TPK4186 - 2023 - Assignment 1 - Group 47

Introduction of the program

My program is divided into a number of modules or files, each of which houses a collection of related classes or functions. The modules are organized hierarchically, with some modules reliant upon others. A single script file named “main.py” serves as the program's main entry point and imports and uses the other modules as necessary.
The application uses a modular approach to programming to enhance code reuse and maintainability. Each module or file has a distinct and clear purpose, and the methods or classes are titled in a way that makes it clear what functionality is intended. The program makes use of object-oriented programming ideas like encapsulation to streamline the organization of the code and cut down on duplication.

Proposed solution

Every class has a main function that checks the functionality for the given class. I have also tried to make a visual representation of both a section, stack, and the ship using print functions.
However, in the Main.py file, I am chronologically solving each of the tasks, as far as that is possible. To see how the program works, uncomment each task in the “Main.py” file. Run the tasks individually. If you run all the tasks at once, the generate_random_code function does not reset, and the container codes will not be from 0000 and upwards.
Further is how each task is solved:

2.1 Containers

Task 1) See def task1() in “Main.py”. Makes use of Container class
Task 2) See def task2() in “Main.py”. Makes use of the Container and ContainerSet class.

Task 3) See def task3() in “Main.py”. Makes use of both Container class and ContainerSet class
Task 4) See def task4() in “Main.py”. This logic is implemented in ContainerSetManager.py

2.2 Ships

Assumptions for the ships:
The length of the ship in the program is 24 cells long, so we can split the ship into six sections of equal lengths. The sections with id 0, 2, 4 represent the starboard of the ship, and section ID 1, 3, 5 represents the portside of the ship. Section IDs 0 and 1 represent the front of the ship, section ID’s 2 and 3 represent the middle of the ship, and sections 4 and 5 represent the back of the ship.
Assumptions for the sections:
A ships’ section is made up of different ContainerStack objects that represent a pillar of containers for each coordinate in the section. We assume that a 40-foot container must be stacked immediately on top of another 40-foot container or directly on top of two consecutive 20-foot containers. This would take up one cell slot in the ship, and therefore the length of each section is half the length-size than it would be without this assumption. Since 20-foot containers are delivered in pairs, if we try to load a 20-foot container into a section, we put this container in a “holding-spot” in the given section represented as a list in the ContainerSection class. When the holding-list reaches a length of 2 (that is the length of these two containers has a length of 40), the pair of containers are sort of “glued” together, and placed in the section. When so, they are unable to be separated and are sorted by weight as one 40-foot container, if stack-sorting is needed later on. It still occupies 2 adjacent grid spaces and requires 2 operations each pop/push. This assumption vastly reduces the amount of operations needed to load a ship in descending weight order, and has little impact on the ships balance after all.
Task 5) Adding a container takes use of the hierarchical structure in ContainerShip, ShipSection and ContainerStack classes. (Placed in the lightest stack in the lightest section)

def task5(): First load a set of containers into a ship, then load a new set of containers into the same ship. Print out every available stack, and then print out the ship. Then look for a ship, remove it and look for the same container again.
Task 6) When loading a ship from a file, we do not read from the same file as we saved it to. Instead we load the ship with the same set that was loaded into the originally saved ship, to ensure that we retrieve an identical saved_ship file when saving this “new” ship again. This is due to the fact that we cannot directly insert a container into a stack in a given section. If we also were to iterate through each line in the saved_ship file and then load them chronologically into a “new” ship, it would not recreate the same ship, as the ship randomly chooses between sections if multiple sections have identical weight. If the ship does not exist, the user is notified and a new ship with the input amount of containers loaded to it. All of this logic is found in the ContainerShipManager.py file.
Task 7) These function are in the ContainerShip class, named load_ship(container_set), and unload_all_containers()
Task 8) This is implemented in the ContainerStack class under the add_container_to_stack function. The functionality of this logic is also confirmed in the ship.are_containers_placed_in_descending_order() function.
Task 9) When loading a random container onto the ship, we are placing it in the lightest stack in the lightest section. This will result in the ship (almost) always being balanced. However since the ship has no weight attribute, it would be unbalanced until we have loaded about 100 containers. Since a container ship most certainly weighs a lot regarding if it has containers on it or not in the real world, we do not check for the ship being balanced for each container we are loading into the ship, but only after the ship is finished loading containers. The stability functions are in the ContainerShip class.
Task 10) My assumption that the ship is always balanced when containers are placed in the lightest spot, removes the need to implement the stability constraints in the loading function.

2.3 Docks

Task 11)
We are keeping track of an operation counter in the ship, and we are adding 1 to it every time we do a push or pop operation on a container. To get the time, we multiply

the operation counter by 4 minutes (240 seconds). The crane functions are all in the ContainerShip class.
The results we get is that a single crane uses about 54000 operations to load a full ship with random containers which equals to around 150 days to load up the ship. To unload a full ship, the amount of used operations is the number of containers in the ship, meaning it takes around 18 days to unload a full ship.

Task 12)

Due to our implementation of a 3x2 grid, we only have **3 cranes** instead of 4. Nevertheless, as 4 is not divisible by 3, this would make it more difficult to compute the front, middle, and end sections.
A solution to this might be to divide the ship into a grid of 12x2, where the first 4x2 portions represent the front, etc. The front, middle, and back sections would be simpler to compute as a result, but it would be more time consuming to find the lightest section, as it would result in more iterations. This would also make finding the lightest stack more time-consuming and result in a longer runtime.
We assume that a crane is assigned to work in 2 adjacent sections of the ship.
The results for loading a full ship are as follows: we find the max amount of operation a crane uses in a section. We then use this operation counter to calculate the time it takes.
It takes around 18000 operations and 50 days to load up a full ship.
It takes around 2100 operations and 6 days to unload the ship.

Other assumptions and discoveries

1) If the ship has to load a set of 4000 containers and the ships capacity is 6500 containers, some containers may still be in the holding-spot after finishing loading this set.This would in practice mean that the ship did not load the entire set of containers, even though the ship has remaining capacity. Because of the assumption that 20-foot containers must be placed in pairs, we assume that the holding containers are placed back into the docks, waiting to be loaded in another ship.
2) For every container that is either pushed or popped from a container stack in a section, we increment the number of operations used and store this value in the stack, and in the section. This makes it easier when calculating the time it

takes to load or unload the ship, as we assign cranes to certain sections. For example, crane 1 would work in section 0 and 1, so calculating the time it takes to load/unload this section, we just multiply the counts of operations in these sections by four minutes.

Experiments

After each task-function in the “main.py” file. I have included some edge-cases, as well as some experiments. Feel free to check them out!