u in self.vertices and v in self.vertices:
            self.vertices[u].add_neighbor(v)
            self.vertices[v].add_neighbor(u)
            return True
        else:
```

```
            return False

    def print_graph(self):
        for key in sorted(list(self.vertices.keys())):
            print(key, sorted(list(self.vertices[key].neighbors)))
```

Test Code

Here we create a new Graph object. We create a new vertex named A. We add A to the graph. Then we add new vertex B to the graph. Then we iterate from A to K and add a bunch of vertices to the graph. Since the add_vertex method checks for duplicated, A and B are not add twice.

```
g = Graph()
a = Vertex('A')
g.add_vertex(a)
g.add_vertex(Vertex('B'))
for i in range(ord('A'), ord('K')):
    g.add_vertex(Vertex(chr(i)))
```

An edge consists of two vertex names. Here we iterate through a list of edges and add each to the graph.

This print_graph method does'nt give a very good visualization of the graph, but it does show the neighbors for each vertex.

```
edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ',
'GJ', 'HI']
for edge in edges:
    g.add_edge(edge[0], edge[1])

g.print_graph()
```

```
A ['B', 'E']
B ['A', 'F']
C ['G']
D ['E', 'H']
E ['A', 'D', 'H']
F ['B', 'G', 'I', 'J']
G ['C', 'F', 'J']
H ['D', 'E', 'I']
I ['F', 'H']
J ['F', 'G']
```

Graph Implementation Using Adjacency Matrix

for undirected graph, with weighted or unweighted edges.

Vertex Class

A vertex object only needs to store its name

```
class Vertex:
    def __init__(self, n):
        self.name = n
```

Graph Class

A graph has three attributes: **vertices** - a dictionary with vertex_name:vertex_object. **edges** - a 2-dimensional list (ie. a matrix) of edges. for an unweighted graph it will contain 0 for no edge and 1 for edge. **edge_indices** - a dictionary with vertex_name:list_index (eg. A:0) to access edges. add_vertex updates all thee of these attributes. add_edge only needs to update the edges matrix.

```
class Graph:
    vertices = {}
    edges = []
    edge_indices = {}

    def add_vertex(self, vertex):
        if isinstance(vertex, Vertex) and vertex.name not in
self.vertices:
            self.vertices[vertex.name] = vertex
            # for loop appends a column of zeros to the edges matrix
            for row in self.edges:
                row.append(0)
            # append a row of zeros to the bottom of the edges matrix
            self.edges.append([0] * (len(self.edges)+1))
            self.edge_indices[vertex.name] = len(self.edge_indices)
            return True
        else:
            return False

    def add_edge(self, u, v, weight=1):
        if u in self.vertices and v in self.vertices:
            self.edges[self.edge_indices[u]][self.edge_indices[v]] =
weight
            self.edges[self.edge_indices[v]][self.edge_indices[u]] =
weight
            return True
        else:
            return False

    def print_graph(self):
        for v, i in sorted(self.edge_indices.items()):
            print(v + ' ', end='')
            for j in range(len(self.edges)):
```

```
                print(self.edges[i][j], end=' ')
            print(' ')
```

Test Code

Here we create a new Graph object. We create a new vertex named A. We add A to the graph. Then we add new vertex B to the graph. Then we iterate from A to K and add a bunch of vertices to the graph. Since the add_vertex method checks for duplicates, A and B are not added twice. This is exactly the same test code we used for the graph and adjacency lists.

```
g = Graph()
a = Vertex('A')
g.add_vertex(a)
g.add_vertex(Vertex('B'))
for i in range(ord('A'), ord('K')):
    g.add_vertex(Vertex(chr(i)))
```

An edge consists of two vertex names. Here we iterate through a list of edges and add each to the graph.

This print_graph method does'nt give a very good visualization of the graph, but it does show the adjacency matrix so we can see each vertex's neighbors.

```
edges = ['AB', 'AE', 'BF', 'CG', 'DE', 'DH', 'EH', 'FG', 'FI', 'FJ', 
'GJ', 'HI']
for edge in edges:
    g.add_edge(edge[0], edge[1])

g.print_graph()
```

```
A 0 1 0 0 1 0 0 0 0 0
B 1 0 0 0 0 1 0 0 0 0
C 0 0 0 0 0 0 1 0 0 0
D 0 0 0 0 1 0 0 1 0 0
E 1 0 0 1 0 0 0 1 0 0
F 0 1 0 0 0 0 1 0 1 1
G 0 0 1 0 0 1 0 0 0 1
H 0 0 0 1 1 0 0 0 1 0
I 0 0 0 0 0 1 0 1 0 0
J 0 0 0 0 0 1 1 0 0 0
```