FIT1008 Introduction to Computer Science (FIT2085: for Engineers)

Interview Prac 2 - Weeks 8 and 9

Semester 2, 2019

Learning objectives of this practical session

To be able to implement and use basic container classes in Python.

Important requirements

For this prac, you are required to:

- **Create a new file/module for each task named `task[num].py` (that is, `task1.py`, `task2.py`, and so on).**
- **Provide documentation for each basic piece of functionality in your code (see below)**
- **The documentation needs to include (in the docstring):**
  - **pre and post conditions**
  - **Big O best and worst time complexity**
  - **information on any parameters used**
- **Provide testing for those functions/methods explicitly specified in the questions. The testing needs to include:**
  - **a function to test it**
  - **a comment indicating the purpose of each test**
  - **at least two test cases per test function. These cases need to show that your functions/methods can handle both valid and invalid inputs.**
  - **assertions for testing the cases, rather than just printing out messages**

  Your tests should be added to the corresponding `test_task[num].py` file
- **VERY IMPORTANT: The code cannot use any of the Python list or string functionality that would make the job trivial (so no slicing, append, prepend, str, len, eq for lists, and `__contains__` for strings, etc). However, you may:**
  - **Track the capacity of your internal array using len (as we have in the `ListADT` skeleton).**
  - **Use str, eq, etc. on *members* of the list (e.g. when implementing `__str__` or `__eq__`).**
  - **Use string methods (e.g. `startswith`) for command processing (determining that `"read small.txt"` is a `read` command with argument `"small.txt"`).**
  - **Use any functions/methods you wish in your *testing* code.**

Supporting Files

To get you started, you will find some supporting files on Moodle under Week 8:

- `ListADT.py`, `Editor.py`: Skeleton files for the `ListADT` and `Editor` classes, which you may use as starting points.
- `test_prac2.py`, `test_task[num].py`, `test_common.py`: Testing harnesses for each of the tasks.
- Additional text files used by the testing harness.

Your code will be tested with an extended version of this harness, so **make sure your code can run inside the harness**. To make the testing work, we have made assumptions about the naming of instance variables, so please follow the naming in the skeletons.

The harness you have been provided is not exhaustive, so don't rely on it to ensure correctness! Remember to add your own tests, as well.

Important: Checkpoint for the end of Week 8

To reach the check point you must complete Tasks 1, 2 and 3. Please remember that reaching the checkpoint is a **hurdle** and you will get a 0 if you do not reach it (read the pracGuide document for more details).

Task 1 [9 marks]

Consider an array-based implementation of a List ADT, as seen in the lectures. In this task you will need to re-implement this array-based List ADT using a class `ListADT` that contains the methods included below. The supplied `ListADT.py` contains a skeleton of the `ListADT` class you need to implement with some (glaringly uncommented) methods (like `is_empty(self)`, `is_full(self)`). The second file is `TestList.py`, which contains code useful to test all `ListADT` class methods (those already given in the skeleton and those you need to define).

To complete this task, you need to add comments to the provided `ListADT` methods, and implement the following ones (properly commented) inside the `ListADT` class (note that the testing for these methods is already provided in `TestList.py`):

- `__init__(self, size)`: Initialises the appropriate instance variables (have a look at `ListADT.py` or change them, if you prefer new names). Please define it in such a way you allow users to provide the size upon creation and, if they do not provide it, you give a reasonably big default.
- `__str__(self)`: Returns a string representation of the list (returns the empty string if the list is empty). Structure the string so that there is one item per line. Recall that this method is the one called by `str(self)`
- `__len__(self)`: Returns the length of the list (that is, the number of elements in the list; 0 if empty). Recall that this method is called from Python by calling `len(self)`
- `__getitem__(self, index)`: Returns the item at `index` in the list, if `index` is non-negative. If it is negative, it will return the last item if `index` is -1, the second-to last if `index` is -2, and so on up to minus the length of the list, which returns the first item. The function raises an `IndexError` if `index` is out of the range `-len(self)` to `len(self)-1`. Recall that this method is called from Python by calling `y = self[index]`
- `__setitem__(self, index, item)`: Sets the value at `index` in the list to be `item`. The `index` can be negative, behaving as described above. Raises an `IndexError` if `index` is out of the range `-len(self)` to `len(self)-1`. Recall that this method is called from Python by calling `self[index] = value`
- `__eq__(self, other)`: Returns `True` if this list is equivalent to `other` (that is, they have exactly the same elements in the same order). Recall that this method is called from Python by calling `self == other`
- `insert(self, index, item)`: Inserts `item` into `self` at position `index`. For example, `insert(self, 0, item)` inserts the `item` in the first position, shuffling back everything else. As before, the `index` can be negative. `IndexError` if `index` is out of the range from `-len(self)-1` to `len(self)`. Hint: Note the range here is slightly different to `__setitem__`. Be especially careful with negative indices: `self.insert(len(self), item)` and `self.insert(-1, item)` should both have the effect of appending `item` to the end of the list (which differs slightly from `insert` for native lists).
- `delete(self, index)`: Returns the item at `index` from the list and deletes it, shuffling all items after it towards the start of the list. The `index` can be negative, behaving as described above. Raises an `IndexError` if `index` is out of the range from `-len(self)` to `len(self)-1`.

Task 2 [2 marks]

Modify your list implementation so that the size of the underlying array can change. That is, if the list becomes full, the size of the array is increased to be 1.9 (rounding up) times the current size. Likewise, the size of the array should decrease by half if the length of the list is less than $\frac{1}{4}$ of the size of the array (i.e., if the elements in the list occupy less than 1/4 of the available space). But make sure the size of the array is never less than 40 (you need to check this upon creation for the user's value and your default value, and also whenever the size decreases). Also, when resizing the list (both increasing and decreasing), make sure you retain the contents of the list, that is, the elements of the old list are copied into the new one in the same order as they were before.

You are allowed to copy and paste any methods from Task 1 that are not affected by the resizing.

Important: make sure you do update the tests of the methods that are affected by the capacity with at least two new tests per such method.

Background for the rest of the prac

The editor `ed` was one of the first editors written for UNIX. In this prac we will use the list class you implemented in the previous tasks to implement a version of a line-oriented text editor based on `ed`. The text editor `ed` is very similar to the common UNIX text editors `vi` and `vim` (it is in fact their predecessor). [1] **However,** our commands will be different from the `ed` commands.

To implement a simple line-oriented text editor, the idea is as follows. Suppose a text file called `small.txt` contains the following three lines of text:

```
Yossarian decided
not to utter
another word.
```

where the string "Yossarian decided" is considered to be in line 1, "not to utter" is considered to be in line 2, etc. We want to store every line in the file in a data type that allows users to easily manipulate (delete/add/print) any line by simply providing the line number they want to modify. This means we should use a list data type (as opposed to a stack or a queue).

Documentation for Python string functions can be found at `https://docs.python.org/3/library/stdtypes.html`

To properly test your code you should use very large text files. There are plenty of online resources that can be used to download files such as e-books. As example, here is one: `http://www.textfiles.com/etext/`.

Task 3 [3 marks]

Implement a function `read_text_file(name)` that takes the `name` of a text file as input and reads each line of text into a string, returning a list of strings associated to the file (i.e., the function converts the text in the file into a list of strings). Note that you should use the `List` ADT implemented in Task 2 for this, that is, the list of strings returned by the function should be an object of the `ListADT` class implemented in Task 2. Test the function using the `small.txt` provided above in the Background, and a similarly small file created by you. Please make sure you close the file after reading it (and verify this by checking the `closed` attribute of the file object). For information on how to read data from a file, please read the tutorial found at `https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files`.

Important: you need to test the `read_text_file(name)` function with a small file containing a few (3+) lines.

CHECKPOINT

(You should reach this point during week 8)

Task 4 [6 marks]

Write a class `Editor` that allows a user to perform the operations shown below using an `ed`-style command prompt. The class should have an instance variable called `text_lines` containing a list of strings (corresponding to the list of text lines being edited), where the list is an object of the `ListADT` class implemented in Task 2. Please implement your interface so that the user enters commands as text (e.g. `print 8`), as opposed to, say, a numeric menu. Also, make sure each command (except `quit`, which is trivial) is implemented by a method with the command's name which takes as argument the string read from input (after taking the command part off the string). For example, command `print`

[1] If you want to find out more about `ed`, you can read the man page (by typing: `man ed` into a linux or MacOS X terminal), or visit, for example: `http://roguelife.org/~fujita/COOKIES/HISTORY/V6/ed.1.html` or `https://youtu.be/BNYpmLH6IjQ` (YouTube tutorial)

`num` will be implemented by `print_num(self,rest_string)` and `delete num` will be implemented by `delete_num(self,rest_string)`. For the `insert num` command we will instead call method `insert_num_strings(self,number,list_of_strings)`, which has a second argument the integer representing `num` and as third argument the list of strings the user gave you to insert, if any. This will help you (and us) test your command implementations. You must add an appropriate test function for the `delete_num` and `insert_num_strings` methods, with a few meaningful test cases.

Hint: An example of user interaction, corresponding method calls, and program responses is shown at the end of the prac sheet.

read filename: which basically calls the function defined in Task 3.

print num: which prints the line of text at position `num`, and if no `num` is given prints all the lines. Note that what the user refers to as line 1 is what `text_lines` stores in position 0. Thus, you need to convert the numbers given by the user to the indices used by your array.

delete num: which deletes the line of text at position `num`, and deletes all the lines if no `num` is given. Same issue as above for `num`s.

insert num: which inserts immediately from position `num` onwards, any line of text subsequently written by the user in the order entered, until the user types a full stop on a line by itself. Note that full stops that are not on a line by themselves should not terminate the input. Do not store the full stop in the list. For example, if the user types `insert 3` and then types lines

Hello
I am well. How are you?
.

then string "Hello" will be added to `text_lines` at position 2 (remember it starts at 0, so Hello becomes the third line in the list), and string "I am well. How are you?" will be added at position 3, with all items in the `text_lines` that used to appear from 2 onwards, now appearing from 4 onwards.

quit: which quits the program.

Important: All errors should be caught and a question mark, `?`, should be printed when an error occurs, so that the user is given the opportunity to provide correct input data. Negative number lines are possible and should be handled following the convention described for the original list implementation. For example, *num = -1* refers to the last line of the text, and inserting from *-1* should append. Also note that 0 is *not* a valid line number, and should be treated as an error.

Tip: When you are testing your code, it is handy to have a source of reasonably large text files that you can use as test data. There is a huge repository of public domain text files at Project Gutenberg's websites. The Australian Project Gutenberg repository is at `http://gutenberg.net.au`. You are encouraged to download a couple of ebooks from here and use them to make sure your code can deal with large files. Be sure to download plain-text format, though – your program is not expected to deal with other formats.

Task 5 [2 marks]

Adds the following new command to the `Editor` interface:

`search string`: which first calls a method `search_string(self, string)` that returns a list with all the line numbers (if any) in which `string` appears, and then prints the elements of the list.

Make sure you test the `search_string(self, string)` method, and catch all errors printing a question mark when an error occurs, as before.

Important: Make sure you do not use inbuilt string methods for doing the searching (such as the __contains__ method). You must implement the search yourself. You may chose to use an iterator.

Task 6 [3 marks]

The sequence of actions made in a text editor can be stored during execution to facilitate undoing actions. When a user chooses to undo an operation the most recent successful `delete` or `insert` action should have

its changes reversed This indicates that a last in, first out data structure would be suitable for storing the actions.

Implement a `Stack ADT` of your choosing (using linked nodes or an array) and use it to implement in your editor the *undo* feature. Add this to your editor using the command `undo`.

Important: to reduce the time taken for this prac, you do not need to comment or write test cases for your stack ADT (and can copy and paste the one from the lectures). Also, think about what you need to store in the `Stack` in order to be able to undo commands these two commands, and try not to store more than you need, or you will run out of memory... **Note:** Inserting zero lines (e.g. `ed.insert_num_strings("0",[])`) is still a successful action.

Example: Editor interaction

Below we give a brief example of the expected behaviour of your `Editor`: the text the user types, the corresponding method call, and the `Editor`'s response.

user command	**method call**	**Editor output**
	`ed = Editor()`	

`insert 0`
`some text.`
`.`
	`ed.insert_nm_strings("0", ["some text."])`	

`print`	`ed.print_num("")`	`some text.`
`read small.txt`	`ed.read_filename("small.txt")`	
`print 2`	`ed.print_num("2")`	`not to utter`
`delete 3`	`ed.delete_num("3")`	
`delete 30`	`ed.delete_num("30")`	`?`

`insert 2`
`this is`
`some input.`
`.`
	`ed.insert_num_strings("2", ["this is",`
	`" some input."])`	

`print 3`	`ed.print_num("3")`	`some input.`

Note: Remember that the second argument to `insert_num_strings` should be a `ListADT`.