Intro

The way that we structure our information in computer science is everything. Each formation presents new possibilities, some incredibly powerful and some incredibly practical. Nonetheless, knowing data-structures and how to implement them will take your coding and your interviews to the next level. For this blog, I would suggest a strong familiarity with Python and OOP, as I will be demonstrating each data-structure in snippets of Python code.

If you would like to see these code snippets in action, the notebook is available for download on my Github: https://github.com/chrism47/datastruct_intro_notebook
I would strongly encourage you to get your hands on these structures to get a feel for what they do.

The Four Broad Categories: not all-encompassing

- Arrays: Types of lists
- Trees: Like a list, but divergent at each index
- Hash: Hashed indices for fast operation
- Graph: Multi-aspect analysis

Arrays: lists of all types

First on the list is **arrays.** Arrays are a basic structure containing a sequence of elements. Based on your language the nature of arrays may be different. For instance, in C++ your variables must have a type declaration as it is a statically-typed language. In dynamically-typed languages like Python, however, you are permitted to use a mixture of different data types; strings, integers, floating-point, complex, and boolean; all in sequence within a single array, or “list” in Python.

Python example:

```
my_array = [1, 5, 8, 2, 18, 17, 43, 12]
for n in range(len(my_array)):
  if my_array[n] % 2 == 0:
    print(f"{my_array[n]}: is even")
  elif my_array[n] % 2 == 1:
    print(f"{my_array[n]}: is odd")
#iterate through the array and seperate odds from evens
```

```
1: is odd
5: is odd
8: is even
2: is even
18: is even
17: is odd
43: is odd
12: is even
```

Linked Lists: memory adaptive, modular lists

Another data structure you may run into is a **linked list.** Linked lists are used in circumstances where memory allocation is a concern, as they are non-contiguous. Each element in the list defines the next location in the list, so wherever they are in memory they know how to find each other.

The implementation of a linked list is a trade-off between memory management and time complexity. Where in an array you can simply target a position: my_array[n]. In a linked list you have to iterate for a certain number of steps, from one node to the next until the desired location.

Python example:

```
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None
#define the node class. Objects will receive desired data

class LinkedList:
  def __init__(self):
    self.head = None
#initialize with empty value. The list will be an object made up
#of node objects
```

```python
  def append(self, data):
    new_node = Node(data)
    if not self.head:
      self.head = new_node
      return
    current = self.head
    while current.next:
      current = current.next
    current.next = new_node
#the data is placed in first available location
#current.next initializes the next position in the list
```

```python
  def display(self):
    current = self.head
    while current:
      print(current.data, end=" -> ")
      current = current.next
    print("None")
#display function iterates through list until current.data == None

my_linked_list = LinkedList()
my_linked_list.append(1)
my_linked_list.append(2)
my_linked_list.append(3)
#initialize and append to list

my_linked_list.display()
# Display the linked list
```

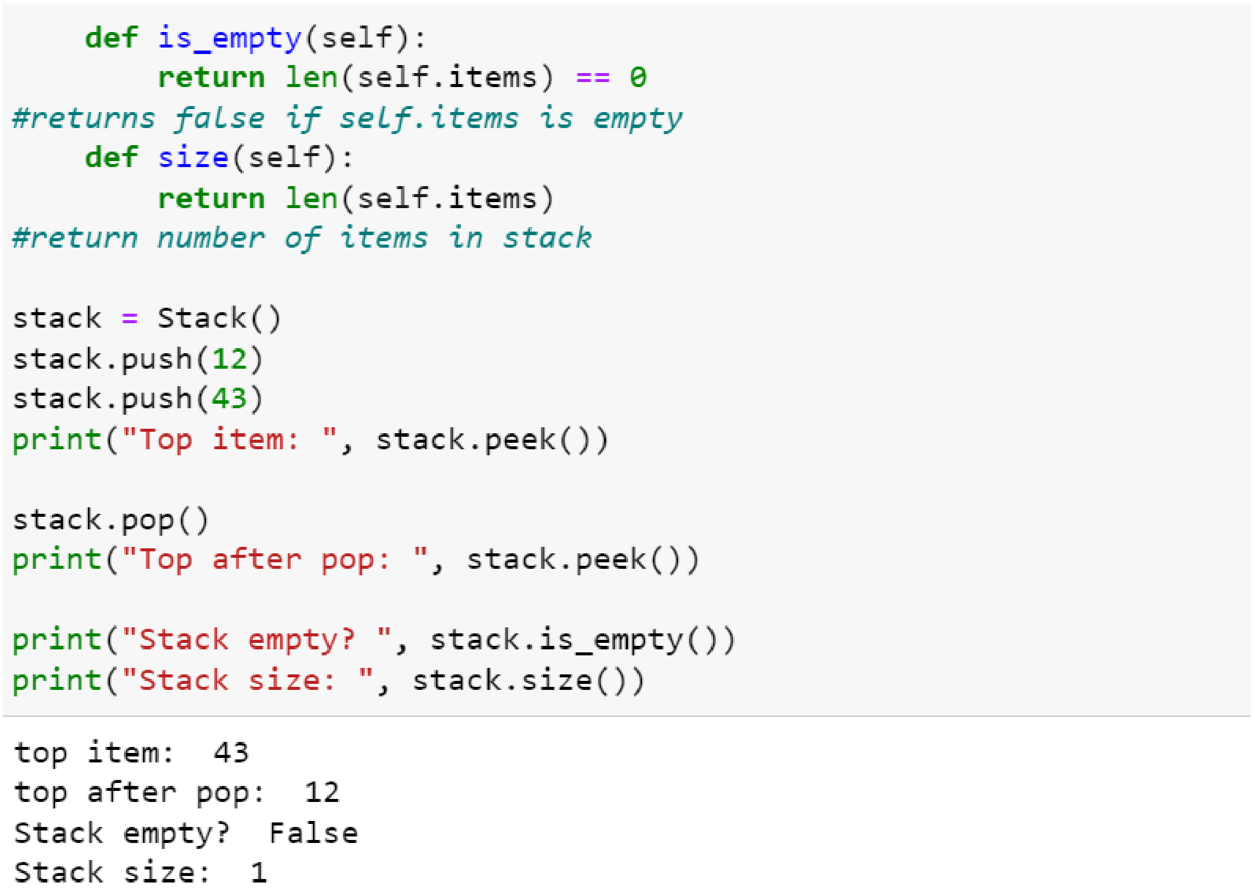
```
1 -> 2 -> 3 -> None
```

Stacks: last in first out

The **stack** is another useful data structure to familiarize yourself with. It utilizes a last-in-first-out (LIFO) method for storing and accessing data. Stacks are under the hood of a lot of behaviors you're likely familiar with; forward-backward in the browser, undo, syntax evaluation in IDEs and PEMDAS operations for example. Its power is in keeping track and structuring processes that need multiple layers of behavior.

Python example:

```python
class Stack:
    def __init__(self):
        self.items = []
#class to initialize empty list objects
    def push(self, item):
        self.items.append(item)
#create "push" behavior for appending
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            return None
#create "pop" behavior for removing with constraints
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            return None
#peek behavior for checking last item on list
```

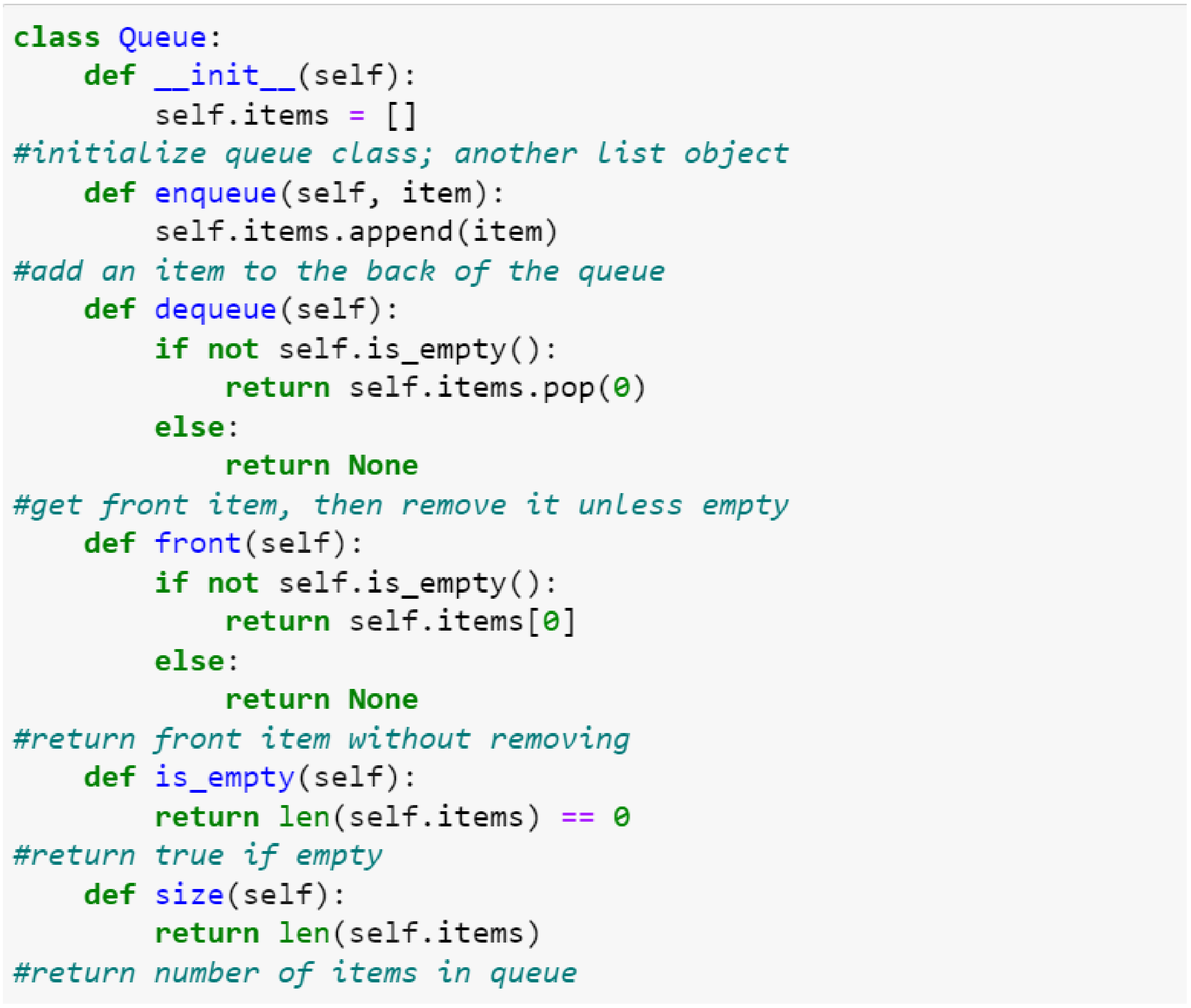


```
    def is_empty(self):
        return len(self.items) == 0
#returns false if self.items is empty
    def size(self):
        return len(self.items)
#return number of items in stack

stack = Stack()
stack.push(12)
stack.push(43)
print("Top item: ", stack.peek())

stack.pop()
print("Top after pop: ", stack.peek())

print("Stack empty? ", stack.is_empty())
print("Stack size: ", stack.size())
```

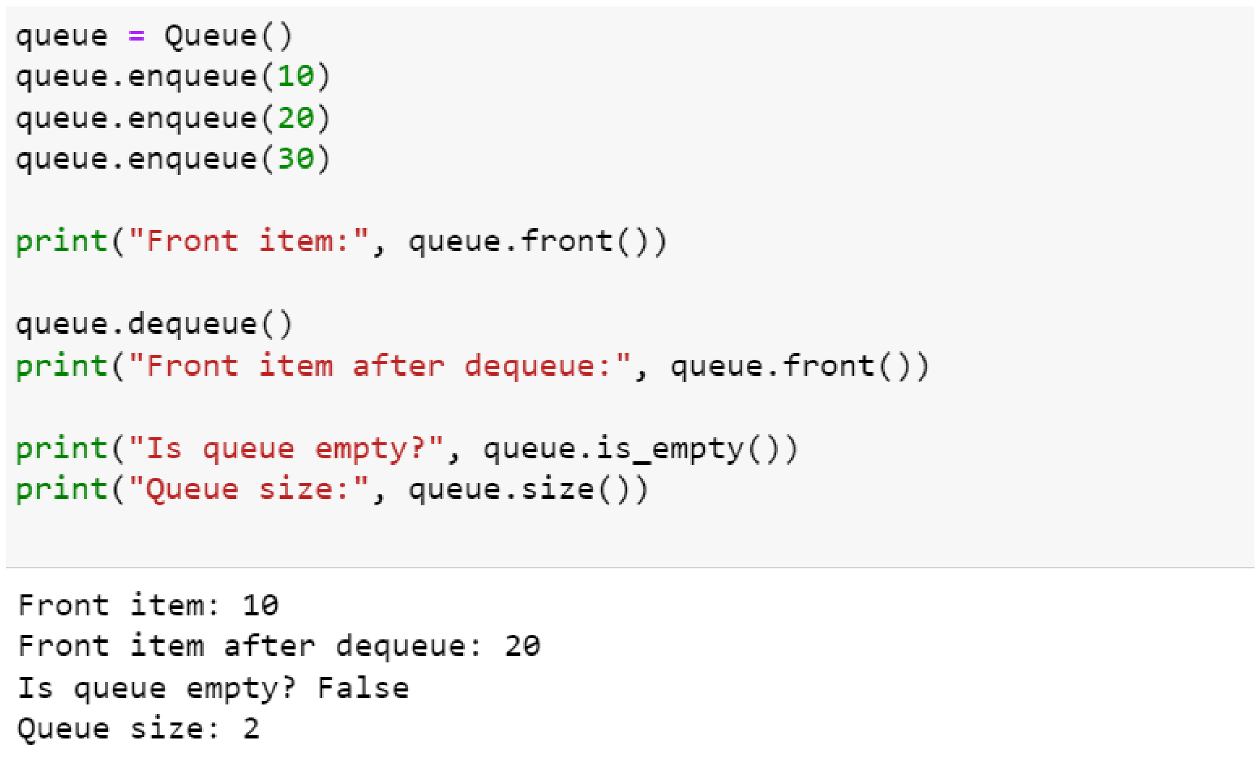
```
top item:  43
top after pop:  12
Stack empty?  False
Stack size:  1
```

Queues: first in first out
If you understand stacks, then wrapping your head around **queues** should be fairly simple. Rather than LIFO they implement first-in-first-out (FIFO). If you picture a line at a grocery store, that is the same function and is even referred to as a queue in business-speak. Queuing systems are implemented for task-scheduling, request handling and printing. Any time you want to prioritize processes in an orderly fashion based on when they begin, a queue is the solution.

Python Example:



```python
class Queue:
    def __init__(self):
        self.items = []
#initialize queue class; another list object
    def enqueue(self, item):
        self.items.append(item)
#add an item to the back of the queue
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            return None
#get front item, then remove it unless empty
    def front(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return None
#return front item without removing
    def is_empty(self):
        return len(self.items) == 0
#return true if empty
    def size(self):
        return len(self.items)
#return number of items in queue
```



```python
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print("Front item:", queue.front())

queue.dequeue()
print("Front item after dequeue:", queue.front())

print("Is queue empty?", queue.is_empty())
print("Queue size:", queue.size())
```

```
Front item: 10
Front item after dequeue: 20
Is queue empty? False
Queue size: 2
```

Trees: nodes and branches

All of the data structures up to this point have been variations of arrays. The next part is dedicated to a different paradigm known as **trees.** The key difference between list-based and tree-based structures could be summed up simply as linear-structure(list-based) vs hierarchical-structure(tree-based). When it is advantageous to have branching throughout different layers, you may consider one of the many tree-based structures.

Binary Search Trees (BST): sorting and searching

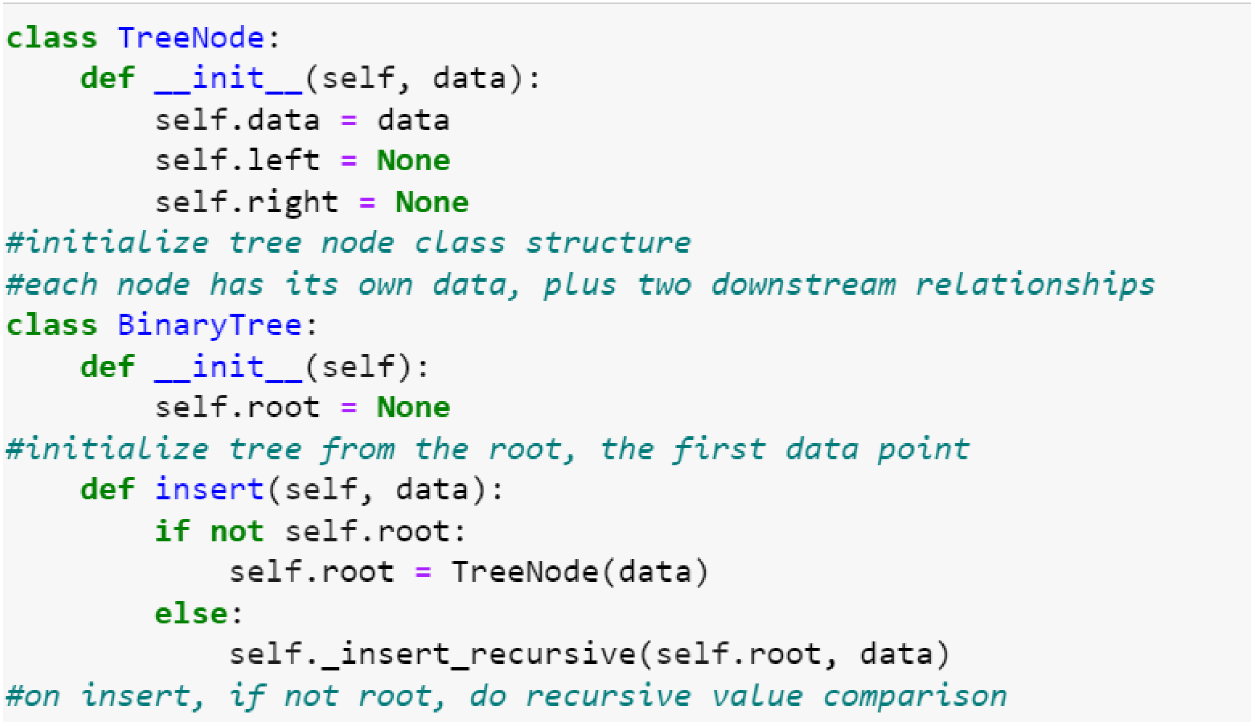
Binary search trees are an efficient organizational structure for reducing complexity of searches. The data in each node is a hint to what the next node may contain. In this case higher values are to the right and lower values to the left; which makes it not only a quick search, but iterable in order.

In the following example, you'll see values entered in a random order and placed in sequence via the structure of the tree:

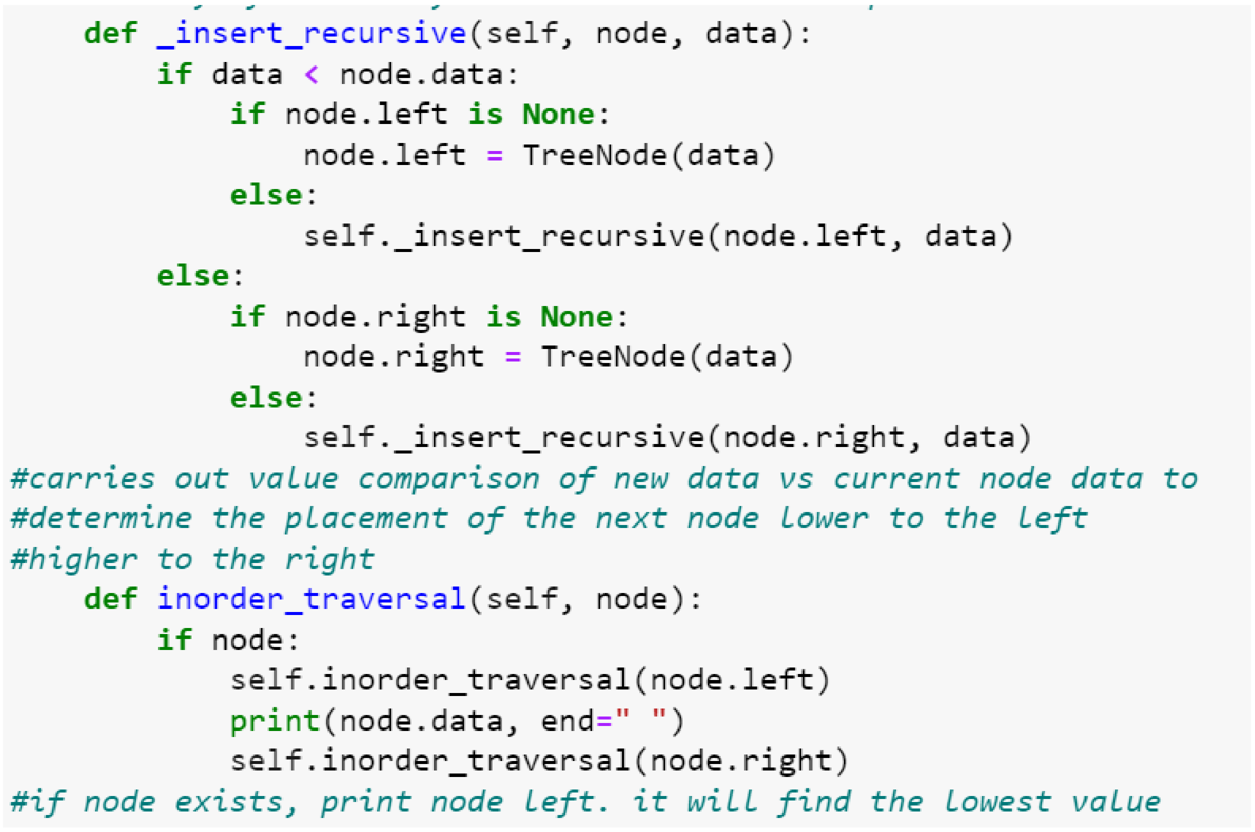
```
     5
    / \
   3   8
  / \
 2   4
```

Binary search structures are used in situations that require disordered inputs to be sorted into an order; ascending or alphabetizing for instance.

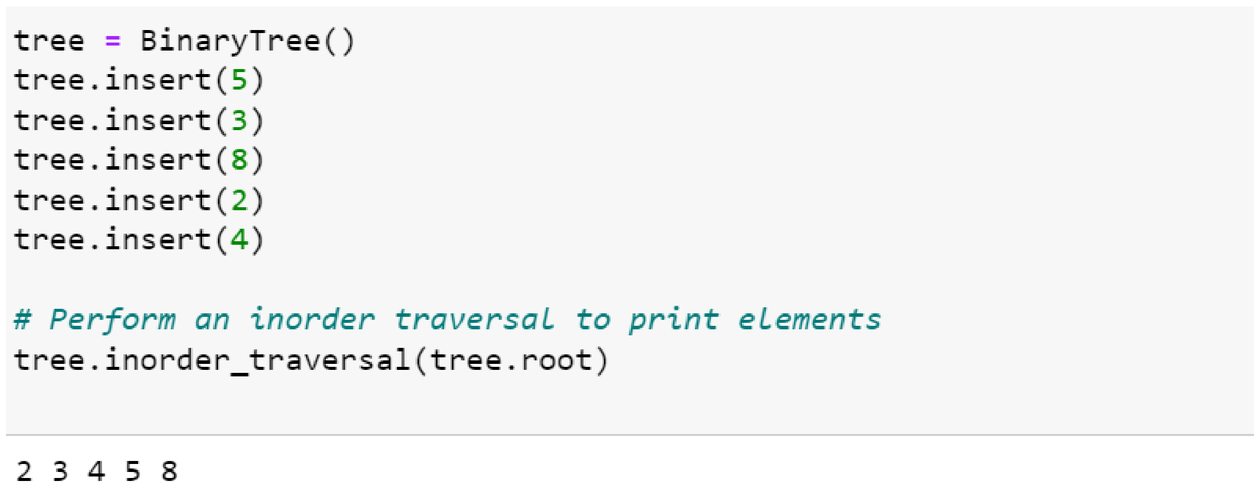
Python example:



```python
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
#initialize tree node class structure
#each node has its own data, plus two downstream relationships
class BinaryTree:
    def __init__(self):
        self.root = None
#initialize tree from the root, the first data point
    def insert(self, data):
        if not self.root:
            self.root = TreeNode(data)
        else:
            self._insert_recursive(self.root, data)
#on insert, if not root, do recursive value comparison
```



```python
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = TreeNode(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = TreeNode(data)
            else:
                self._insert_recursive(node.right, data)
#carries out value comparison of new data vs current node data to
#determine the placement of the next node lower to the left
#higher to the right
    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(node.data, end=" ")
            self.inorder_traversal(node.right)
#if node exists, print node left. it will find the lowest value
```



```
tree = BinaryTree()
tree.insert(5)
tree.insert(3)
tree.insert(8)
tree.insert(2)
tree.insert(4)

# Perform an inorder traversal to print elements
tree.inorder_traversal(tree.root)
```

```
2 3 4 5 8
```

For an experiment, if you want it to print highest-to-lowest just change the order of "inorder_transversal." If you iterate from the right to the left the values will descend. It is due to the nature of this tree that left to right generates low-to-high.

It is important that we distinguish the difference between a **binary tree** and a **binary search tree**. While the search tree is a binary tree, it is of the ordered variety. A *binary tree* is defined as a structure in which each node has at most a left and right branch. What defines the content of the next node is application specific and could be ordered or disordered.

Heaps: what's this bubbling

Another version of the tree structure is the **heap.** Heaps are similar to the binary search tree in that they apply order to the information. The ordering is slightly different though and often applied to caching and queuing due to the "heap" property. The parent and child relationships of each node are defined and redefined until the desired value either "bubbles up" or "bubbles down." The so-called bubbling is the trading of position between parent and child values.

Python example:

```
class MaxHeap:
    def __init__(self):
        self.heap = []
    #initialize the head of the heap, the max
    def insert(self, value):
        self.heap.append(value)
        self._heapify_up(len(self.heap) - 1)
    #determine placement from value entry
    def _heapify_up(self, index):
        parent_index = (index - 1) // 2 #find parent node
        while parent_index >= 0 and (
            self.heap[parent_index] < self.heap[index]):
            #loop while greater
            self.heap[parent_index], self.heap[index] = (
                self.heap[index], self.heap[parent_index])
            #swap parent and child values
            index = parent_index
            parent_index = (index - 1) // 2
            #find new parent index from new index
            #while parent value is larger, switch places
```

```
    def extract_max(self):
        if len(self.heap) == 0:
            return None
        max_value = self.heap[0]
        last_value = self.heap.pop()
        if len(self.heap) > 0:
            self.heap[0] = last_value
            self._heapify_down(0)
        return max_value
    #if empty, None
    #get last_value from heap, if not yet empty evaluate next node
    def _heapify_down(self, index):
        left_child_index = 2 * index + 1
        right_child_index = 2 * index + 2
        largest_index = index
        #store current index in largest_index, will change
        #value until largest
        if (left_child_index < len(self.heap)) and (
            self.heap[left_child_index] > self.heap[largest_index]):
            largest_index = left_child_index
            #if left is larger, replace current with left
        if (right_child_index < len(self.heap)) and (
            self.heap[right_child_index] > self.heap[largest_index]):
            largest_index = right_child_index
            #if right is larger, replace current with right
        if largest_index != index:
            self.heap[index], self.heap[largest_index] = (
                self.heap[largest_index], self.heap[index])
            self._heapify_down(largest_index)
            #if largest not index, swap index and recurse
    def is_empty(self):
        return len(self.heap) == 0
        #true if empty
```

```
heap = MaxHeap()
heap.insert(10)
heap.insert(20)
heap.insert(15)
heap.insert(30)
heap.insert(25)

print("Max elements extracted in descending order:")
while not heap.is_empty():
    print(heap.extract_max(), end=" ")
    #print values in descending order;
    #in summary: find max then pop max then restart
```

```
Max elements extracted in descending order:
30 25 20 15 10
```

Trie: “try” that sequence

So far we've talked a lot about value organization, but the data types we work with are more versatile than simple numbers. Sometimes you'll need a structure for your string data. In the **trie** your string data is split and sequenced for retrieval. Each node is marked as either the end of the word or not. In the case of a word search it will compare the value at each node, if the words match and have matching end of word flags it will return matching status. Without the end of word indicator you may have unintended matching, but with it you have a simple indicator for exact matching. The trie can be adapted to do less exact comparisons as well, for operations like spell-check, DNA analysis, predictive text or search functions.

Python example:

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
#initialize node class.
class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]

        node.is_end_of_word = True
#constructs new nodes and branches from characters
#the final char is marked by end_of_word to signify the sequence
```

```python
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
#matches sequence in char per char form, if end doesn't match
#return false
```

```python
trie = Trie()
trie.insert("apple")
trie.insert("app")
trie.insert("banana")
trie.insert("bat")
#add some values to trie

print(trie.search("apple"))
print(trie.search("app"))
print(trie.search("banana"))
print(trie.search("batman"))
#search for words in trie
```

```
True
True
True
False
```

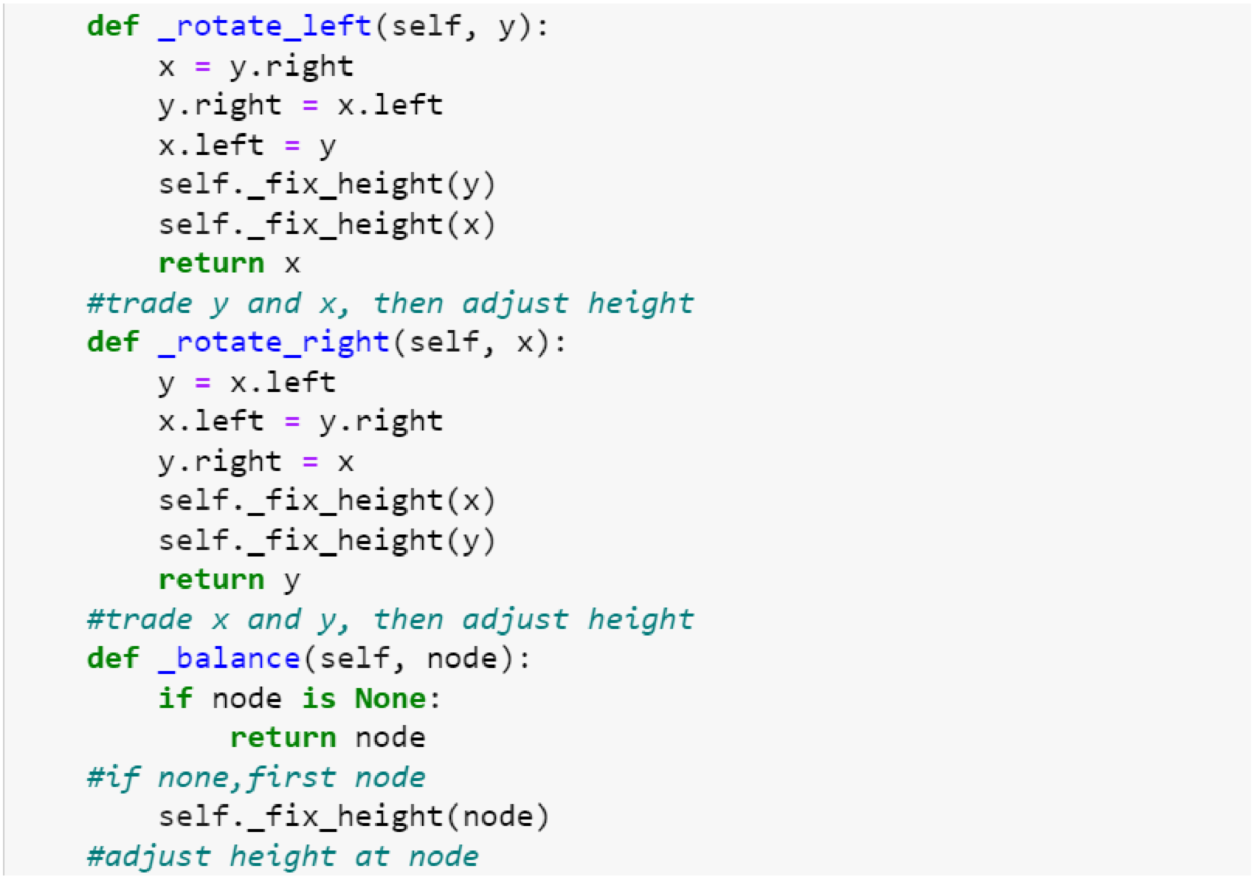
Where trees are concerned, their main advantage over list-types is their organizational variability. This is an asset that helps to facilitate quicker retrieval times. Another tree-style structure is the **Adelson-Velsky and Landis (AVL)**; named after the two Soviet mathematicians who introduced the concept in 1962. The AVL structure is designed to maintain a “height”, regulating the length of the branches. It makes search more efficient, but makes storage more complicated.

Adelson-Velsky and Landis (AVL): fitting in

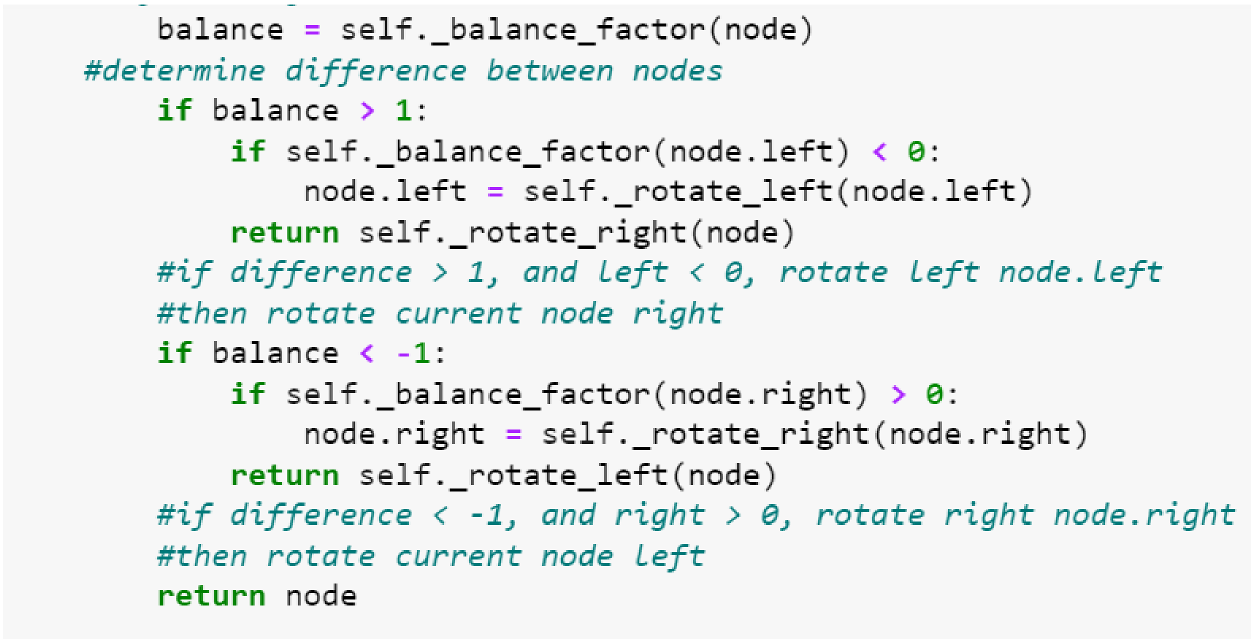
AVLs are a form of BST and the first type of tree in this writing that showcases self-adjustment; a characteristic you’ll see in the next few examples. Self-adjustment is a term I use to describe a data structure that manipulates its own structure upon the input of new data. In this case, the AVL tree maintains a height across nodes. Whenever a piece of data is inserted, the tree measures the relative node lengths and uses “rotations” to rebalance the structure. It won’t allow the difference in “height” to be more than 1.

Python example:

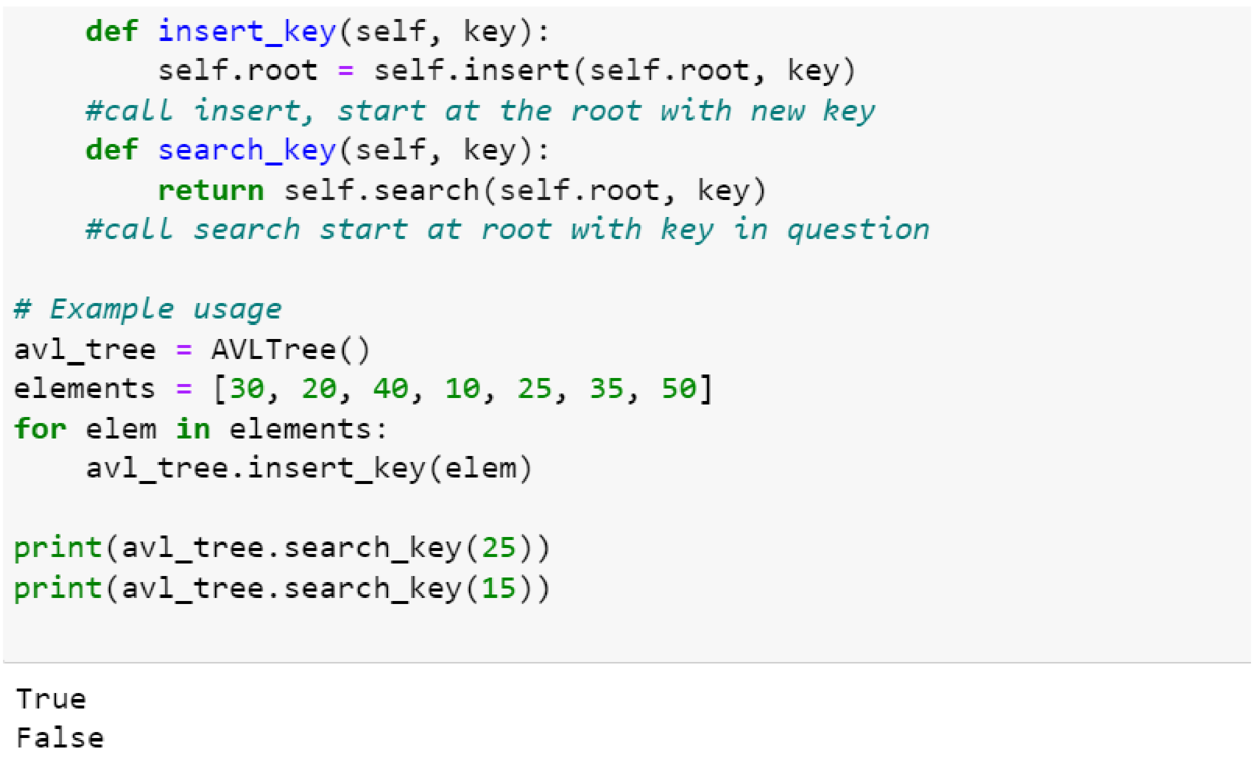
```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1
    #make node class with height limit of one

class AVLTree:
    def __init__(self):
        self.root = None
    #instantiate empty root and AVL tree instructions
    def _height(self, node):
        return node.height if node else 0
    #determine height, length,  of node
    def _balance_factor(self, node):
        return self._height(node.left) - self._height(node.right)
    #determine difference between branch lengths
    def _fix_height(self, node):
        node.height = max(self._height(node.left),
                          self._height(node.right)) + 1
    #reassigns height value from longest
```



```python
    def _rotate_left(self, y):
        x = y.right
        y.right = x.left
        x.left = y
        self._fix_height(y)
        self._fix_height(x)
        return x
    #trade y and x, then adjust height
    def _rotate_right(self, x):
        y = x.left
        x.left = y.right
        y.right = x
        self._fix_height(x)
        self._fix_height(y)
        return y
    #trade x and y, then adjust height
    def _balance(self, node):
        if node is None:
            return node
    #if none,first node
        self._fix_height(node)
    #adjust height at node
```



```python
        balance = self._balance_factor(node)
    #determine difference between nodes
        if balance > 1:
            if self._balance_factor(node.left) < 0:
                node.left = self._rotate_left(node.left)
            return self._rotate_right(node)
        #if difference > 1, and left < 0, rotate left node.left
        #then rotate current node right
        if balance < -1:
            if self._balance_factor(node.right) > 0:
                node.right = self._rotate_right(node.right)
            return self._rotate_left(node)
        #if difference < -1, and right > 0, rotate right node.right
        #then rotate current node left
        return node
    def insert(self, root, key):
        if root is None:
            return Node(key)

        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        return self._balance(root)
    #on insert if none, add root
    #if key less than root, insert left recursive; else same right
    #return balance of root
    def search(self, root, key):
        if root is None or root.key == key:
            return root is not None

        if key < root.key:
            return self.search(root.left, key)
        else:
            return self.search(root.right, key)
    #if empty root, replace with new key
    #if less than root, move left, if higher move right
```



```
    def insert_key(self, key):
        self.root = self.insert(self.root, key)
    #call insert, start at the root with new key
    def search_key(self, key):
        return self.search(self.root, key)
    #call search start at root with key in question

# Example usage
avl_tree = AVLTree()
elements = [30, 20, 40, 10, 25, 35, 50]
for elem in elements:
    avl_tree.insert_key(elem)

print(avl_tree.search_key(25))
print(avl_tree.search_key(15))
```

```
True
False
```

Because AVL trees are a form of BST and implement a shortening process, they are a great tool for reducing search times; especially where the indices can be ordered numerically. The already-powerful order of BSTs is exacerbated when the length of left and right branches is equal. With heaps, BSTs and AVLs under your belt you should be able to pick up other types of trees without too much difficulty. For our next data structuring paradigm we will discuss **Hash Tables.**

Hash Tables: arrays^arrays…

When your data gets more complex and you don't have time for all the filtering, it's time to implement a hash table. Hash tables are an interesting approach because they allow for a ton of variability. Essentially data is stored in an array of potential arrays. The data inside each index of the associated array is a key: value pair. This allows for storage of various data-types: string, num, float, boolean, and even more arrays (potentially with their own arrays). The point is hash tables are an adaptive solution. They do have their issues though as well, including; sizing, complexity, and collisions.

In the following example, for each key value pair, the key is run through a hashing function and indexed based on the output. "Apple" becomes 12345, 12345 is a large number that makes no sense in a small array of 10 indices. The 12345 is then adjusted with the modulo and the size of the table, to be placed at index = 8. Whenever another value is stored there, you have a collision. To handle the collision your data is appended to the associated array at index=8. It then can be accessed by using the get() function.

Python example:

```python
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [None] * size
#initialize table attributes, size determined in call
    def _hash_function(self, key):
        return hash(key) % self.size
#use pythons hash() to convert key, adjust per size of table
    def insert(self, key, value):
        index = self._hash_function(key)
        if self.table[index] is None:
            self.table[index] = []
        self.table[index].append((key, value))
#insert via hash indexing, if None change to [] and append
    def get(self, key):
        index = self._hash_function(key)
        if self.table[index] is not None:
            for stored_key, value in self.table[index]:
                if stored_key == key:
                    return value
        raise KeyError(f"Key '{key}' not found")
#send query through hash function and search for index

hash_table = HashTable(size=10)
#hash_table object of size 10
```

```python

hash_table.insert("apple", "a fruit from the apple tree")
hash_table.insert("banana", "a fruit from the banana tree")
hash_table.insert("grape", "a fruit from the grape vine")
#use a key to store some info.

print(hash_table.get("apple"))
print(hash_table.get("banana"))
print(hash_table.get("grape"))
#get the associated data
```

```
a fruit from the apple tree
a fruit from the banana tree
a fruit from the grape vine
```

Hash mapping with a good collision handling approach is an effective solution to storage of complex data. It can dramatically reduce search, insert and delete times due to its direct indexing and lookup approach.

Our final data structure on the list is one used for managing complex data that may be interrelated. It uses a tool that has been used for modeling relational data for a very long time: the graph.

Graphs: the plot thickens

Graph data structures are a great approach to linking data points. Made up of vertices and edges, graphs are able to be simply modeled using Python's built-in hash table: the dictionary. Through a process of linking the keys to the associated arrays of other keys, each key can be related to any number of other keys.

The most notable and easily understandable implementation of graph data structures is social media. The main node is the user. From the user, secondary nodes: friends, pages, photos, posts etc. From the secondary nodes relationships between other users and their secondary nodes. This leads to a fairly simplistic means of mapping complex relationships.

For introductions, here are two examples of very basic graphs; one **undirected** and one **directed.** Undirected graphs are mapping all of the relationships between a number of nodes with no concern for directions, while directed graphs simply draw lines.

Undirected Python example:

```python
class Graph:
    def __init__(self):
        self.graph = {}
#initialize graph class
    def add_node(self, node):
        if node not in self.graph:
            self.graph[node] = []
#if new node, add the key of 'node,' with blank array
#if not, do nothing
    def add_edge(self, node1, node2):
        if node1 in self.graph and node2 in self.graph:
            self.graph[node1].append(node2)
            self.graph[node2].append(node1)
#if both nodes are present append each to corresponding arrays
    def __str__(self):
        return str(self.graph)
#convert self.graph to human-readable format
my_graph = Graph()
#instantiate graph object
```

```
my_graph.add_node('A')
my_graph.add_node('B')
my_graph.add_node('C')
my_graph.add_node('D')
# Add nodes

my_graph.add_edge('A', 'B')
my_graph.add_edge('B', 'C')
my_graph.add_edge('C', 'D')
my_graph.add_edge('D', 'A')
# Add edges
print(my_graph)
```

```
{'A': ['B', 'D'], 'B': ['A', 'C'], 'C': ['B', 'D'], 'D': ['C', 'A']}
```

Directed Python example:

```
class DirectedGraph:
    def __init__(self):
        self.graph = {}
#initialize graph class
    def add_node(self, node):
        if node not in self.graph:
            self.graph[node] = []
#if new node, add the key of 'node,' with blank array
#if not, do nothing
    def add_edge(self, from_node, to_node):
        if from_node in self.graph and to_node in self.graph:
            self.graph[from_node].append(to_node)
#draw relationship between from_node and to_node
    def __str__(self):
        return str(self.graph)
#generate human-readable
```

```
my_directed_graph = DirectedGraph()
# Create a directed graph

my_directed_graph.add_node('A')
my_directed_graph.add_node('B')
my_directed_graph.add_node('C')
my_directed_graph.add_node('D')
# Add nodes

my_directed_graph.add_edge('A', 'B')
my_directed_graph.add_edge('B', 'C')
my_directed_graph.add_edge('C', 'D')
my_directed_graph.add_edge('D', 'A')
# Add directed edges
print(my_directed_graph)
```

```
{'A': ['B'], 'B': ['C'], 'C': ['D'], 'D': ['A']}
```

When we are manipulating data we have options, and many more options than we have here. However, I think it would be inappropriate to go any further without diving into algorithms as well. For that reason we will end here and pick it back up in another blog.

Thanks for reading!

Find more at: chrismoulton.dev