Project 3: A List ADT

Duration: 6 hours
Reminder: Read through this entire project description before you begin working on it.

Introduction

Up until now the projects you have been completing have focused on introductory algorithms (searching and sorting). With this project we take our first look at implementing a data structure as part of an abstract data type (ADT). In previous projects you have utilized the built-in Python list class. This class offers us a convenient and efficient ADT that can be used to store any kind of data we would like, and to easily perform numerous useful operations (such as searching and sorting) on that data. In this project you will create your own list-type ADT called SList (short for Sorted List), and mimic several of the capabilities of the built-in list class. *It should go without saying – but to be clear, you are NOT allowed to leverage Python lists in order to implement SList* 🤨

Completing this project will help you to recognize the differences between an ADT and a data structure, strengthen your algorithm design and analysis skills, familiarize you with linked lists, develop your understanding of dynamic type systems, and enhance your object-oriented programming abilities.

"Begin with the end in mind": As always you should read through this entire document before beginning your project, as having a complete picture of where you are trying to go in the end will help you to make good decisions along the way.

Part 1: SList

Your principal objective for this project is to create SList, an ADT capable of storing lists of information in ascending order, and performing basic tasks on that information (such as inserting, searching, removing, etc.). Ultimately SList must be able to perform these operations on ANY data type that meet certain minimal requirements – but thanks to Python's dynamic type system we can do much of the work by initially thinking about a single data type. For simplicity's sake let's start with ints. With this decision in mind your objective for part 1 is to create a sorted list class that can store a list of integers in ascending order.

To help you get started we are providing you with template/skeleton code in the files which lists the methods that you will need to implement as a bare minimum (in truth you will almost certainly need to add at least one or two helper functions in order to complete the project – don't be afraid to do so). These files are not meant to be perfect in any way, they are merely intended as a starting point and you are not required to use them, but it is highly recommended. Also – please note that you may need to make improvements to the code/comments you find within these files in order to satisfy pylint. Again – don't be afraid to make changes as needed. ***If you choose to utilize***

these files you will need to rename them appropriately to meet the project requirements before submitting.

Step 1 – The SListNode Class (done for you)

As a convenience for you we have already created an inner/nested class for you called SListNode. You are free to modify this class as you see fit, but it should remain an inner class to the SList class. Note that this kind of encapsulation is great for readability, maintainability, and portability.

Step 2 – The SList Class

The SList class is the most significant piece of this project. As mentioned above – you would be wise to implement this class as though it will only ever store int values. This will make it easier to think about, and we will see that thanks to dynamic typing – will be enough to make it work in the general case as well if we are just a little bit careful.

SList must implement the following methods:

- **insert(value)** : value is inserted in the list at the appropriate location to maintain a non-decreasing ordering. If two values are identical, the newer value should appear AFTER the older value (i.e. ascending order of arrival time). For instance if the list contains the values 1, 5, 6, 9 and an additional 6 is inserted, then the list would contain 1, 5, 6, 6, 9 and the second 6 would be the newly inserted 6. For simple integers this doesn't matter – but for more complex objects it might. This function does not return a value
- **remove(value)** : Removes the first occurrence of value from the list. Returns True if a value was found and removed, False if not.
- **remove_all(value)** : Removes ALL occurrences of value from the list. Doesn't return a value.
- **size()** : Returns the current number of values/nodes in the list.
- **find(value)** : Searches for the first occurrence of value in the list. If found, the value is returned; returns None otherwise.
- **__str__()** : Returns a string representation of the contents of the list. This string should start with '[' and end with ']', and the string representations of the list items should be comma separated, and listed in order. Note the somewhat recursive nature of this method as it will need to call __str__() on the individual list items…

We recommend implementing these methods and creating simple unit tests on your own to validate their behavior.

Step 3 - Make it Iterable

In addition to the methods listed above, your SList class must support several of the "goodies" that Python lists make available to us. The first of these is that your SList class must be iterable so that it can be used in for loops etc. as we are accustomed to doing with "regular" Python lists. For a class to be iterable it must implement or otherwise provide access to the following methods:

- __iter__() : Returns an iterator object *(something that has implemented __next__())*
- __next__() : Returns (or yields) the next value in a sequence – in this cas the next value in the list. *(may not be explicitly necessary if you choose to implement a generator function)*

Step 4 – Show us the Brackets!

Being able to access an item at a give position in a list is an extremely useful capability. You must extend SList to support position based indexing via the square bracket notation we are used to (e.g. theList[3]). To do this you must implement the __getitem__() magic method as follows

- __getitem__(index) : Returns the value stored at position index in the list.

Once again we suggest that you implement these methods and test your code to validate its performance. The is_sorted() function provided in the supplied p3_main_template.py file may be useful to you at this point.

Part 2: The Course Data Type

As mentioned above, MyList is supposed to be able to operate on any data type that meets certain minimum requirements. In this part of the project, you will create a custom data type called Course (store it in a module called course.py) that meets these requirements; and observe/demonstrate that SList operates as expected in conjunction with this class.

Step 1 – Course

Course is an ADT intended to be utilized as part of a program that manages student grades. Course must be capable of storing the course number and name for a particular class, as well as the number of credit hours associated with the course, and the grade earned. The ADT must implement the following interface:

- constructor: Must accept number, name, credit_hour, and grade parameters (in this order). Must have default values for all parameters and must validate all parameter types and values (hint: utilize isinstance()). If a parameter is of the wrong type or is an inappropriate value (e.g. no negative grades allowed) the constructor must raise a ValueError exception containing a meaningful error message.
- number(): Returns the course number as an integer
- name(): Returns the course name as a string

- credit_hr(): Returns the number of credit hours as a floating-point number
- grade() : Returns the grade as a numeric value in range 4.0 – 0.0
- __str__ (): Magic function. Shall return a string representing the contents of course according to the following format: "cs<course_number> <course_name> Grade: <grade> Credit Hours: <hours>". As an example: "cs1030 Introduction to Computers Grade: 3.2 Credit Hours: 4.0"

Implement this class and validate these methods to your satisfaction before proceeding to the next step.

Make it comparable

Your SList ADT utilizes comparators in order to perform tasks such as maintaining the order of the list and finding elements inside of the list. For this to work, the objects stored in the list must be "comparable". You will need to extend the Course ADT by overriding several magic methods that are automatically called when operators such as <, <=, etc. are invoked. These include AT A MINIMUM:

- __eq__(other) : Returns True if self's value is equal to other's, False otherwise
- __lt__(other) : Returns True if self's value is less than other's, False otherwise;
- __le__(other) : Returns True if self's value is less than or equal to other's, False otherwise;

Note: In all of these cases you must decide what it means for things to be greater, less, equal, etc. For the Course ADT we wish to compare course based on their course numbers (i.e. 1400 would be considered < 1410 etc.)

You may need/want to implement more comparator methods that those listed here. These methods should return True or False based upon a comparison between the object they are called on, and another object or value. As an example, an implementation of __eq__() might look like this:

```
def __eq__(self, other):
    cnumb = other
    if isinstance(other, Course):
      cnumb = other.number()
    return self.number() == cnumb
```

After you have implemented your Course ADT you should validate its behavior using the calculate_gpa() and is_sorted() functions that we have provided in the p3_main_template.py file. These functions will be leveraged when testing your submission.

What to Submit:

- SList.py
- Course.py
- A short 1-minute video showing your code running. Make sure the link is unlisted and not searchable, and paste the link in a comment with your submission.