Andrew Taylor
atayl136

Data Structures Lab 1 Analysis – Prefix to Postfix Conversion via a Stack

Introduction

The goal of this assignment was build a prefix to postfix expression converter from a stack without library functions and use it to process a provided input file and a file of test cases I created to test the program. I am new to programming and I am only able to do so in Python so that was my language of choice. This shaped my choices for the project as I will discuss later.

I took Python 101 with Johns Hopkins, and received an A in the course, but I don't see myself as a very good programmer. Having a read a book or two on the subject, I know there is so much more that can be done, and I feel I know just enough "to be dangerous". My method is simple. I don't use things if I don't need to, so I don't turn to things like polymorphism, and I don't employ classes unless I'm building out a lot of data objects. I tend to make a lot of functions and cobble together my programs until they work. I am unaccustomed to testing them against extreme cases and input errors so this project was educational for me. I apologize if it something of a Rube Goldberg machine inside, when you look.

I had to think for quite some time about many of the problems presented by the implementation along the way. I was also hung up on many little formalities of practical coding, but I was able to find the answers I needed. The resulting code is functional, and although it tends to lack dimension, it has taught how to think about the users experience and how to respond to different extreme cases.

The Data Structure

The data structure required here was a stack. Stacks can be implemented either by an array, in a linked or hybrid implementation. To make this choice, I looked at my options in Python. There are 4 basic data types in Python, and also there are strings. Sets wouldn't do because they are unordered. Dictionaries were inappropriate because stacks are not key/value pairs (for one), tuples don't fit the profile because they are immutable so they wouldn't truly be a stack, and strings can't be used because they don't form "buckets" for items to be pushed and popped off the stack. That left Lists. Stacks can be formed out of lists, so I chose to build my stack data structure based on that. This was justified because I can get all of the behavior of a stack out of a Python list, by treating as such, and ignoring certain list like capacity like intermediate insertions or deletions. I decided to treat the beginning of my list as the bottom of the stack, and the end of my list as the top of the stack, so that as the list grows and shrinks the action of pushing and popping would be properly reflected. In testing, the stack worked as expected.

To build out the ADT, I made some functions. Library functions were not allow so made simple functions to create the stack (create a list) and manipulate the stack. To push to the stack, I looked for a way to add to a Python list with just mathematical operators but there was none. I had to settle on a function so I chose .extend() – it extended my list by one data item. I was able to pop from the list with just slicing, however, and return the sliced value. One side effect of this approach was that the list could be sliced into nothing. Another function to check for an empty list was accomplished by checking the length and also the type (for None), and this was used in places to prevent popping from an empty list or NoneType list. Optional methods such as deep copy were considered but deemed unnecessary. I felt all of these choices were justified to avoid library functions and fit the needs of the application.

The data structure works well for the application precisely because the LIFO nature of the stack lends itself to the reversal of the prefix notation to a postfix notation. By storing some operands on the stack, popping them appropriately and joining them to an operator, the operators are naturally brought over while the operands stay consistently sequenced. This is due to the LIFO structure of the stack. This is why a stack makes sense as a data structure for this task.

Iterative Solution Versus Recursion

The stack, the conversion, the data processing, and the whole program in general makes heavy use of an iterative approach. The manipulation of the stack is iterative in that it is done with a for loop that considers the input string one character at a time. Iteration is used in other places in the program, there are a good many for loops involved. Considering them all, it is true that any of them could, perhaps, be replaced with a recursive solution but that wouldn't be the best option in my mind. Thinking through the problems presented, there isn't any clear and glaring recurrence relation that would persuade me to consider it. It is just one less character, one more string. Moreover, some compelling base case should exist that jumps to mind so that someone just hearing of the problem would know when the recursion should stop. "The end of the list" for a list of 10 items is reached quickly enough, but perhaps if you have an operation traversing 1000's of lines of a spreadsheet using a recursive line of code would be compact, appropriate, just better in that case.

It is my understanding that a recursive solution comes with a cost, it can build into a memory burden because it builds upon the previous step, saving each step in memory as separate entry on its way to some final answer. It is for this reason that Operating Systems come with built-in limits on recursive calls. It can be more desirable when each entry is an individual result that is desired. Some find recursive code to be more elegant, but if you allow it, I call it confusing. It makes the programmer want to look back up the page rather than continue reading down the page of code. This is my personal feeling. However, for some mathematical relationships, like the Fibonacci sequence, or Golden Ratio, it is indispensable. It is in those cases where I would feel the problem is naturally recursive and merits a recursive solution. It is for these reasons and under these conditions only that I find recursion to be preferable.

Design

To get a handle on design, I took a look at the sample project. I had never created a program like that before. In our previous class we always turned in one or two .py files so I started my concept of modularity from that basis. I knew I had a few tasks to perform: Take in the data, Process the data, Convert the data, Output the data, Provide Statistics. I decided to segment my program roughly along those lines so I ended up with 5 files: a main driver (which was also responsible for file I/O), a converter file, a process file, the Stack source code, and runtime statistics. These files fell into place naturally as I planned top-down. I did things this way because besides the implementation of the stack very few requirements were given.

The main file is where the program first begins so it is where I decided to read the input. I parsed and processed the input as dictated by the Programming Assignment Guidelines. After that it is built into strings character by character and then into a list of input strings. The process file is where I connected this input to the conversion, and the conversion to the output. The conversion file imports the stack source code directly. The main driver imports the runtime tools. Finally, the main driver organizes the above into a main() function and runs it, once, generating a report in the console and a log written to file. Generally I tried to keep it simple and yet efficient. This was the motivation behind my design decisions.

Efficiency

Stacks are among the simplest of data structures. In an array, the push function a single element is inserted at the last position. This takes a single memory allocation operation which is done in constant time. There is no auxiliary space, no extra space being used. Similarly, in a linked list only a new node is created and the pointer of the last node is updated. This includes only memory allocation operations and so insertion is done in constant time. No auxiliary space is being used. In an array implementation, the pop function is an arithmetic operation only, the top pointer is decremented by 1, a constant time function. No extra space is being used. In a linked list, the pop function deletes only the first node and the top pointer is updated in a constant time operation. No extra space is being used. In both cases, the peek function accesses a single memory location. It is in constant time with no extra space used. The empty check function is an arithmetic function, it is in constant time and uses no extra space. Similarly for checking the size of the stack. In all these cases the time complexity of a stack is O(1) and the space complexity is also O(1). My implementation does not differ from this in any respect.

Error Checking

At the outset of the project, I had intended to impress with liberal use of the try…except structure and a complicated system of custom exceptions to cover any and every error I could think of that could go wrong with the program. It quickly became clear that this was not going to be a satisfactory approach for me, as I began to deal with the simple case of acting on a non-existing stack. In the raw, this is a Python NameError. It hasn't been defined. I wanted it to be an exception of my own definition. After looking for the answer for a very long time I found it mentioned that Python doesn't allow this to be overridden at the module level, it can be, but inside a function, or only when the module loads. When a function is called on a non-existing stack, Python will revert back to its own message. This left a bad impression on me. But this shouldn't happen if my program is written correctly anyway.

It then occurred to me that the unavoidable errors anticipated in the application were of a data input in nature, so for that and the following reasons I decided to purely implement print statements warning the user rather than any try…except statements and exceptions:

1) Python does not allow overriding built-in exceptions at the module scope level. They are only triggered when loaded. That is, functions that call non-existing stacks will have Python's error not mine.

2) Exceptions are meant to halt a program, and I wanted to process as much of the file as possible for the user. It is possible to continue execution but I don't think that is in fact what was in mind with their creation, thus the traceback. With my selected approach the user is informed of the error but receives maximum output.

3) Print statements give just as much information when done correctly, because the point is the program functions save some problem that must be brought to the users attention.

4) Exceptions to me are more fitting to problems with the program features, requiring the attention of the programmer, than formatting details about the input for the end user. When was the last time you encountered an exception as an end user? This is a really bad sign when this happens, it means the program doesn't even work.

The errors of interest were common input errors: invalid characters, transposed characters, missing characters, things of that nature. I tried to build in a print statement (and thus one in the log) that would catch each of these scenarios, and account for any key or key combination the user could press on the keyboard. All things being equal it seemed like a simple question, either the character fit the required format or it didn't. Either the line fit a valid syntax, or it didn't. I couldn't think of many scenarios beyond that to test. Extreme cases were included in my test cases as required. There could be infinite ways for a user to "fix" an invalid input, so the best I could do is give them a general idea of why it didn't qualify. The errors I warned to the user included:

1) Invalid character (not a letter or operator)
2) Invalid syntax (too many operators)
3) Invalid statement (lack of either letters or operators)
4) Attempt to Pop from an Empty Stack
5) Extra Space in input string ignored

Enhancements

It was difficult for me to invent some enhancement to this project. In the end I took some inspiration from the sample project and built a module for runtime statistics. This simple module helped track the time the program takes, and the size of the files. I also started keeping track of the errors. The resulting report when printed to console and logged to file includes:

1) The date and time the converter was started
2) How many nanoseconds the program took to run
3) The total lines of input
4) The number of bad characters there were
5) The size of the input after it was read
6) The total lines of output, to compare
7) The lines converted to postfix without error
8) The size of the output before it was written to file

These were just some basic extras I could think of that may be of interest to the user. Other ideas I had included an infix converter, or a solver, but those ideas proved to be to tough to implement in the time I had. I hope to think about this harder for Lab 2 and do something a little more special. One of the challenges I've always felt about early programming is the constraint of print statements and text. I feel there's only so much one can do, so next time I might try to display a graphic, or do something fun.

Lessons Learned

I had a lot of fun with this project, although I did bang my head against the wall more than once. My lessons learned fall into three major buckets:

1) I don't understand Exceptions and Error Handling.
   If Python stops your program when there's a problem, why include your own exceptions that do the same? That means to some extent you've anticipated the problem. If it's programming related, you should fix it, and that should not occur. I need to read and review why and where to use exceptions in particular when they don't interrupt program execution because I find the traceback (and stopping to debug) the most useful part of them. Perhaps in this case they weren't needed but they are something basic I should know by now. I was left confused.

2) I'm stuck on functions.
   I got used to the basic tools of Python and have not taken enough time to experiment with all the language has to offer. This really amounts to some exploration and consideration of technical matters prior to mapping out an implementation. I consider this study outside of the scope of this course. Bottom line, I need to improve my skills with better organization and better techniques and I hope the feedback I receive will point me in the right direction.

3) Library functions were made because of certain problems I encountered.
   Doing things without the library functions made me realize why they are there. Suddenly finding a stack of NoneType was a surprise to me, for instance. I think it is good for me to start thinking in terms of these problems because that dynamic no doubt exists on higher paradigms and one day I'll have to solve something without a library where once I depended on one.

Starting from scratch, I would do some things very differently. First, I would not worry about the stack implementation. It was simple and there were not many ways that came to mind for me to do it. I thought a lot about it and perhaps I am missing something, but in Python you cannot add to a list without a function, and you cannot create an array without NumPy. I would spend more time improving the technique of the overall program, understanding classes again, looking up basic stuff like inner and outer functions, just trying to do something more Pythonic. What I did is very the same wrench turned over and over. I would ask some questions in the discussion forum (way ahead of time because they might go unanswered) about what would be best, bounce ideas off other students, and learn from them. I think I will try that for the future Labs. Thank you!