Assignment 2 Report

LeTian Wang 301421744 CMPT 473 FALL 2024

Prerequisites

To successfully run this program, ensure that the following tools are installed and properly configured on your system:

1. Python: The `csv2json.py` script is written in Python, so you'll need Python installed on your machine. Version 3.6 or higher is recommended.
2. Pandas: The script relies on the **pandas** library for data manipulation. You can install Pandas using the following command:

```
pip install pandas
```

4. Shell: The test harness (`run_all_tests.sh`) is a shell script, and a compatible shell environment is required to execute it. On Windows, it is recommended to use **Git Bash** as the terminal to provide a Unix-like environment.

Setup Instructions

To run the test harness and the `csv2json` program, you need to clone my repository. The repository contains the `csv2json.py` Python script as well as the test harness and data files necessary to perform the testing.

Clone the repository using the following command:

```
git clone git@github.com:130-jwang3/CMPT_473_e2.git
```

Specification of Program Under Test (PUT)

Program Under Test: `csv2json.py`

Description

`csv2json.py` is a Python script that converts a CSV file into JSON format. The script provides various options for customizing the conversion, such as selecting specific columns or rows, defining custom separators, and more.

Inputs

Positional Arguments

- `infile` *(string)*:
  The input CSV file. Default is `data.csv` .
- `outfile` *(string)*:
  The output JSON file. If not provided, the default output file is `[infile_basename].json` .

Optional Arguments

- `-S, --separator SEPARATOR` :
  Specifies the separator used in the CSV file. Default is a comma `,` .
- `-H, --headerline HEADERLINE` :
  The line number to use as column names. Default is the first row (`1`).
- `-c, --columns COLUMNS` :
  The number of columns to crop to.
- `-u, --usecols USECOLS [USECOLS ...]` :
  A list of column names to use. This is useful for selective conversion of specific columns.
- `-n, --names NAMES [NAMES ...]` :
  Custom column names to use instead of those in the header.
- `-N, --nrows NROWS` :
  The number of rows to read from the CSV file.
- `-s, --skiprows SKIPROWS` :
  The number of rows to skip at the beginning before reading.
- `-r, --userows USEROWS [USEROWS ...]` :
  List of specific rows to use.
- `-a, --append APPEND [APPEND ...]` :
  Additional columns to append to the output.
- `-p, --printdata` :
  Prints the formatted JSON data to the console after conversion.

Outputs

JSON File

The output is a JSON file containing the data from the CSV file, formatted according to the specified parameters. By default, the output file is `[infile_basename].json` . The JSON output will use the following structures based on the input arguments:

- **Array of Records**: The JSON output will be an array of records (objects), where each record corresponds to a row in the CSV. Column names are used as keys, and cell values are used as values.

Error Messages

Standard error messages will be displayed for issues such as:

- **Missing Input File**: If the specified input file does not exist.
- **Improper Formatting**: If the CSV file cannot be properly parsed (e.g., due to incorrect delimiter or other issues).
- **Invalid Arguments**: If any of the command-line arguments are improperly provided.

CSV Specification (Input Format)

The input CSV file should follow RFC 4180 specifications:

- **Field Separator**:
  The default field separator is a comma `,`, but it can be customized using the `-S` flag.
- **Headers**: (did not satisfy) By default, no headers are assumed (`-H 0`). The user can specify which line should be treated as the header using the `-H` flag. If no headers are provided, default names will be used for columns.
- **Fields**:
  The CSV fields may include special characters, and proper handling of double quotes is expected for fields containing commas or line breaks.

JSON Specification (Output Format)

The output JSON file follows RFC 8259:

- **Array of Objects (Records)**:
  Each CSV row is converted into an object in JSON. If a header row is specified, the keys for each object are derived from the header values. If no header is provided, generic keys such as `"col_1"`, `"col_2"`, etc., will be used.

Example Input-Output Mapping

1. **Input CSV File** (`data.csv`):

```
name,age,city
Alice,30,New York
Bob,25,Los Angeles
```

2. **Command**:

```
python bin/csv2json.py simpleTestRun/data.csv simpleTestRun/data.json
```

3. **Output JSON File** (`data.json`):

```
[
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "Los Angeles"}
]
```

Input space partitioning

```
[System]
-- specify system name
Name: csv2json pairwise test model
```

```
[Parameter]
-- general syntax is parameter_name : value1, value2, ...
Input_File_Exists (boolean) : true, false
Header_Line (enum) : NO_HEADER, FIRST_LINE, SPECIFIC_LINE(>=2)
Input_Source (enum) : STDIN, DISKFILE
Output_Destination (enum) : STDOUT, DISKFILE
Limit_Rows (boolean) : true, false
Skip_Rows (boolean) : true, false
Custom_Column_Names (boolean) : true, false
Column_Selection (enum) : ALL_COLUMNS, LIMIT_COLUMNS, USE_SPECIFIC_COLUMNS
Explicit_Row_Selection (boolean) : true, false
Append_Columns (boolean) : true, false
Print_Data (boolean) : true, false
Separator_Type (enum) : COMMA, SEMICOLON, TAB, CUSTOM
Number_Of_Records (enum) : ZERO, GTZERO
Consistent_Field_Count (boolean) : true, false
Field_Type_In_Record (enum) : ESCAPED, NONESCAPED, MIXED

[Constraint]
-- this section is also optional
Input_Source="DISKFILE"=>Input_File_Exists=TRUE
Output_Destination="DISKFILE"
(Field_Type_In_Record == "ESCAPED" || Field_Type_In_Record == "NONESCAPED") => Number_Of_Records == "GTZERO"

Input_File_Exists==TRUE
Input_Source=="DISKFILE"
```

Constraints explanation

The following constraints ensure meaningful and realistic test cases:

1. Input Source and File Existence:
    - If Input_Source is "DISKFILE", then Input_File_Exists must be TRUE.
    - All tests assume the input file exists (Input_File_Exists == TRUE) and use a disk file (Input_Source == "DISKFILE").
2. Output Destination:
    - The output must be a disk file (Output_Destination == "DISKFILE") to enable file-based verification.
3. Field Type and Records:
    - If Field_Type_In_Record is "ESCAPED" or "NONESCAPED", there must be records available (Number_Of_Records == "GTZERO").

These constraints simplify the tests by avoiding **unrealistic** or **error-specific** scenarios, focusing instead on validating the conversion functionality.

Running all ACT generated tests with Shell

```
sh ./run_all_tests.sh
```

Final Report

How many tests did ACT generate?

13

How many of these tests were successful/passing?

4

How many tests would have been generated if I didn't use pairwise testing?

221,184 (by combining all the possible input parameter partition)

Tradeoffs of Pairwise Testing

While pairwise testing is efficient and allows for a broad coverage of parameter interactions, there are limitations and tradeoffs:

- **Limited Coverage of Higher-Order Interactions:**
    - Pairwise testing focuses on covering every pair of parameter values at least once, which means higher-order interactions involving three or more parameters may not be fully tested. For instance, specific combinations involving more than two interacting variables might be missed if those combinations are particularly problematic.

This implies that bugs that might be caused by a combination of 3 or more factors go untested.

Errors Discovered During Testing

The following errors and limitations were discovered in the `csv2json.py` program during the evaluation:

1. No Option to Set Header Line to None:
    - The program did not have an option to specify `None` as the value for the `--headerline` parameter. This led to issues when attempting to handle CSV files that did not contain any header rows. For example, attempting to specify a header line as "None" resulted in argument parsing errors. Such limitation restricts the flexibility of the tool and violates the RFC 4180 specifications.
2. Inconsistent Number of Fields Per Row:
    - The program did not properly handle cases where rows in the CSV file had different numbers of fields. If the number of fields per row varied, it would often lead to errors or incorrect parsing behavior, preventing successful conversion to JSON. This inconsistency is common in messy or real-world data, highlighting a robustness issue in the implementation.
3. Tab Separator Not Accepted:
    - The program could not handle `\t` (tab) as a separator correctly. Although the shell script attempted to pass the tab character using `-S $'\t'`, the program would fail to parse the separator argument properly. This suggests an issue either with how arguments are parsed or with the handling of escape sequences, which significantly limits the ability to work with tab-delimited files.
4. Order of Operations Affecting Limit Columns:

   - The `--columns` parameter did not work as intended due to the order in which operations were applied based on the provided flags. Specifically, applying row selection, skipping rows, or appending columns before limiting columns could cause unexpected results or prevent columns from being properly limited. This indicates a design flaw in the sequence of execution of the program.

Reflection on experience.

Looking back, it was comparatively easy to build up the first testing infrastructure using shell scripting, and ACT's ability to produce paired tests substantially decreased the amount of tests needed while ensuring meaningful coverage. This made the testing process efficient, as it avoided the exhaustive nature of full combinatorial testing. However, creating test cases that fulfilled the test oracle was difficult because it necessitated carefully structuring inputs and expected outputs that corresponded with the specifications.

Managing special characters, like tabs, across several contexts and handling edge cases, such incorrect CSV data, caused challenges as well. Several problems would have been avoided if the Python program had included a more reliable argument validation method. Additionally, using an integrated testing framework such as pytest could have made the process go smoothly, especially when handling complex edge cases and JSON outputs. Pairwise testing effectively revealed important issues with data consistency and argument handling, but there are definitely cases that I did not cover.