Project 1 for Info W18 Python Bridge Course

Author: Shawn Kessler

Section 1 - How to use the application:

Command line usage: python3 start_library.py [-h] [--test] [file_name]

-h provides command line help. --test will run unit tests rather than the library system menu. Unit tests are trivial at the moment. file_name is the location of the json file that stores your library; if it is not specified then the default is used, library.json.

For interactive use you should be able to simply run “python3 start_library.py”. This will provide you with a preconfigured library with a small set of rooms, cases, shelves, and books rather than starting with an empty library. If you’d rather start with an empty library, pass in the name of a file that does not exist.

Once you’ve started the interactive system the program will print out a layout of the library (without the books) to give you a feel for what the library looks like before starting in on any tasks. You will be presented with an action menu that has a set of actions you can perform. To perform any action enter the corresponding set of keys and press enter. From there, follow the directions on the screen.

The most complex logic is around adding a book. That is the place most likely to contain errors and if one were interested in trying to break the system I would go there first; you can start this process by using either the “ab” or “abs” action at the main menu. Finding a book, based on any of the available attributes is the second most likely place to discover issues with the program if they exist (these can be invoked with either “fbt”, “fba”, or “fbg”). Beyond that the other features are pretty straightforward. When you are done you can either quit with saving or quit without saving. If you quit with saving the library file is overwritten with the new data you

created while using the system. If you started with an empty library then the new JSON file is created on quit with save. If you experience an error during use of the program your changes will not be saved to the json file.

Files of interest: library.py contains all the class definitions for library type objects. start_library.py has all classes and function (mostly functions) used in the menu system implementation. Initially they were all in one file but since I'd never made my own module and included it in another Python file I thought I'd break them into two logical pieces and see what that process looked like.

Section 2 - Project Development Narrative

The first thing I wanted to try--because Java can't do it--was create classes with multiple parent classes. The initial construction of the class hierarchy wasn't hard, but it took me a while to make Containable.print_human_readable_full_location work like I wanted without making Library a subclass of Containable, which didn't make logical sense to me. Namily I wanted the Library printed as part of the description of the hierarchy and for Library to only extend Container.

Another difficulty with multiple inheritance, for me, was figuring out when/where to invoke super(). This was only needed in the __init__ methods for my use. With single inheritance I only needed to call super().__init__ in my subclasses. But when a class, like Room extended both Container and Containable, I eventually realized I needed to call super().__init__ in each super class's __init__ methods as well to make sure both of them get invoked when a new Room object is instantiated. I had first assumed that I'd need to somehow invoke both of the super classes __init__ methods directly from the

subclass--perhaps with some arguments for each call to super that told it which class I wanted, but that seemed less maintainable since it would mean I'd have to change those calls if the parent class names ever changed--but it appears that if you have Room(Container, Containable) and call super() in every one of those classes' __init__ methods, then the super from Room will invoke Container's __init__ and the super from Container will invoke Containable's __init__ method. This wasn't exactly intuitive because Containable isn't a super class of Container, but it seems to work. I'm still not certain if this is the correct way to chain the super calls but it achieved my goal.

My second self-induced challenge was using JSON as my file store format rather than pickle. I wanted JSON so that I could read the output files, but pickle otherwise would have been a more natural choice since it handles custom object types by default while the JSON dump and load utilities do not. This ended up being the major place in the project where I couldn't make enough sense of the Python documentation to implement what I wanted. I eventually ended up elsewhere on the Internet for more in-depth examples (http://www.diveintopython3.net/serializing.html). The decode and encode classes I built feel heavy handed but as best I can tell the conversion back and forth to JSON and Objects has to be pretty explicit.

While building the menu functionality I pondered making Menu a class, but since there would only ever be a single instance of that class it felt unnecessary. Plus I wasn't sure how/if you could pass methods as arguments--the same way you can a function--without instantiating an instance of the object type, which was a feature I wanted for my main menu configuration. I think I could have made the action methods all static methods in a Menu class in order to achieve the same effect, but then having a class full of static methods seemed unuseful, so I stuck with functions.

As mentioned in the “how to use” section above, logic-wise the hardest stuff to code was inserting books and handling the resulting shift of other books to different shelves .