EIMETALS project

The program expression

In this project, I completed the programming task for `Primetal Technologies`. I employed an object-oriented programming (OOP) approach as I needed to update values in a list using various methods.

Given that certain modifiers, such as `@dryrun` and `@sorted`, apply the input method on a copy of the current list, I placed the `param_parser` function as a `decorator` at the beginning of the Inventory class. Then, prior to each method, I applied this decorator.

```
def param_parser(func):
    def wrapper(self, *args, **kwargs):
        self.current_working_num = copy(self.nums)
        modifiers = kwargs.get("modifiers", [])
        repeat = 1

        if any(item in modifiers for item in ['@once' , '@twice' , '@often']):
          repeat = 0

        commit = True
        sorted = False
        dryrun_flag = False

        for mod in modifiers:

            if mod == "@once":
                repeat += 1

            elif mod == "@twice":
                repeat += 2

            elif mod == "@often":
                repeat += 10

            elif mod == "@dryrun":
                commit = False
                dryrun_flag = True

            elif mod == "@sorted":
                sorted = True
                commit = False
                self.current_working_num.sort()

        for _ in range(repeat):
```

```
            result = func(self, *args, **kwargs)

        if commit:
            self.nums = copy(self.current_working_num)
            self.current_working_num = []
        elif dryrun_flag:
            print(self.current_working_num)

    return wrapper
```

Since it has been mentioned in the EIMETAL_en.pdf file that there can be more than one modifier, which means we could potentially have `@once @twice` in the modifier part of the input, we should include the number of executions that each modifier determines. In this case, we should execute the operation 3 times (one time because of @once plus 2 times because of @twice).

Furthermore, since three types of modifiers affect the number of executions(`@once`, `@twice` and `@often`), I only wrote the if conditions for these cases. For the other two modifiers, I simply set flags.

for each method I have used `*args` and `**kwargs`. the `*args` is for those method that we could have several inputs (such as insert); due to symmetry observance I used *args in all functions, and I used `**kwargs` to read modifiers.

As the input can contain special characters, I used the `Regex` library to extract the operation, arguments, and modifiers.

The executer function utilizes the above parameters to detect and apply the method to the inputs, taking modifiers into account.

Here is a sample of the output:

```
> show
[]
> insert 1 29 1 23 9 13 14 16 7
> pop 2 @dryrun
[1, 29, 1, 23, 9, 13, 14]
> show
[1, 29, 1, 23, 9, 13, 14, 16, 7]
> index 29
1
> index 29 @sorted
8
> show
[1, 29, 1, 23, 9, 13, 14, 16, 7]
> esi
Operation unknown
> remove 0 @once @twice
> show
```

```
[23, 9, 13, 14, 16, 7]
> push 2 @often @dryrun @sorted
[7, 9, 13, 14, 16, 23, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
> show
[23, 9, 13, 14, 16, 7]
> isSorted
false
> sort
> isSorted
true
> show
[7, 9, 13, 14, 16, 23]
> exi
Operation unknown
> exit
```

Run time estimation:

in the following table, I have brought the run time estimation of each method (without modifiers):

Method	Worst-Case Time Complexity	Reason
insert	O(n)	Since we have to go though all inputs one by one and append them, the big-O notation if O(n)
pop	O(n)	in the worst case we have to pop all of the elements of the list, which means the Time Complexity is O(n)
show	O(n)	Since we should go through all of the list the time complexity if O(n)
exit	O(1)	in this case we just set a flag so this is O(1)
index	O(n)	we should find index of an element so we should go through all elements of the list
get	O(1)	we should check only one element of the list

Method	Worst-Case Time Complexity	Reason
remove	O(n)	in the worst case we should remove all elements of the list.
insertFront	O(n)	Inserting elements at the beginning of a list requires shifting existing elements, so it has a time complexity of O(n), where n is the number of elements to be inserted.
popFront	O(n)	Similar to insertFront, it has a time complexity of O(n) because it may involve shifting elements
sort	O(n*log(n))	The Python list sort() has been using the Timsort algorithm since version 2.3. This algorithm has a runtime complexity of O(n.logn).
isSorted	O(n)	This method should also go though all of the elements of the list
push	O(n)	similar to `insert`
EOF	O(1)	similar to `exit`