EEE111 SP1 Documentation

Nile Jocson <novoseiversia@gmail.com>

October 17, 2024

Contents

1 Overview

The program is a CLI supply and inventory monitoring program based on the EEE111 SP1 specifications provided. It accepts the following commands:

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

The program will accept commands indefinitely, exiting only when the `exit` command is entered. If an invalid command is entered, the program prints out the help text.

2 Parsing

Parsing utilities were created in order to simplify parsing inputs. This eliminates the need for long if-elif-else chains and complicated string comparisons, and instead abstracts them into an easy-to-use API. This API consists of two dataclasses and two functions:

```
@dataclass
class TransformInfo:
 convert : type | Callable
 position: int

@dataclass
class ParseRule:
 transforms : list[TransformInfo]
 find_string: str | None = None

def parse_rules(
 rules: list[ParseRule],
 args : list[str]
) -> list[Any] | None

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

The inner workings of this API can be broken down into three processes.

2.1 Input

The parsing API uses input in the form of `list[str]`. It does not handle actual program IO, nor does it handle the splitting of the input lines. This input type is derived from the return type of `str.split()`, which is used to split user input by a specified delimiter. From now on, the term "arguments" will be used to refer to user input with type `list[str]` given to the parsing API.

A rule determines if an argument is valid. It specifies a list of transforms which contains either types that the argument must be convertible to, or functions of which the argument must be a valid parameters of. In addition, it may also specify a string which the argument must be equal to; this is used to disambiguate between command names. If any one of these were not satisfied, the argument is invalid. A rule is defined with the `ParseRule` dataclass.

Most programs however, take in multiple arguments, thus requiring multiple rules. Rules are simply a `list[ParseRule]`, where each rule has a corresponding argument with the same index. Each element is validated against its

corresponding rule. If any one rule is not satisfied, the whole argument list is invalid.

Here are some examples of rules, and their valid and invalid arguments:

Rules	Valid	Invalid
`int`	`"5"`	`"a"`
`int, int`	`"5", "10"`	`"5", "a"`
`"add", int, int`	`"add", "5", "10"`	`"sub", "5", "10"`
`"abs", int`	`"abs", "-5.4"`	`"absv", "-5.4"`

If the arguments given to `parse_rules` are invalid, it returns `None`.

2.2 Output

As said before, rules specify a list of transforms. These transforms are defined with the `TransformInfo` dataclass. A transform consists of either a type or a function, and an integer. The argument corresponding to the rule containing the transforms is either converted to the given type or fed into the given function, the results of which are inserted into a list at the index indicated by the given integer.

Here are some examples of rules with transforms, and their input and outputs:

Rules	Input	Output
`{[{int, 1}]}, {[{float, 0}]}, {[{float, 2}]}`	`"10.0", "5", "6"`	`5.0, 10, 6.0`
`{[{int, 1}], [{float, 0}], [{float, 2}]}`	`"94"`	`94, 94.0, "94"`
`{[{str, 0}], "abs"}, {[{float, 1}]}`	`"abs", "94"`	`"abs", 94.0`

If the arguments given to `parse_rules` are valid, it returns a `list[Any]` of the transformed arguments.

2.3 Rulesets

While a rule is for a single argument, and rules are for multiple arguments, rulesets are for multiple commands. A ruleset is simply a `list[list[ParseRule]]`, or a list of lists of rules. Rulesets allow the given arguments to be parsed against multiple different rules, and transforming them against the first rules that successfully validate them. This process is governed by the `parse_rulesets` function.

For example, given the ruleset:

```
"add", int, int
"sub", int, int
 "abs", float
```

the following arguments will be validated by the selected rule, returning the outputs:

Arguments	Selected	Output
`"add", "5", "8"`	`"add", int, int`	`"add", 5, 8`
`"sub", "90.4", "3.13"`	`"sub", int, int`	`"sub", 90, 3`
`"abs", "-144"`	`"abs", float`	`"abs", -144.0`
`"sin", "90"`	`None`	`default`

Note that if no rules successfully validate the arguments, the `default` argument of `parse_rules` is returned.

3 Database

The database used by this program consists of items, one of which then consists of a name, a quantity, and a daily usage. In order to make database handling simple, an API of one dataclass, two aliases, and three functions were created.

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

type SupplyDatabase = dict[str, StockInfo]
type SortedSupplyDatabase = list[tuple[str, StockInfo]]

def parse_database(
 filename: str
) -> SupplyDatabase

def supply_database_compare(
 left: tuple[str, StockInfo],
 right: tuple[str, StockInfo]
) -> int

def sort_supply_database(
 database: SupplyDatabase
) -> SortedSupplyDatabase
```

This API can be broken down into two parts.

3.1 Formats

The parsed format of the database is defined by the `SupplyDatabase` alias. This alias is dictinary consisting of an `str` key corresponding to the item name and a `StockInfo` value which includes the remaining two item values, as well as another two computed values:

- `remaining_days` is how many days it will take for the stock of the item to run out, given the daily usage.
- `deficit` is how much of the item should be acquired in order to satisfy the shortage on the day it runs out.

The input format of the database is a comma-separated values (CSV) file with a variable amount of rows corresponding to each item, and three columns corresponding to each item value.

The conversion from the input format into the parsed format of the database is handled by `parse_database`. This function takes in the filename of the input file, parses it, and returns a populated `SupplyDatabase` instance, including the two calculated values. The input file can be parsed by splitting each line using commas as the delimiter, then using `parse_args` with the rules

```
{[{str, 0}]}, {[{int, 1}]}, {[{int, 2}]}
```

If the input file format is invalid, `parse_database` raises a `RuntimeError`.

3.2 Sorting

Since Python does not have a built-in sorted dictionary (unlike C++, with its `std::map`), the `SupplyDatabase` must be manually sorted in order to support some of the commands specified in 1. A custom sorting order is required by the specification:

- `remaining_days`, in ascending order
- `deficit`, in descending order
- `name`, in ascending lexicographic order

In order to sort using this order, a custom comparator is needed. This custom comparator is the `supply_database_compare` function. This comparator function follows the specification or `functools.cmp_to_key`; if the item on the left is less-than, equal, or greater-than the item on the right, the function should return a negative value, zero, or a positive value, respectively.

The convenience function `sort_supply_database` simply calls `sorted` on the `SupplyDatabase` with `supply_database_compare` as the comparator. The result of this sort is defined by the `SortedSupplyDatabase` alias, which is a list of tuples of the item values. This change in container type is required since the `SupplyDatabase`, which is simply a `dict`, cannot preserve the order of its elements, and as such cannot be sorted on its own.