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 first `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>` argument, 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:

- For the input rule `int`, the argument `"Hello"` is invalid, since that cannot be converted into an integer. However, `"500"` is valid.
- For 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:

- For the input rules `["exit"]`, the arguments `["exit", "1"]` are invalid, since the amount of arguments is not equal to the amount of input rules. However, `["exit"]` is valid.
- For the input rules `[int, str]`, the arguments `["Hello", "World"]` are invalid, since `"Hello"` is not an integer. However, `["500", "World"]` is valid.
- For 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]`.

These converted arguments then move on to the next step:

2.3.2 Application of output rules

An output rule can either be an integer or a tuple of an integer and a type, i.e. `OutputRules = list[int | tuple[int, type]]`. The integer corresponds to an index in the argument list, and if a type is specified, then the element in that index is converted into that type. The converted element is then appended into the list of fully processed arguments which will be returned from the function. For example:

- The output rules `[2, 1, 0]` will rearrange `["A", "B", "C"]` into `["C", "B", "A"]`.
- The output rules `[2, [1, str], 0]` will rearrange and convert `[5, 8, 13]` into `[13, "8", 5]`.

This rearranged list is then returned from the function.

2.3.3 Rulesets

Generally, and especially in this program, user input is not validated only using one set of input and output rules. Instead, since this program needs to handle 6 different commands, it needs to check against 6 different sets of input and output rules, or rulesets, i.e. `RuleSet = tuple[InputRules, OutputRules]`. In order to make this a trivial task, another parse function has been defined

```
parse_rulesets(
 rulesets: list[RuleSet],
 args: list[str],
 default: list[Any]
) -> list[Any]
```

which takes in a list of arguments and tries to parse them against the given rulesets. The first valid parse is returned, or if all parses were invalid, then the value of the `default` parameter is returned instead.

2.4 Database logic

2.4.1 Deserializing the database

Some classes, aliases and functions have also been defined in order to make it easier to handle the information inside the supply database. The function

```
parse_database(filename: str) -> SupplyDatabase
```

Opens and reads the CSV file denoted by the given filenme. It then parses the file line by line using the docopt string

```
<item_name:int>
<item_current_amount:int>
<item_daily_usage:int>
```

reusing the parser logic in 2.3 and populates a `SupplyDatabase` instance, and returns it. A `SupplyDatabase` is a `dict[str, StockInfo]`, which contains stock information for each item. A `StockInfo` is then defined as follows:

```
@dataclass
class StockInfo:
 quantity      : int
 daily_usage   : int
 remaining_days: int
 deficit       : int
```

A tuple was not used for the purpose of storing stock information since there is no way to distinguish between the different values other than using their indices. By using dataclasses, names can be given to the elements, making the source code more readable.

2.4.2 Sorting the database

In order to meet the requirements of `runs_out` and `run_outs`, there needs to be a way to sort the `SupplyDatabase` based on three factors. The function

```
get_sorted_supply_database(
 database: SupplyDatabase
) -> SortedSupplyDatabase
```

was made for exactly this purpose.