Assignment 1
IN608: Intermediate Application Development Concepts
Semester Two, 2021

Directions

Below you will find a list of programming problems. You are not expected to do all of them, but instead to choose a subset and complete those. Each problem is worth some number of marks, indicated at the start of the problem. You should probably attempt to complete a set of questions worth about 100 marks. You can't get more than 100 marks on the assignment, but you may want to attempt more than 100 marks worth of problems to maximise your score.

Start your work on this assignment by visiting this GitHub Classroom link: `https://classroom.github.com/a/s0tUsFCT`. After creating your assignment repository, clone it to your workstation. Create a "`working`" branch off the main branch. For each problem, create a subdirectory named for the problem number/name (e.g.: `01-stacks`) and put your code in the appropriate subdirectory. When you are ready to submit your work for marking, issue a pull request, selecting `tclark` as the reviewer. Note that you get one attempt to submit your work. However, if you make a mistake in your submission, like forgetting to include some files, just let the lecturer know.

Your submission can probably be done using only standard library modules, but it may require some additional libraries in order to run. If this is the case, include either a `Pipfile/Pipfile.lock` or a `requirements.txt` file at the top level of your repository.

This assignment is work 30% of your total mark in this paper. Each problem will be marked on

- Correctly solving the problem;
- Coding style and organisation;
- Robust exception handling;
- Adherence to SOLID principles;
- Application of appropriate types, patterns, and other techniques.

Assignment submissions are due on 5:00 PM on 1 October.

Problems

1. Stacks (15 marks): In Python it's pretty natural to implement a stack using a list to store its data. Another common method is to use a *linked list*. A linked list is made up of `Node` objects with two fields. One field holds the node data and the other is a reference to the next `Node` in the list. These `Node` values are chained together to form a list. For the last item on the list, the next node value is `None`.

Write a `Stack` class that uses a linked list for its date storage. Implement `push`, `pop`, `peek`, and `is_empty` methods. Include a `main` section in your file that demonstrates the stack's operations.

2. Postfix calculator (15 marks): Typical calculators use `infix` notation. To enter a calculation, you typically enter the first operand, then the operator, then the second operand. For example, to add 1 and 2 we enter a "1",

then a "+", then a "2". But there was a time when *postfix* calculators were popular. With such a calculator you enter the first operand, then the second operand, and finally the operation. So, $1 + 2$ is entered "1", then "2", then "+". There is an advantage to this when entering a recursive expression like $3 + (16/5)$. The first operand, "3" is easy enough, but the second operand is an expression of its own. With a postfix calculator you would enter "3", then "16", "5", "/", and finally "+".

The most straightforward way to implement a postfix calculator is using a stack. Write a postfix calculator program that takes user input via `stdin` and carries out its operations using a stack. Your calculator should implement addition, subtraction, multiplication, and division. It should be interactive, reading the postfix expression from standard input. The user will enter the complete expression on one line, e.g. "1 1 2 + *" and then "enter". You may assume that the numbers input will only be integers. You may *not* assume that the entered expression is a well-formed postfix expression, so be sure to handle that error.

3. Supermarket queues (20 marks): For this problem you will use software queues to model supermarket queues. First, implement shopper objects that include a `checkout_time` attribute that is randomly assigned a value between 60 and 500 seconds. Generate 1000 shopper objects and insert them into 8 queues that represent supermarket checkouts.

The simulation logic is just a loop. Each pass through the loop represents one second of simulated time. Each pass through the loop:

- Decrement the checkout time for the shopper at the front of each queue. If a shopper's time reaches zero, remove that shopper from their queue.
- Every tenth pass through the loop, generate a new shopper and add them to a queue, until a total of 1000 shoppers is reached.

Keep track of how many seconds (loop iterations) are required to complete the run.

Run four simulations with the following variations:

1. A new shopper selects a queue at random.
2. A new shopper selects the shortest (based on number of shoppers) queue.
3. Like (1), except two of the eight queues are "express queues" that are only available to shoppers having `checkout_time` ≤ 90.
4. Like (2), except two of the eight queues are "express queues" that are only available to shoppers having `checkout_time` ≤ 90.

When your program is run, it should perform each of the simulation types and report the results for comparison.

A tip: It would be easy to write this program with a lot of copy-and-paste monkey code. Use a little design thinking to write good, flexible classes with reusable methods. This problem is a good chance to demonstrate your understanding of SOLID.

4. Decorators (15 marks): Write a function decorator that wraps a function and logs details about the execution to a file. In particular, every time the function is called, write a log message including:

- a timestamp;
- the name of the function called;
- the arguments supplied to the function;
- the value returned;
- any exceptions raised.

Your decorator should pass the return value of the decorated function back to the caller. Any exceptions raised **must** be propagated upstream. Include a `main` section that demonstrates the use of your decorator.

5. Iterators (20 marks): We know that Python lists are *Iterable*, i.e., they produce *Iterators*. In problem (1) we brought up the idea of a linked list. Python lists could be implemented using linked lists, but they're not. For this problem, write a `LinkedList` class that implements a list using a linked list for its data storage. Your class should

1. implement an `append` method,
2. implement a `pop` method,
3. report its length when queried with the `len()` function.
4. implement the Iterator pattern so it can be used in a `for` loop.

Include a `main` section in your file that demonstrates the `LinkedList`'s methods.

Note that there is a decision to be made here about which end of the list is which.

6. Sequences (20 marks): This is an extension to problem 5. You could do this problem without doing problem 5, but it's likely that if you do one, then you'll do both. For this problem you're working with the `LinkedList` class described above. Implement the required methods to make your `LinkedList` a *Mutable Sequence*. For guidance on what methods to implement, see the documentation for `collections.abc.MutableSequence`. It's not necessary that your LinkedList inherit from this abstract base class, but you may choose to do so.

Note that if you do both problems 5 and 6, you may do so by just submitting one file or set of files. That is, you can write one `LinkedList` class that is both an Iterable and a Sequence. Just be sure to include a comment in your code making it clear that you've done this.

7. Observers (20 marks): This problem is inspired by Trademe. Write an auction simulator with two classes `AuctionItem` and `Bidder`. Implement the Observer pattern with the `AuctionItem` as the subject, and the `Bidder`s as the observers. When a bidder bids on an item, that bidder should be automatically added as an observer. Observers are notified whenever a new bid is placed by any bidder.

Each `Bidder` shall have a maximum bid attribute. When the bid price for an `AuctionItem` exceeds a `Bidder`'s maximum bid, that bidder drops out and unregisters from observing the item. When a bidder receives a notification of a new high bid, that bidder should bid 1 to 10 dollars higher than the current bid. (The increase is determined randomly each time.) The bid should never exceed a bidders maximum, however.

Demonstrate your classes' use with a main section that simulates an auction of one item with ten bidders.

8. Factories: (20 Marks): Given a `Book` class like the following:

```
Book:
    title: string
    authors: list of strings
    ISBN: string
    publisher: string
    copyright: date
    categories: list of strings
```

Write a program that lets the user enter information about multiple books. When the user chooses to save the entered information they should be able to choose from JSON, YAML, and XML formats. Write the the entered information to a text file in the chosen format. Sort the books in the output file by title.

Optional (10 marks): Allow the user to chose to sort the output by title, authors, or copyright date.

Resources

For coding style guidelines, refer to PEP 8 (`https://pep8.org/`. You may also find Pylint (`https://pylint.org/`) or flake8 (`https://pypi.org/project/flake8/`) useful.

I *strongly* recommend that you use Pipenv (`https://pipenv.pypa.io/en/latest/`) to manage your project environment, especially if you need to import any extra modules outside of the standard library. Real Python has a good article about Pipenv at `https://realpython.com/pipenv-guide/`.