Recap + Supplementary for Practical 07

Ian Wern Han, Lim
lim.wern.han@monash.edu

20 September 2014

Abstract

This is a short recap and supplementary document for Practical 07. The goal of Practical 07 is for the students to have a practice with classes and text processing; both of which will be used for Practical 08 again. Supplementary Python files are also included with this file.

1 Author's Note

- Example codes are coded based on my personal programming style such as variable names using current and new. There is no right or wrong in programming styles and also no need to follow mine as it is meant to help my understanding as I have my own conventions to it.
- This document is written by myself personally as a extra help to supplement for your studies. Besides that, it will provide a good insight and overview of the practical you have done as it is more than just marks for you in those practicals. I am not bounded to consistently produce such document and penalize for the possible mistakes here. I will however try my best for this.

2 Abstract Data Types (ADT)

We have gone through ADT in the earlier practicals and from this practical, there are still students who struggle with it. Attached with this document is a Stack ADT implementation I have edited as reference (Stack.py).

2.1 Common Mistakes: Returning Errors/ Exceptions Messages

It is misleading to return error or exception messages as the caller function would interpret it as the returned value from executing the callee. For example, when it is coded for a pop() operation to return "Stack is empty" message, the caller would interpret the top of the stack contain "Stack is empty" before the pop() operation when the stack is empty itself. This do not adhere to the ADT behaviour of pop() to remove and return the top item of the stack. Instead, when the list is empty; an exception should be raised by the pop() operation.

2.2 Common Mistakes: Printing

The functions within ADT should perform functions as they are defined. For example, the function pop() in the Stack ADT should just remove the top item of the stack and return it out. There should not be a printing for that item or printing an exception caught (or any manually coded errors). The reasons are:

- A machine do not understand the prints. For example, a caller would not understand the error or exception printing of the callee function. Likewise the caller could only use values or objects returned rather as opposed to the values being printed to the user.
- Other scripts, class or functions which uses the Stack ADT might not require the printing and thus such prints are unwanted behaviours.

2.3 Common Mistakes: Privacy Leaks

For an ADT class, one should not be able to access the variables directly from another script, class or function (of another file). These variables should only be allowed to be accessed and modified through the accessors and mutators from the ADT class themselves. In python's convention, private methods are prefixed with double underscores (__) and private variables are prefixed with single underscores (_). The underscores are indicators to avoid invoking or accessing those private functions or variables.

2.4 Weak Understanding: Exception Handling

Overall, the exception handling is still weak where often every exception is caught by a single except instead of targeted ones. Exception handling is also handled within the callee itself instead of letting it be handled by the caller with

the callee raising the exception. A way for exceptions to be handled is shown in Figure 1.

Do note that assert (in the lecture notes) will raise an assertion exception which need to be handled with a try-except.

```
    def pop(self):
        """
        The correct pop implementation
        """
        if self.is_empty():
            raise Stack.StackEmptyError("The stack is empty")
        self._theTop -= 1
        return self._theStack[self._theTop+1]

def test_pop():
    print("=======================================")
    print("Testing Stack.pop().")
    current_stack = Stack(10)
    try:
        print("Popping empty stack")
        print("Item: " + str(current_stack.pop()))
    except Stack.StackEmptyError as error:
        print("Exception: " + str(error))
    except:
        print("Unwanted Exception Found")
    try:
        current_item = 1
        print("Pushing item: " + str(current_item))
        current_stack.push(current_item)
        current_item = 2
        print("Pushing item: " + str(current_item))
        current_stack.push(current_item)
        print("Popping")
        print("Item: " + str(current_stack.pop()))
        print("Popping")
        print("Item: " + str(current_stack.pop()))
    except Stack.StackEmptyError as error:
        print("Exception: " + str(error))
    except:
        print("Unwanted Exception Found")
```

Figure 1: Example for exception handling

3 Other Practical 07 Mistakes and Comments

These are just some of the other mistakes which caused marks lost:

- Documentation. Documentations are still incomplete. Most of them are due to last minute work. As per the marking guide, if documentations are lacking; you will be given 0 marks regardless of how well is the code.
- Question reading. There are many who often missed out some key parts of the question. For example in Practical 07, it is written in bold for test functions to be written. Some did not have them, some wrote test cases only instead.
- Testing. As many of you have learnt test functions. It is advised to use be using automated test functions from now on. They do not need to be complicated ones (the standard). To read more about test functions (complex ones which some of you are in fact using), have a look at http://pymbook.readthedocs.org/en/latest/testing.html. Besides that, do test your code properly. Many did not test the codes well enough that I am able to discover mistakes and even un-handled exceptions. There is no point in making weak test functions that do not test the code well enough.
- Codes are not modular (Modular code). I have noticed many of you do not separate your code into functions and what worse is to have everything within the menu function. Such programming practice is very unhealthy especially when it comes to complexity analysis. The right way to do it is to separate codes into functions, allowing you to perform easier complexity analysis on those smaller code chunks. As the menu will now only call out the functions, there is no need for the menu to have complexity anymore.
- Complexity analysis. There are still some who assume the best case for the input to be a small value. Do remember (mid-term test question 5 as well) that you are doing complexity analysis for a general size N input. Thus the best case is not when the size is small. Besides that some single line code do not have a O(1) complexity as it could be Python's short hand writing. An example of this is the initialization of the array in the Stack in practical 07.