Countries_CSV to HTML and LaTeX formats

Practical Work I

Subject

Processing Languages

Teacher

Alberto Simões

Student

João Rodrigues, 16928

LESI

Summary

Present report, is meant to document an academic work.

The project was meant to work with a csv file, in order to generate an HTML and a LaTeX file, using regular expressions.

Index

Contextualization

As mentioned on summary, this project has academic purposes, having origin on a practical work to be evaluated on the 2nd year subject, **Languages Processing**, from the course LESI – Bachelor on Computer Systems Engineering.

Project consists on applying content taught on the subject's class, where the student should, mainly, be able to create regular expressions that adapt to the situation in hands, and apply them on a lex compiler. Besides that, the program is meant to be developed in Python and generate an HTML and LaTeX file, with the format the student intend to.

The work was given from the subject's teacher, Alberto Simões, on the current lective year of 2020/2021, and enunciation is available on this project's git, in Portuguese language.

Topic picked

For implementation, it was given 3 possible themes, where the main objective was the same, as described on previous chapter, with the recognition of tokens. What changes from theme to theme, is the file format.

To choose the theme, it was created a word document, to preview possible ways to represent the data after processed. For theme C, it wasn't found a decent way to do so, so B started to be developed to check how would be its implementation, and ended up being picked up.

Theme B consists on processing csv files with the follow characteristics:

- Lines started with character ***#***, are considered comments;
- A field may be started with quotation marks, which is used to have commas on a value, as csv used commas to separate values;
- User must be able to choose columns to be shown.

Mentioned previews are available on project's git, under the name "*ideia_tp.docx*".

Approach

Being the tokens' recognition and generation of HTML and LaTeX files the main objective, the project was guided towards that direction.

Regular Expressions

This first step was to develop regular expressions to cover csv's file needs. For this step was used an online website to help on verifying if the expressions were matching what was intended – **Rubular** (https://rubular.com/). At first, it was identified the followed tokens:

- **QMFIELD –** Which is a shortly representation of "Quotation Marks Field". This token is meant to match strings inside quotation marks. The reason for dividing this field from non quotation marks ones, is simply to make things simplified, as quotation marks are optional, and its characters are being removed before being stored.
- **NFIELD –** Short representation of "Normal Field", which is simply a string without quotation marks.

- **COMMA –** Identifies commas. This match is being used to increment an index, which will be explained later.

- **NEWLINE –** Identifies when line is going to change. This match is being used to restart an index, and save a line's information, but this will be detailed later in the document.

Now, this is all good as regular expressions, they identify what is meant to be identified, but something is lacking, which is the meaning of the values we have in hands. But this can only be explained after breaking down the structure used.

A class for each instance

A csv file is described, not only, for having values separated by commas on each line, but also for having the first line as columns' names and the rest of lines as values, where a line could be a record or an instance. So, having this is mind, we can say each line, after the first one, is an object, where each column is an attribute, and each cell is its value.

Based on that, it was created a file to hold the class called Country, on *country.py* file. Initially, columns' list was being populated manually with columns values of a file, which later was changed to a dynamically approach. So, in the end, class Country has simply an empty list, and 2 functions: one to clean an instance *clean()* and one to simply present object's values for debug needs *present()*.

But if class Country is simply that, how the values are added to its attributes, if the attributes are not specified on the class? Well, it turns out that Python has a cool method called *setattr()*, which assigns a value to a class attribute and, if that attribute wasn't created on class yet, the method is responsible to do so. This question really made me think it wouldn't be possible, based on previous experience with other programming languages, so was a surprise when I found that, when what I was searching, was how I could add to an attribute, having it in string, so I couldn't do class."attribute" = value .

At this point, we know how we can assign values into an attribute that wasn't defined on the class, and to reach its value, it's also possible to use *getattr()*, which is used on *present()* method, in case you were thinking how that method was working without any attributes to work with.

Gather Info vs Gather columns

To add the values into an object, the object needs to know in which attribute, that value belongs to, so, there's a need to find out first, which are the class attributes, which of course, are the first line, but if they are so, with the tokens presented before, we would have 2 functionalities for same matched tokens, first line – gather columns' names (= class's attributes) and the rest of lines, store data into attributes.

At first, the only idea was to create another python file to simply work the first line, and so, have the other working the rest of the text. This approach isn't very appealing as it demands the creation of another file, where its being used same regular expressions.

After asking for help to the teacher, I found out that states would be the solution I would implement. States gives a meaning/context to the data retrieved, and that was what was missing on the final of the Regular Expressions subchapter.

States: header + body

All that was needed for the needs in mind, was store columns' names in one context, and store values on the other context. With that in mind, header was assigned as the state for the columns' names and body, for the values.

In terms of implementing, its simply just to define on a list of states, where the value ‘exclusive’ was assigned to both, as the token should only be recognized in one environment and not more than one, where the tokens would be the same, because on both, we have same string’s formats, it’s all about the different operations we want to do on each.

For this to work logically, *lexer* starts on the state of header and it changes to state of body, when a newline (‘\n’), is found. It was created a method called *t_DUMMY()*, merely to define the state of INITIAL, which wasn’t something I said it exists on the states’ list, but is something that exists by default, and demands to be defined, so this function is only there, so it has any rule, which means it didn’t necessarily have to be a function, it could simply be a regular expression.

Storing data into objects

Each country found, is stored on an object called Country, and each is allocated on a list called *countries*. So far, we would be able to retrieve the columns’ names (=attributes) and values, and we have the *setattr()* method to do so, but only with *lex*, we aren’t able to say on each token, which attribute it should store the value, on, as all allow same format, between them. That’s when the index comes in, and that’s why Country class has an empty list called *columns*. On first line, each value is stored into that list, so that, then, when it goes to state of *body*, where it has the occurrences’ values, it can use an auxiliar variable *column_index*, to navigate through that list, on each line.

The way *column_*index works, is simple, when the instance of CountryLexer – class where lex functions are defined and all processing is made – is built, *column_index* is initialized as 0. When a comma is found, this index is incremented by 1 value and when a newline appears, it returns to 0 value, so in the next line it’s restarted, to iterate the list of columns.

The store on object occurs on tokens **NFIELD** and **QMFIELDS**, as they’re the unique tokens, matching strings. The store of an object to the list of countries, is made on **NEWLINE** and on **EOF**, which was a token, then, created to match the end of file. This **EOF** is used in case the last character of a string or file, is not ‘\n’, which, then, would make the program lose the last record.

On storing an object to a list, it’s created a copy, using the function *copy.deepcopy()*, so that the variable is passed by value and not by reference, which allow the reutilization of same object on running program. For safety, the object has its attributes as **None**, after storing.

CSV Comments

It’s considered a comment, a line starting with a ***#*** character, so on other positions, ***#*** is part of the string of its field. In terms of implementation, there’s a filter for strings that start with cardinal and another for cardinal on other position.

Cardinal on beginning of string

For cardinal at beginning, there’s the token ***HASHTAG***, which is defined in both *header* and *body* states. This token is responsible for leading with such strings, and so, it verifies whether the field that is reading is the first one or not. Such verification is made via *column_index*, which is 0 when in the first field.

On first field, means we’re leading with a comment, so, the processor is routed to a new state, called **COMMENT**. **COMMENT** is responsible to lead with the comment section, by recognizing the rest of the line where the comment is made, and return back to the state it was, previous, in. To return to previous state, it stores on an auxiliar variable *previous_state*, the state the program was in, before jump to *comment* state. This return is made, once it reaches a newline.

Cardinal on middle or end of string

Previously, this character wasn’t being recognized, so it was added to **NFIELD** and **QMFIELD** to do so. Now, in order to these expressions not recognize the situation before (a cardinal on beginning of string), they were positioned after **HASHTAG** token.

This differentiation also allows the tokens to keep their functionality and not having to change it to identify if is leading with a comment or not, they just simply save it as a field.

User input columns to be chosen

One of the requirements, was for the user to be able to pick the columns he wanted to show up. For presentation, it was decided to use first column as a header, and the rest of columns’ values would be itemized as information for it. Having this context, it wouldn’t make a lot of sense if the user didn’t use first column, as it would take out meaning from the information being shown, so, it’s defined that the first column is always shown.

Way of choosing

To avoid the input of inexistant columns and make things easier for the user, the question for columns is made after the lex process, where the files’ columns are already known. With such information, the columns available are shown, and the user writes 1 by 1, the columns he pretends, being warned if he picks first column (just to say that it's an obligatory column) and when writes an inexistant column. This process ends when is sent a space character. To retrieve user input, is being used the function *input()*.

File input

User is able to choose the file he intends to be processed, by writing its name after the program name when calling on command line. For this, it's used the function *sys.argv*.

Files generation

Both HTML and LaTeX generators are defined with same structure, on a class, just varying the syntax that each has. On following subtitles it's explained the present processes.

Initialization

The file being originated is populated with document information, defined below:

- **HTML**
    - **Type of document** - HTML
    - **Page language** - English
    - **Character Encoding** - UTF-8
    - **Title** – Browser's window name
    - **Page title** – It's used <h1> tag

- **LaTeX**
    - **Type of document** – A4 page (size), book (document format)
    - **Title font** – to make it bigger
    - **Title**
    - **Start document**
    - **Show title**

Fill Data

After the initialization, is time to add content to the files. As before, here's the structure:

- **HTML**
  - **Country name** – emphasized with <h2> tag
  - **Itemization of columns and its values**
    - **Start itemization** - <ul> tag
    - **Individual point** - <li> tag
    - **Column name** – Bolded with <strong> tag
    - **Value** – followed by a colon.
    - **End itemization** - </li> tag

- **LaTeX**
  - **Country name** – emphasized with \section to be a chapter and \Huge to make it bigger
  - **Space between this header and its components** - \vspace for that purpose
  - **Itemization of columns and its values**
    - **Start itemization** - \begin{itemize}
    - **Individual point** - \item
    - **Column name** – Bolded with \textbf
    - **Value** – followed by a colon.
    - **End itemization** - \end{itemization}

End document

For final, the document must be closed. On HTML, it's closed *<body>* and *<html>* tags, while on LaTeX it's really closed document, by using *\end{document}*.

Process automatized for user

For practical use, there's a function on both classes, to call all functions mentioned before, which is called: *auto_fill_file()*.