EEE111 SP1 Documentation

Nile Jocson <novoseiversia@gmail.com>

October 11, 2024

Contents

1 Introdution

This is the documentation for my EEE111 SP1 submission. Included here is documentation for each user-defined function, rationales for how the logic is written, how user commands are executed, etc. A reference for each user-defined function and class is also included.

2 Documentation

2.1 Overview of the program

The program is a CLI supply and inventory monitoring program, created for a hospital, based on the EEE111 SP1 specifications provided. It accepts user input, and allows the following commands (in docopt form):

```
<file_name:str> needed_now
<file_name:str> needed_in <X:int>
<file_name:str> runs_out
<file_name:str> <N:int> run_outs
help
exit
```

- `<file_name:str> needed_now` prints out the amount needed to fulfill the item shortage for the current day.
- `<file_name:str needed_in <X:int>` prints out the amount needed to fulfill the item shortage for the following `X` days.
- `<file_name:str> runs_out` prints out the first item to run out, and in how many days it will[1].
- `<file_name:str> run_outs` prints out the virst `N` items to run out, and in how many days they will[1].
- `help` prints out the help text.
- `exit` exits the program.

2.2 Input file

The commmands `needed_now`, `needed_in`, `runs_outs`, and `run_outs` takes in a `<file_name:str>` parameter, which is a string corresponding to the filename of the CSV file containing the hospital's supply database. This file can have a variable amount of rows, but the amount of columns is fixed. These columns are as follows:

1. Item name: str
2. Item current amount: int
3. Item daily usage: int

[1]These commands will sort in this order in the case of conflicts: by amount of days to run out (ascending), by amount of deficit (descending), and by lexical order of their names (ascending).

2.3 Parser logic

The program relies on a set of enums and functions in order to make it easier to parse user input into commands and their arguments, not relying on tedious chains of if-elif-else statements. These comprise the parser logic of the program. The core of this logic is the function

```
parse_rules(
 input_rules: InputRules,
 output_rules: OutputRules,
 args: list[str]
) -> list[Any] | None
```

which parses the given string arguments and outputs a validated, converted and reordered arguments, based on the given input and output rules. This makes it extremely easy to validate user input, and parse them into well-defined lists of commands and their arguments. The logic of this function can be broken down into two steps:

2.3.1 Application of input rules

An input rule can be either a type or a string. If it is a type, then that type must be constructable from the argument under that rule. If it is a string, then that string must be the same as the argument under that rule. Otherwise, the argument is invalid. For example:

- With the input rule `int`, the argument `"Hello"` is invalid, since that cannot be converted into an integer. However, `"500"` is valid.
- With the input rule `"exit"`, the argument `"exita"` is invalid, since the two strings are not equal. However, `"exit"` is valid.

For a list of input rules `InputRules = list[type | str]`, the list of arguments must be the same length, and each argument must be valid with its corresponding input rule, for the list of arguments to be valid. For example:

- With the input rules `["exit"]`, the arguments `["exit", "1"]` are automatically invalidated since the amount of arguments is not equal to the amount of input rules. However, `["exit"]` is valid.
- With the input rules `[int, str]`, the arguments `["Hello", "World"]` are invalid, since `"Hello"` is not an integer, thus invalidating the entire list. However, `["500", "World"]` is valid, since each argument follows its corresponding input rule.
- With the input rules `["abs", int]`, the arguments `["absv", "-5"]` are invalid, since `"absv"` is not the same as `"abs"`. However, `["abs", "-5"]` is valid.

If all string arguments are valid, then a list is created of the converted arguments based on the input rules. For example:

- With the input rules `["add", int, int]`, the arguments `["add", "5", "8"]` are converted into `["add", 5, 8]`.
- With the input rules `["floor", float]`, the arguments `["floor", "4.8"]` are converted into `["floor", 4.8]`.