Relatório Projeto PL

Grupo 60 :
António Luís de Macedo Fernandes (a93312)
José Diogo Martins Vieira (a93251)
João Silva Torres a93231

Março 26, 2022


Contents

1 Introdução

No âmbito do desenvolvimento do primeiro projeto da unidade curricular de Processamento de Linguagens foi-nos proposto desenvolver um conversor de ficheiros CSV para o formato JSON.

Para tal, tivemos de por em prática o que nos foi ensinado durante as aulas, de forma a representar o nosso conhecimento. Dito isto, fizemos uso de expressões regulares e filtros de texto. A linguagem utilizada para realizar este trabalho foi Python.

Ao longo deste relatório, iremos abordar quais as estratégias utilizadas pelo grupo, bem como alguns testes realizados.

2 Estratégias utilizadas

2.1 Ficheiro CSV Normais

Primeiramente começamos por converter ficheiros CSV normais, sem nenhum dos requisitos adicionais. Para tal, começamos por definir inicialmentes os diferentes tokens para posteriormente, com recurso do Lex, passarmos à leitura do ficheiro. Utilizamos então os seguintes tokens:

- SEP - Separador vírgula (',')
- TEXT - Tudo o que seja valor
- NEWLINE - Mudança de linha

As expressões regulares utilizadas na definição destes tokens foram:

- SEP - `r','`
  Apenas dá match com o caratér ','
- TEXT - `r'(?P<quote>[\"\']).+(?P=quote)|[\^,\n]+'`
  Dá match a tudo o que esteja entre quotes (`"` ou `'`) ou tudo o que não seja vírgula ou `\n`
- NEWLINE - `r'\n+'`
  Dá match com o caratér `\n`, uma ou mais vezes, caso haja linhas vazias para serem ignoradas

Temos assim definidos os tokens necessários para ler o csv.
Falta ainda diferenciar os headers (primeira linha do ficheiro), dos restantes valores.
Para tal, criámos o seguinte estado do lexer.

```
("header","exclusive")
```

(Este estado vai servir para diferenciar o tratamento do header com os restantes valores.)
O lexer vai ter três variáveis nossas:

- header - que guarda a informação do header.
- line - que guarda a informação da linha atual.
- values - que guarda a informação de todas as linhas.

Começamos por indicar ao lexer para inicializar no estado header. Deste modo, na primeira linha, quando existir um match para o token TEXT, o valor deste irá ser adicionado na lista de headers. Quando chegar ao fim da linha, ou seja, quando há match no token NEWLINE, já temos a nossa lista dos headers completa. De seguida, iremos passar para o estado "INITIAL", o estado default para o resto das linhas. Agora, para cada match do token TEXT, iremos adicionar na lista "line". Quando chegamos ao fim dessa linha, adicionamos a lista "line" à

lista "values", e inicializamos a lista "line" numa lista vazia. Repetindo-se assim este processo para o resto das linhas. Desta forma, quando o lexer atingir o fim do ficheiro, iremos utilizar duas listas das três que temos:

- headers - que contém o nome de todos os headers
- values - que guarda uma lista dos valores para cada linha

Posto isto, passámos à construção do dicionário.

Para isso, iremos percorrer a lista dos values de forma a tratar de uma linha de cada vez. Pegando assim em cada linha, iremos percorrer a linha juntamente com os headers e associar o header ao respetivo valor. Repete-se este processo para todas as linhas e temos o nosso dicionário completo.

2.2 Listas

2.2.1 Adição em Listas

Para adicionarmos conteúdo numa lista, tivemos de acrescentar um novo token (MULT). Este tem como objetivo indicar o número de colunas que a lista vai abrangir.

- MULT - `r'{(?P<number>\d+)}'`
  Dá match quando um número se encontra dentro de chavetas (`{}`).

NOTA : Utilizamos um named group (number) para posteriormente ir buscar o resultado capturado.

Este token é apenas utilizado no header de um ficheiro. Quando houver um match, vemos qual o valor que está contido no grupo number (acima mencionado). De seguida, vamos buscar o último elemento da lista headers , uma vez que sabemos que é esse o elemento a que se está a referir. Posteriormente, adcionámos esse elemento N-1 vezes à lista de headers. (N é o número obtido no grupo). Desta forma, sabemos que na construção do dicionário, este tipo de elementos é para ser colocado numa lista. Para tal, antes de se colocar o elemento no diconário, verificámos se o header que vamos adicionar é o único na lista de headers. Se for, então procedemos normalmente. Se não for, verificamos se já existe alguma correspondência paraa esse header. Se não existir correspondência, criámos uma lista e adiciona-se o elemento na mesma. Caso exista correspondência, apenas adicionamos o elemento no final da lista.

Deste modo, conseguimos fazer a adição correta dos elementos em lista.

2.2.2 Listas com intervalos

Para adicionar este requisito, a nossa antiga definição para o token de MULT não era suficiente, precisa de alterações.

Fizemos as devidas alterações para verificar o intervalo:

- `r'{(?P<number>\d+)(,(?P<number2>\d+))?}'`

Desta forma o grupo number captura o primeiro número, e o grupo number2 captura o segundo.

Desta, guardamos o valor do grupo number como o nosso valor minimo e de seguida verificamos se existe o grupo number2. Caso exista, guardamos esse valor como o nosso valor máximo. Caso não exista, o nosso valor máximo será igual ao valor mínimo.

Anteriormente ficávamos sempre à espera de um determinado número de valores, agora esse valor pode variar. Para isso, precisamos de saber quantos valores estão em falta.

Para resolver este problema, criámos o token SKIP. Este token tem como função identificar quando um valor se encontra vazio.

Como é sabido, não se pode identificar strings vazias. Porém sabemos que quando um valor está em falta, três coisas podem acontecer:

- Pelo menos 2 vírgulas seguidas no meio da frase
- Pelo menos 1 vírgula e um seguidos
- Pelo menos 1 vírgula no ínicio da linha

Para verificar estas três condições a expressão regular para este token é a seguinte:

```
r'(?m)((?P<init>\^,+)|,{2,}\n*|,\n)'
```

Nota: é utilizado (?m) para âar match no ínicio de cada linha e não apenas da string

Esta expressão regular vai verificar se existe uma vírgula ou mais no inicio da lista ou 2 ou mais vírgulas seguidas (podendo neste caso conter um nova linha ou o caso de uma vírgula seguida de uma nova linha)

Antes de abordar como bão ser tratados os matches deste token, vamos referir uma pequena alteração feita relativamente aos headers.

Anteriormente, caso um header fosse uma lista íriamos repetir esse elemento na lista de headers e na criação do dicionário verificar-se-ia se o elemento era único ou não na lista. Ora, decidimos melhorar esta abordagem, já a pensar também na adição da função. Alterar a lista de headers, de uma lista com apenas nomes para uma lista de pares compostas por nome e um par com o número de colunas. Deste modo, ao adicionar um valor para a lista de headers adicionamos com o valor e (1,1), e caso depois se verificar que esse elemento irá ser uma lista, alteramos esse valor de acordo.

Desta maneira também se torna mais simples a construção do dicionário. Uma vez que apenas se precisa de verificar se o segundo elemento do par é (1,1) ou não. Caso não seja, vai se percorrer a lista da linha e adicionar os respetivos valores, o número de vezes que se encontra no par. Caso o número de elementos na lista seja menor que o valor minimo da lista, a lista não é adicionada ao dicionário.

Feitas estas alterações, iremos agora abordar o que é feito quando ocorre um match do token SKIP.

Primeiramente, é importante referir que a definição deste token deve se encontrar por cima da definição do token SEP. Pois caso esteja em baixo deste, nunca irá ocorrer match do SKIP,

pois o SEP dá match apenas com ’,’. Desta forma ao meter a definição primeiro, garantimos que verifica primeiro o padrão do SKIP e só se este falhar, é que dá match no SEP.

Quando há um match, começamos por verificar quantos valores estão em falta e para isso fazemos uso do seguinte cálculo:

```
count = (match.count(’,’)) + (’\n’ in match) -1
```

Este cálculo é feito caso o match não seja no inicio da linha. Caso seja, apenas se verificar o número de vírgulas.

De seguida, adicionamos None à lista da linha, conforme os números de valores em falta. Tendo o cuidado de invocar a função que trata da newLine, caso um newLine se encontra no match.

Para a criação do dicionário, quando se trata de uma lista, iremos verificar se é None ou não, e caso seja não se adiciona à lista e não percorre mais a lista à espera de valor para esse header.Deste modo, conseguimos implementar a falta de números quando se trata de uma lista.

2.3 Funções de agregação

Para este requisito final, tivemos a necessidade de criar mais um token (FUNC). Este token tem como objetivo identificar a função a que vai corresponder o header.

Para tal usamos a seguinte expressão regular:

```
r’::(?P<func>\w+)’
```

Dá match quando existe dois pontos seguidos de uma palvra.

Utilizamos um named group, para posteriormente ser mais fácil capturar o valor contido no grupo.

Para implementar este requisito, tivemos que realizar algumas alterações na nossa lista de headers.

Esta vai passar de uma lista de pares, que guarda nome e número de colunas, para uma lista de triplos. Neste triplos guardará, não só o que guardava anteriormente mas também a uma lista com as funções associadas.

Para tal, primeiramente quando adicionamos um valor à lista de headers, adicionamos com este campo da função a a uma lista vazia (`[]`).

Desta forma, quando existir um match do token FUNC, iremos alterar o terceiro elemento deste triplo, adicionando o valor guardado no grupo capturado func.

Posteriormente na criação do dicionário, após ter todos os elementos de uma lista prontos a ser adicionados, verificamos se há alguma função associada a esse header.

Se o campo for [], adiciona-se a lista normalmente. Caso contrário, percorremos a lista, verificamos o nome da função e calculamos o valor resultante. Assim, em vez do nome do header, associa-se o nome do header juntamente com o nome da função separados por um underscore e adicionámos ao valor resultante.

As funções que implementamos foram as seguintes:

- sum - Somatório da lista
- media - Média da lista
- median - Mediana da lista
- mode - O valor mais frequente (moda)
- range - Intervalo de valores (máximo - mínimo)
- all - Todas as funções acima representadas

2.4 Construção do ficheiro JSON

A construção do JSON é feita da seguinte forma. Incialmente, fazemos uso de uma indentação de 4 espaços. Essa identaçao será utilizada para cada linha quando fazemos uso da função spaces:

De seguida percorremos a lista de dicionários anteriormente criada. Para cada dicionário, iremos percorrer os seus itens como pares (key,value) com recurso da função items. Para cada par, verificamos qual o seu tipo para saber como temos de o escrever. Existem três tipos que podem ser reconhecidos:

- float/int
- list
- str
- None

Para os números,escrevemos o valor sem aspas. Para a lista, escrevemos os valores entre parenteses retos e separados por vírgulas. Para a str, escrevemos o valor com aspas. Paea o None, escrevemos null. Para todos, a key é sempre entre aspas seguida de ":".

3 Testes

alunos.csv

```
Número,Nome,Curso
3162,Cândido Faísca,Teatro
7777,Cristiano Ronaldo,Desporto
264,Marcelo Sousa,Ciência Política
```

Figure 1: CSV Normal (alunos.csv)

alunos.json > ...

```
[
    {
        "Número": 3162,
        "Nome": "Cândido Faísca",
        "Curso": "Teatro"
    },
    {
        "Número": 7777,
        "Nome": "Cristiano Ronaldo",
        "Curso": "Desporto"
    },
    {
        "Número": 264,
        "Nome": "Marcelo Sousa",
        "Curso": "Ciência Política"
    }
]
```

Figure 2: Resultado JSON (alunos.json)

alunos2.csv

```
Número,Nome,Curso,Notas{5},,,,,
3162,Cândido Faísca,Teatro,12,13,14,15,16
7777,Cristiano Ronaldo,Desporto,17,12,20,11,12
264,Marcelo Sousa,Ciência Política,18,19,19,20,18
```

Figure 3: CSV com lista tamanho fixo (alunos2.csv)

```json
[
    {
        "Número": 3162,
        "Nome": "Cândido Faísca",
        "Curso": "Teatro",
        "Notas": [12,13,14,15,16]
    },
    {
        "Número": 7777,
        "Nome": "Cristiano Ronaldo",
        "Curso": "Desporto",
        "Notas": [17,12,20,11,12]
    },
    {
        "Número": 264,
        "Nome": "Marcelo Sousa",
        "Curso": "Ciência Política",
        "Notas": [18,19,19,20,18]
    }
]
```


Figure 4: Resultado JSON (alunos2.json)


```csv
alunos3.csv
Número,Nome,Curso,Notas{3,5},,,,,
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,
```


Figure 5: CSV com lista com intervalo (alunos3.csv)


```json
{} alunos3.json > ...
[
    {
        "Número": 3162,
        "Nome": "Cândido Faísca",
        "Curso": "Teatro",
        "Notas": [12,13,14]
    },
    {
        "Número": 7777,
        "Nome": "Cristiano Ronaldo",
        "Curso": "Desporto",
        "Notas": [17,12,20,11,12]
    },
    {
        "Número": 264,
        "Nome": "Marcelo Sousa",
        "Curso": "Ciência Política",
        "Notas": [18,19,19,20]
    }
]
```


Figure 6: Resultado JSON (alunos3.json)


```csv
alunos4.csv
Número,Nome,Curso,Notas{3,5}::sum,,,,,
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,
```


Figure 7: CSV com função de agregação (alunos4.csv)

Figure 8: Resultado JSON (alunos4.json)

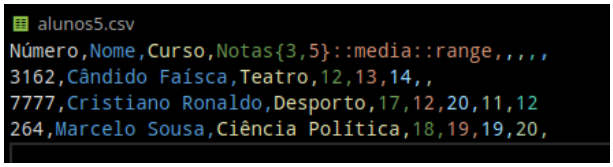

Figure 9: CSV com funções de agregação (alunos5.csv)

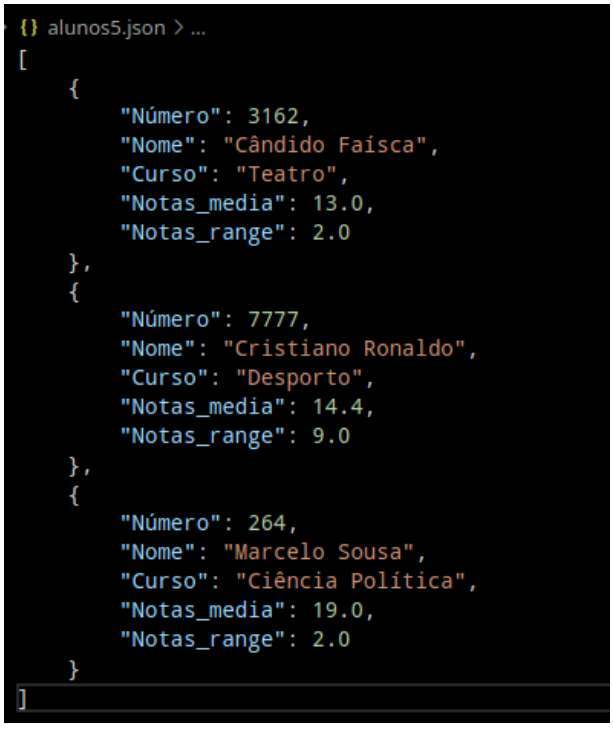

Figure 10: Resultado JSON (alunos5.json)

Para testar também algumas situações que podem acontecer e que nós procuramos ter resposta, como por exemplo:

- Valor em falta para uma coluna
- Valores em falta para o número mínimo de valores na lista
- O separador entre duas quotes

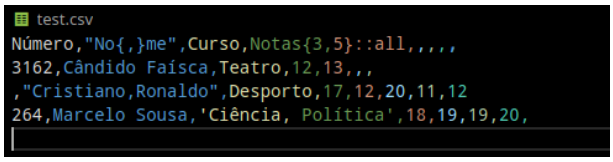

Figure 11: CSV para teste de possíveis erros(test.csv)

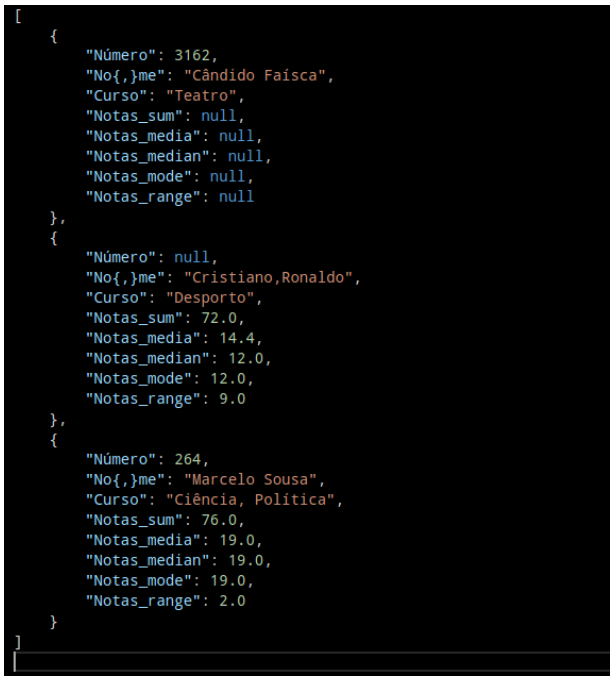

Figure 12: Resultado JSON (test.json)

4 Conclusão

Para concluir, consideramos que, neste primeiro projeto, conseguimos desenvolver com sucesso o que nos foi pedido.

Posto isto, cremos estar mais familiarizados com a linguagem Python e julgamos ter tido uma boa evolução no que diz respeito ao uso das expressões regulares bem como a utilização do módulo ply, mais precisamente, o lex.

Assim, acreditamos que, de uma forma geral, o grupo trabalhou bem e teve um aproveitamento positivo, visto que cumpriu com tudo o que foi pedido.