



UNIVERSIDADE DO MINHO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA

Processamento de linguagens - Trabalho Prático #1

Ano Letivo 2021/2022

Gonçalo Pereira (a93168)

Tiago Silva (a93277)

22 de setembro de 2022

1 Introdução

O presente relatório refere-se à realização do trabalho prático 1, cujo o objetivo era de um total de três problemas escolher um e proceder à implementação de uma solução. O enunciado escolhido foi "Ficheiros CSV com listas e funções de agregação" que, resumidamente, requer a implementação de um conversor de um ficheiro CSV para formato JSON, utilizando o Python e os seus módulos `re` e `ply` para gerar os filtros de texto.[1][2][3]

2 Análise do problema

Em primeiro lugar, antes de qualquer implementação foi necessário estudar os possíveis formatos do ficheiro CSV de input e o formato JSON esperado de output.

2.1 Características do CSV

Todos os campos de uma linha do ficheiro csv utilizam uma vírgula como separador de campos. O ficheiro csv está dividido em duas partes: o cabeçalho e a informação. O cabeçalho corresponde à primeira linha do csv e é obrigatório, pois este identifica as colunas e possíveis operações sobre a informação das mesmas. Todas as linhas seguintes ao cabeçalho são essa informação. Tanto o cabeçalho como as linhas de informação têm que ter o mesmo número de separadores. Operações:

- $\{M\}$, este operador indica a informação da coluna a que se aplica é uma lista com M elementos
- $\{N,M\}$, este operador indica a informação da coluna a que se aplica é uma lista com no mínimo N e no máximo M elementos
- $::$, este operador sinaliza a aplicação de uma função aos dados da coluna a que se aplica

- **::sum** , soma todos os elementos (função aplicada a uma lista)
- **::media** , realiza a média aritmética dos elementos a que se aplica função aplicada a uma lista)
- **::quadrado** , eleva todos os elementos da lista ao quadrado
- **::max** , seleciona o maior elemento de uma lista
- **::min** , seleciona o menor elemento de uma lista

De notar, que para o campo o csv permite qualquer caractere excepto as virgulas, e ainda assim, caso o campo esteja delimitado por aspas, até virgulas são permitidas. [4]

2.2 Características do JSON

O ficheiro json que é gerado a partir do csv vai ser uma lista de dicionários em que cada linha do csv corresponde a um dicionário no ficheiro json. Enquanto o csv permite praticamente tudo nos seus campos, o json é um pouco mais restrito. Como o json utiliza as aspas que abrangem os nomes e os valores, aspas no no meio destes não são permitidos, para além disso, não são permitidos também \n. Na resolução da aplicação decidimos contar como inválidos os csv que apresentem essas características nos seus headers, já se o problema estiver apenas nas linhas, estas serão simplesmente descartadas, mas o csv será na mesma convertido para json. [5].

3 Resolução

3.1 Interpretação do cabeçalho

Como já referido, o cabeçalho corresponde à primeira linha do csv. Essa linha é a que dá contexto a toda a informação do csv, pelo que é a primeira a ser lida e interpretada pelo analisador léxico (tokenizer) desenvolvido com recurso à biblioteca PLY.[3] O tokenizer é responsável por analisar o cabeçalho e ao mesmo tempo ir montando uma expressão regular que é capaz de dar match a linhas válidas de informação do csv que estejam de acordo com os critérios do cabeçalho. Caso verifiquemos que o header é inválido o programa para e não é feita a conversão para json.

Por exemplo o header, **Número,Nome,Curso,Notas3,5::max,,,**, passado no **Tokenizer** gera a expressão regular `(?P<a>([^\n,]+)|("[^\n"]+"))?(?P([^\n,]+)|("[^\n"]+"))?(?P<c>([^\n,]+)|("[^\n"]+"))?(?P<d>(\d+(\.\d+)?)|(\d+(\.\d+)?)|(\d+(\.\d+)?)|(\d+(\.\d+)?)?)` que dará match com o resto das linhas do csv.

Em seguida, será possível perceber como esta expressão é construída no **Tokenizer**.

3.2 Tokenizer

O tokenizer que desenvolvemos tem 3 tokens: **FIELD**, **LISTFIELD** e **SEPARATOR**. Criamos, ainda, 6 variáveis que auxiliam a criação e o controlo da expressão regular.

3.2.1 Variáveis

As variáveis criadas são:

regex, string com a expressão regular a ser construída;

fields, uma lista com os tuplos (de 3 elementos) com algumas informações do campos do header;

nfields, número de campos do header (usada para posterior validação do header);

ncommas, número de vírgulas (usada em comunhão com a nfields para posterior validação do header);

nlistcommas, número de vírgulas das listas (controla as vírgulas que aparecem posterior às listas);

id, identificador do campo (incrementa de 'a' para 'z', e quando atinge 'z', é adicionado um 'a' à esquerda).

3.2.2 Separator

Começando pela definição do campo mais simples, temos o **SEPARATOR**, em que a expressão regular é apenas ***r'***, ',', ou seja, uma vírgula.

Quando reconhecido, este verifica se a variável ***nlistcommas*** está a zero, caso esteja, adiciona uma vírgula ao ***regex*** e incrementa em um o número de vírgulas (***ncommas***). Caso contrário, apenas decrementa em um o ***nlistcommas***.

3.2.3 Field

Como visto anteriormente, o csv aceita como campo tudo aquilo que não for vírgulas, uma vez que são o seu separador. No entanto, ainda possibilita ter vírgulas no campo caso este esteja todo envolvido em aspas. Queríamos deixar o campo ter aspas, por exemplo: Nom"es, no entanto o json não aceita. Tendo isto em conta, a definição do token **FIELD** possui a expressão regular ***r'([^\",]+)|("[^\"]+")'***.

Quando captado, como representa um campo que pode ser qualquer coisa, é adicionado ao ***regex*** ***r'(?P<id>([^\",\n]+)|("[^\n"]+))?'***, sendo ***id*** a variável criada no lexer. De notar que, como json não suporta ***\n*** impedimos a captura desse caractere, e apesar de não permitimos no header, permitimos campos vazios no conteúdo das colunas, daí haver um ***?*** no final da expressão. Gostaríamos de colocar o valor do token captado, em vez do identificador, no entanto, o modulo re do python é muito restrito no que toca ao nome que podemos atribuir aos grupos.[2] Adicionamos ainda, à lista ***fields*** um tuplo com o ***id***, o valor do token captado e uma string vazia que identifica que é um campo normal. Por fim, é incrementado o ***id*** e o ***nfields***.

3.2.4 Listfield

Passando ao token mais complexo do tokenizer, temos o **LISTFIELD**. Os campos de listas são do tipo **NOME{MIN,MAX}::FUNÇÃO**, sendo que, o **MAX** e o **::FUNÇÃO** são opcionais. Vendo a sua expressão regular por partes, primeiro, temos o **NOME**, que é reconhecido da mesma forma que no **FIELD** com a expressão ***r'([^\",]+)|("[^\n"]+)"'***, em seguida, uma vez que **MIN** e **MAX** são inteiros **{MIN,MAX}** é captado pela expressão regular ***r'\{[0-9]+(,[0-9]+)?\}'*** e por último, **::FUNÇÃO** com a expressão ***r'(:[a-zA-Z_]+)?'***. Juntando tudo, temos a expressão que reconhece o token **LISTFIELD**, ***r'([^\",]+)|("[^\n"]+)"\{[0-9]+(,[0-9]+)?\}(:[a-zA-Z_]+)?'***.

Quando captado, como representa uma lista temos que construir o regex de forma a capturar todos os elementos da mesma. Primeiro temos que ter em conta que, caso haja **::FUNÇÃO**, então, é obrigatório que os elementos da lista sejam números, uma vez que, só reconhecemos funções que mexem com números.

Visto isto, podemos prever dois tipos de elementos na lista:

num: ***r'((\d+|\d+\.\d+)?)'***, apenas números.

abc: ***r'([^\n,]+)'***, qualquer tipo de elemento.

Começamos a construir a expressão com ***r'(?P<id>'***, assim como fazíamos no **FIELD**, para o grupo ser identificado facilmente mais tarde, e, em seguida, adicionamos á expressão o reconhecimento dos elementos obrigatórios. Para isso adicionamos **elemento + *r'***, ' (lembrando que

elemento é **num** ou **abc**) MIN-1 vezes e **elemento** no fim. Quanto aos elementos opcionais, controlados pelo MAX, adicionamos ao regex `r',' + elemento + r'?'` (o elemento é opcional, mas a vírgula não) MAX-MIN vezes. Por último, concluímos a construção do regex com `r')'`, fechando o parênteses no início.

Como já colocamos as vírgulas na expressão regular, as próximas capturadas pelo PLY, não podem ser adicionadas, por isso colocamos o *nlistcommas* com MAX para depois no **SEPARATOR** não ser colocado essas vírgulas.

À lista *fields* adicionamos um tuplo com o *id*, o valor do token captado e o **::FUNÇÃO**. Caso seja uma lista normal, metemos uma string `"::"`, para diferenciar mais tarde, dos campos comuns.

Por fim, assim como no **FIELD**, incrementamos o *id*, e ainda, incrementamos o *nfields* com MAX (caso não exista MAX, acrescentamos com MIN)

3.3 Escrita do ficheiro Json

Após a análise do cabeçalho pelo tokenizer, é obtido uma expressão regular que dá match com uma linha de acordo com os critérios do cabeçalho e uma lista de tuplos na variável *lexer.fields* que dão informação e contexto de cada coluna. É feita a leitura para memória do resto do ficheiro e realiza-se um **finditer** ao ficheiro carregado para memória usando a expressão regular gerada pelo tokenizer. Assim é obtido do **finditer** uma lista de match objects, em que cada match object tem o conteúdo de uma linha devidamente agrupada e classificada como definido na expressão regular. Para ser feita a conversão recorre-se a dois ciclos aninhados, o de fora itera sobre a lista de match objects e do de dentro itera sobre a lista de tuplos. A cada match object é gerado o seu dicionário com a aplicação do método **groupdict()** e a lista de tuplos é percorrida. A cada iteração da lista é verificado o segundo índice do tuplo para ver o tipo de coluna (normal, lista, lista com sum, etc...), esse valor, sinaliza a forma como os valor têm que ser processados para JSON. O índice 0 fornece o id para obter-se o valor do dicionário daquela coluna e o índice 1 fornece o nome da coluna, com estas duas informações é escrita uma entrada do dicionário. Ao fim de se iterar a lista obtém-se um dicionário escrito em JSON que corresponde a uma linha do CSV. O processo repete-se até se chegar ao fim da lista de match objects, isto é, até todas as linhas do CSV terem sido convertidas e escritas em formato JSON. Importante salientar que a conversão obedece a todos os critérios de um ficheiro JSON definidos na secção 2.2.

3.4 Demonstração

DataSet CSV:

```
Número,Nome,Curso,Notas{3,5}::max,,,
3162,Cândido Faísca,Teatro,12,13,14,,
7777,Cristiano Ronaldo,Desporto,17,12,20,11,12
264,Marcelo Sousa,Ciência Política,18,19,19,20,
```

JSON:

```
1  [
2    {
3      "Número": 3162,
4      "Nome": "Cândido Faísca",
5      "Curso": "Teatro",
6      "Notas": 14.0
7    },
8    {
9      "Número": 7777,
```

```

10         "Nome": "Cristiano Ronaldo",
11         "Curso": "Desporto",
12         "Notas": 20.0
13     },
14     {
15         "Número": 264,
16         "Nome": "Marcelo Sousa",
17         "Curso": "Ciência Política",
18         "Notas": 20.0
19     }
20 ]

```

DataSet CSV:

```

date,home_team,away_team,"home_score,away_score"{2},,tournament,city,country,neut
1872 - 11 - 30,Scotland,England,0,0,Friendly,Glasgow,Scotland,FALSE
1873 - 03 - 08,England,Scotland,4,2,Friendly,London,England,FALSE

```

JSON:

```

1  [
2      {
3          "date": "1872-11-30",
4          "home_team": "Scotland",
5          "away_team": "England",
6          "home_score,away_score": [0, 0],
7          "tournament": "Friendly",
8          "city": "Glasgow",
9          "country": "Scotland",
10         "neutral": "FALSE"
11     },
12     {
13         "date": "1873-03-08",
14         "home_team": "England",
15         "away_team": "Scotland",
16         "home_score,away_score": [4, 2],
17         "tournament": "Friendly",
18         "city": "London",
19         "country": "England",
20         "neutral": "FALSE"
21     }
22 ]

```

4 Conclusão

Um vasto número de aplicações implementam conversores e analisadores léxicos e fazem de uso dos mesmos métodos usados no desenvolvimento deste trabalho prático. Qualquer que seja o caso é necessário haver sempre uma análise metódica do que é a sintaxe do que tem que ser

convertido para que seja definido o seu conjunto de tokens e de regras. Com a realização deste trabalho, acreditamos ter desenvolvido uma melhor capacidade na escrita de expressões regulares para a descrição de padrões em streams de texto, e percebemos a versatilidade funcional que estas oferecem, no desenvolvimento de processadores de linguagem regulares. Conseguimos ainda, com a utilização de Python, um melhor domínio sobre os módulos re e ply na geração de filtros.

Referências

- [1] <https://docs.python.org/3/howto/regex.html>
- [2] <https://docs.python.org/3/library/re.html>
- [3] <https://www.dabeaz.com/ply/>
- [4] <https://datatracker.ietf.org/doc/html/rfc4180page-2>
- [5] <https://jsonformatter.curiousconcept.com/#>
- [6] <https://regex101.com/>