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:

- Item name: `str`
- Item current amount: `int`
- 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

In order to make code more readable when dealing with multiple items in a database, a `dataclass` (C `struct` equivalent) `StockInfo` was created. This allows us to refer to each element, i.e. item name, item quantity, item usage, etc. by variable names, instead of indices if tuples were used instead. The definition of StockInfo is as follows:

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

An alias `SupplyDatabase = dict[str, StockInfo]` is also defined. These are used in the function

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

which opens a CSV file, parses it with the parser logic in 2.3, calculates the remaining days and deficits for each item, and returns a `SupplyDatabase` instance with all the gathered stock info.

2.4.2 Sorting the database

Sorting the database is necessary for the correct output for the commands `runs_out` and `run_outs`. The alias and function

```
SortedSupplyDatabase = list[tuple[str, StockInfo]]

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

were defined for this purpose. A new alias is necessary since a Python `dict` is not sorted by default, and a separate function is necessary since Python's `sort` and `sorted` functions cannot take in a custom comparator (unlike C++'s `std::map<T>` and `std::sort()`)

In order to properly sort the database, the function takes advantage of the fact that Python's default sorting algorithm is a stable sort. This means that it will not touch the order of equal-keyed elements. With this in mind, sorting with multiple criteria can be done by sorting multiple times, in reverse order of the criteria.

For example, in order to sort the list

```
[(1, 5), (1, 2), (1, 9), (2, 5), (3, 6)]
```

by the first element first, then the second element, it is necessary to sort by the second element first:

```
[(1, 2), (1, 5), (2, 5), (3, 6), (1, 9)]
```

and then to sort by the first element second:

```
[(1, 2), (1, 5), (1, 9), (2, 5), (3, 6)]
```

This fact extends to sorting with any amount of criteria, including the sorting criteria defined for `runs_out` and `run_outs`.

2.5 Parsing user input

2.5.1 Commands

Recall that multiple different commands with different parameters were defined in 2.1. While the different command names will be present in valid user input, they are present simply as strings, and not explicitly as command names. In order to explicitly define each command, an enumeration was defined:

```
class CommandType(Enum):
 NEEDED_NOW =  0,
 NEEDED_IN  =  1,
 RUNS_OUT   =  2,
 RUN_OUTS   =  3,
 HELP       =  4,
 EXIT       =  5,
 INVALID    = -1,
```

This makes invalid command names easily handleable, eliminating the need to constantly check strings. A convenience function was still defined in order to easily convert between strings of command names in any case and the well-defined command types, returning `CommandType.INVALID` for invalid command names:

```
_missing_(cls, value: object) -> Any
```

This function also allows the conversion between strings and enums in the parser functions in 2.3 as a typed output rule. For example, with the output rules `[(1, CommandType), 0]`, the arguments `["Hospital1.csv", "needed_now"]` will be converted into `[CommandType.NEEDED_NOW, "Hospital1.csv"]`.