Welcome, Data Structures

Introduction

Roberto Reynoso
Date Started 11/17/2021
Date finished 12/12/2021

Contact

reynoso.roberto.v@gmail.com

Stack

Introductions

Stacks within Python work very similar to how we interact with stacks of items in real life. You can imagine having to sort a stack of books. You would probably have to remove the book on the very top first to get to the next one and so fourth until there are no books left. We would call this proces "Last In - First Out" (LIFO), the reverse of this process would be called "First In - Last Out" (FILO).

A stack is a linear data structure that stores items in either (LIFO) or (FILO).

Functions

Functions used to implement stacks, depend on the method of Implementation you are using.

Implementation using list:

- *append()* - to add elements.
- *pop()* - removes the element in (LIFO) order.

Implementation using collections.deque:

- *deque()* - from collections import deque set stack = deque().
- *append()* - to add elements.
- *pop()* - removes the element in (LIFO) order.

Implementation using queue module:

- *maxsize* - max number of items allowed in the queue.
- *empty()* - If the queue is empty Return True, otherwise Return False.
- *full()* - If the queue has maxsize items, Return True. If maxsize=0, It'll never Return True.
- *get()* - Removes an item from the queue and Returns it. If the queue is empty, it won't Return until an item is available.

- *get_nowait()* - It will Return and item once one is immediately available, else raise Queue Empty.
- *put(item)* - Put an item into the queue. Will wait if the queue is empty until there is a free slot, before adding the item.
- *put_nowait(item)* - Without blocking, put an item into the queue.
- *qsize()* - return the queue's size (num of items in it). If there is no free slot immediately available, raise QueueFull.

Implementation using singly linked list:

- *getSize()* - Get the number of items in the stack.
- *isEmpty()* - If the stack is empty Return True, otherwise Return False.
- *peek()* - Return the top item in the stack. Raise an exception, if the stack is empty.
- *push(value)* - Into the head of the stack, push a value.
- *pop()* - Remove and return a value in the head of the stack. Raise an exception, if the stack is empty.

Implementation and Examples

There are multiple ways to implement a stack in Python.

Using list

You can use list objects to create stacks, which can push using append() and pop() to remove the top item from the stack. The biggest issue in using this method is speed issues, which can hold it back if the stack continues to grow and grow.

```
stack = []

# To push an element in the stack
# Use append()
stack.append("dog")
stack.append("cat")
stack.append("bird")

print(f"Stack: {stack}")

# (LIFO) order
# Use pop() to remove from stack
print("\nElements removed from the stack:")
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

Using collections.deque

The syntax for using collections.deque over lists is exactly the same, but you have to import the library and have the variable equal to a deque() type. This is the preferred method over lists, since it is faster. It provides a O(1) speed over an O(n) speed. You will then just use the same methods as you would with lists, to form your stack in (LIFO) order.

```
from collections import deque

stack = deque()

# Pushing items in the stack
stack.append("dog")
stack.append("cat")
stack.append("bird")

print(f"Stack: {stack}")

# Removing Items
print("\nElements removed from stack:")
print(stack.pop())
print(stack.pop())
print(stack.pop())

# checks if the stack is empty
if not stack:
    print(f"\nStack after elements are removed: {stack}")
```

Using queue module

The queue module also has a (LIFO) order queue, which is basically a stack. The put() function inserts the data into the queue and the get() takes data out from the queue.

```
from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize=3)

# qsize() show the number of elements
# in the stack
print(stack.qsize())

# put() is to push elements in the stack
stack.put("dog")
stack.put("cat")
stack.put("bird")

print(f"Full: {stack.full()}")
print(f"Size: {stack.qsize()}")

# get() remove elements from the stack
print("\nElements popped from the stack")
print(stack.get())
print(stack.get())
print(stack.get())

print(f"\nEmpty: {stack.empty()}")
```

Using singly linked list

The singly linked list has two methods that are suitable to implement a stack, which are addHead(item) and removeHead(). This will run in constant time.

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


class Stack:
   def __init__(self):
       self.head = Node("head")
       self.size = 0

   # String representation of the stack
   def __str__(self):
       cur = self.head.next
       out = ""
       while cur:
           out += str(cur.value) + "->"
           cur = cur.next
       return out[:-3]

   # Get the current size of the stack
   def getSize(self):
       return self.size

   # Check if the stack is empty
   def isEmpty(self):
       return self.size == 0

   # Get the top item of the stack
   def peek(self):
      # are we peeking an empty stack
       if self.isEmpty():
           raise Exception("Peeking from an empty stack")
       return self.head.next.value

   # Push a value into the stack.
   def push(self, value):
       node = Node(value)
       node.next = self.head.next
       self.head.next = node
       self.size += 1

   # Remove a value from the stack and return.
   def pop(self):
       if self.isEmpty():
           raise Exception("Popping from an empty stack")
       remove = self.head.next
       self.head.next = self.head.next.next
       self.size -= 1
       return remove.value
```

```
#  To Run the code
<p class="mume-header " id="to-run-the-code"></p>

if __name__ == "__main__":
    stack = Stack()
    for i in range(1, 11):
        stack.push(i)
    print(f"Stack: {stack}")

    for _ in range(1, 6):
        remove = stack.pop()
        print(f"Pop: {remove}")
    print(f"Stack: {stack}")
```

Problem to Solve

Create a program that will ask the user to add items to the stack and then will prompt the user to remove a number of items of their choice from the stack and then return and print the stack.

Solution

Set

Introduction

Sets are another way to keep data/items within a container. The properties of a set are that, Order does not matter, Contains unique elements, and Contains no duplicate values.
Some facts about sets include that "elements of a set" are reffered to as a "member". A venn diagram can be used to explain how sets function, the middle is what two sets could have in common.

Sets vs Lists, similarites are that they are both mutable (can be changed after its creation), collect single elements.
Differences, are that sets have no index, sets are not a sequence.

Sets vs Dictionaries, similarities are that they are both unordered collections and are optimal for "lookup" operations.
Differences, are that sets have no key-value pairs, set "look-up" is a test for membership. "True or False: does it exist in the set?"

Extra information, membership testing will tell us how sets are related to eachother. Euler's Diamgram can be used to show how a subset are the middle of a superset. If two sets have nothing in common they are called disjoint sets. Venn's diagram, can be used to show how a three subsets are within a superset. The superset is called the Union. The inersection is the inner part, which is what is incommon between the two sets. Where it overlaps is called the difference, which is what is unique

and not what is incommon between the two sets, so the outsides, which in Python the left most set. Lastly, the Symmetric difference is equivalent to the Union minus the Intersection.

Functions

- *add(element)* - Add one element to a set.
- *update(iterable)* - Add many elements to a set.
- *clear()* - Remove all elements from a set.
- *remove()* - Remove an element from a set; it must be a member. If the element is not a member, raise a KeyError.
- *discard()* - Same as remove() but does not raise an error.
- *pop()* - Remove and return an abitrary set element, Raises KeyError if the set is empty.
- *copy()* - Make a copy of a set.

Implementation and Examples

Basics of sets

```
# Creating a set
my_set = {}

# Sets - order does not matter
print({1, 2, 3} == {3, 2, 1}) # True

# Lists - order does matter
print([1, 2, 3] == [3, 2, 1]) # False

# Only unique values will be printed
print({1, 2, 3, 4, 1, 2, 3, 4, 3, 2, 1, 4, 1, 2})

# What will be printed?
fours = {4, 4.0} # 4 will be printed out, since Python evaluates our code from left to right
# This prints?
challenge = {4.0, 4, 5.0, 5} # 4.0, 5.0

# Sets print mixed each time
# Sets can only contain immutable
twos = {"two", 2, 2.22, ("dos", "Spanish")}

# Print booleans
booleans = {True, False}

# Print(none_set)
none_set = {None}

# Unhashable types

print({ [1, 2, 3] }) # unhashable type list
print({ {'hello': 'world!'} }) # unhashable type dict
print({ {'hello sets!'} }) # unhashable type set
```

Remove duplicates from a list

```
uniques_list = list(set(duplicates_list))
sorted_uniques_list = sorted(set(duplicates_list))
```

Subtract the elements of one list from another

```
def subtract_lists(list_1, list_2):
    subtracted_list = set(list_1).difference(list_2)
    return sorted(subtracted_list)
# or
def subtract_lists(list_1, list_2):
    subtracted_list = set(list_1) - set(list_2)
    return sorted(subtracted_list)
```

Sets are Optimized For Membership Testing

```
import timeit

test_range = range(1000, -1, -1)

list_ = list(test_range)
set_ = set(test_range)

list_time = timeit.timeit('0 in list_', globals=globals())
set_time = timeit.timeit('0 in set_', globals=globals())

print('0 found in list after', list_time, 'seconds!') # 0 found in 7+ seconds
print('0 found in set after', set_time, 'seconds!') # 0 found in under 0.03 seconds

# Sets are faster, so they are more optimal for Membership testing
```

Problem to Solve

Create a program that will prompt the user to add and remove from atleast two different sets and then compare the sets to find the unique values.

Solution

Tree

Introduction

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.

The circles are called nodes. The top node is called the root. The node above the left and right child nodes are called the parent. The nodes at the very bottom of the tree are referred to as the leaves.

You would count levels from (0 or 1) to the next level starting with the first root and so fourth. These levels are reffered to as depth.

Height level starts from the bottom, we can say that the row of the last nodes given are level one and then go up from there.

Typees of Binary Trees include, the Complete Binary Tree, which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
The Full Binary Tree, (someti,es referred to as a proper or plane binary tree) is a tree in which every node has either zero or two children.

Traversing a Tree (Implementation)

Binary Tree Traversals

Tree Traversal: Process of visiting (checking and/or updating) each node in a tree data structure, exactly once.

Unlike linked lists, one-dimensional arrays ect., which are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order.

There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.

Pre-order Traversal

- Check if the current node is empty/null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.

In-order Traversal

- Check if the current node is empty/null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.

Example

```
class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        # add inorder and preorder form here
        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal

# pre-order-> 1-2-4-5-3-6-7-8-
# in-order-> 4-2-5-1-6-3-7-8-
# post-order-> 4-2-5-6-3-7-8-1-
#             1
#          /     \
#         2       3
#        / \     / \
#       4   5   6   7
#                    \
#                     8

# Set up tree:
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)
tree.root.right.right.right = Node(8)
print(tree)

# in-order form
print(tree.print_tree("inorder"))
```

Problem to Solve

Make two more functions to return and print in, pre-order and post-order form. The output you are looking for is above in the Tree example within the comments (Use the Startout code).

Startout

Solution