EEE111 SP1 Documentation

Nile Jocson <novoseiversia@gmail.com>

October 11, 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 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 complicatede 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 logic of this API can be broken down into three processes.

2.1 Input

2.1.1 Arguments

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 was 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.

2.1.2 Rules

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` class.

Most programs however, take in multiple arguments, thus requiring multiple rules. Rules are simply a `list[ParseRule]`, where each element has a corresponding argument (corresponding rules and elements have 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`. If not, it moves on to the next process.

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, 2.0
{[{int, 0}, {float, 1}, {str, 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.