Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de
Processamento de Linguagens

Ano Letivo de 2021/2022

Trabalho Prático 1
Ficheiros CSV com Listas e Funções de Agregação

Grupo 5
a93234 - Diogo Matos
a83630 - Duarte Serrão
a93208 - Vasco Oliveira

27 de março de 2022

Índice

1 Introdução

O presente relatório, da primeira fase do projeto da cadeira de Processamento de Linguagem, do curso de Engenharia Informática, da Universidade do Minho, visa explicar a metodologia aplicada aquando da conversão de ficheiros **CSV** para ficheiros **JSON** com uso do conhecimento adquirido ao longo do semestre sobre a linguagem de programação **Python**, bem como as suas bibliotecas (re, ply), e conceitos como expressões regulares e analisadores léxicos. Com a realização deste projeto, pretendemos consolidar o nosso conhecimento relativo ao reconhecimento de padrões textuais de modo a obter expressões regulares, a transformação de ficheiros textuais em *tokens* para demais processamento e modificação, bem como aprender novas metodologias para filtragem de texto.

1.1 Enunciado: Ficheiros CSV com listas e funções de agregação

Neste enunciado pretende-se fazer um conversor de um ficheiro CSV (Comma separated values) para o formato JSON. Para se poder realizar a conversão pretendida, é importante saber que a primeira linha do CSV dado funciona como cabeçalho, e que define o que representa cada coluna.

No entanto, neste trabalho, os **CSV** recebidos têm algumas extensões.

Listas

Listas com tamanho definido

No cabeçalho, cada campo poderá ter um número N que representará o número de colunas que esse campo abrange. Por exemplo, imaginemos que ao exemplo anterior se acrescentou um campo **Notas**, com N = 5 ("alunos2.csv"):

Isto significa que o campo **Notas** abrange 5 colunas. (Reparem que temos de meter os campos que sobram a vazio, para o **CSV** bater certo).

Listas com um intervalo de tamanhos

Para além de um tamanho único, podemos também definir um intervalo de tamanhos N, M, significando que o número de colunas de um certo campo pode ir de N até M. ("alunos3.csv")

Funções de agregação

Para além de listas, podemos ter funções de agregação, aplicadas a essas listas. Veja os seguintes exemplos ("alunos4.csv"e "alunos5.csv"):

```
Número,Nome,Curso,Notas{3,5}::media,,,,,
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,
etc.
```

Resultado esperado

O resultado final esperado é um ficheiro **JSON** resultante da conversão dum ficheiro **CSV**. Por exemplo, o ficheiro "alunos.csv"(original), deveria ser transformado no seguinte ficheiro "alunos.json":

No caso de existirem listas, os campos que representam essas listas devem ser mapeados para listas em **JSON** ("alunos2.csv"):

No caso em que temos uma lista com uma função de agregação, o processador deve executar a função associada à lista, e colocar o resultado no **JSON**, identificando na chave qual foi a função executada ("alunos4.csv"):

1.2 Descrição do Problema

O principal problema consiste na transformação de um ficheiro **CSV** para um ficheiro **JSON**, mas para tal é necessário subdividi-lo em diferentes etapas.

Inicialmente, é necessário conseguir identificar os diferentes tipos de itens ou *tokens* que poderão aparecer num ficheiro **CSV**, dos quais será necessário encontrar uma expressão regular que os identificará, bem como possíveis estados diferentes no qual se encontrem.

Após devido tratamento textual é necessário transformar o conteúdo no formato pretendido **JSON**, aplicando as diferentes funções interpretadas aquando da leitura do *header*.

Finalmente, basta imprimir, no ficheiro fornecido, a lista de dicionários pretendida.

2 Modelação da Solução

Para se chegar a uma solução deste problema, teve de haver uma fase de modelação, em que se pensa de uma forma bastante abstrata. De forma a não haver repetição de certos pontos mencionados, optou-se por já se apresentar algumas funções finais nesta secção.

2.1 Primeira Fase - Cabeçalho Simples

Nesta primeira fase, decidiu-se começar pelo cenário mais simples possível, em que cada campo do cabeçalho corresponderá a um campo de um dado registo. De um modo geral, qualquer linha, incluindo o cabeçalho, tomaria a seguinte forma genérica:

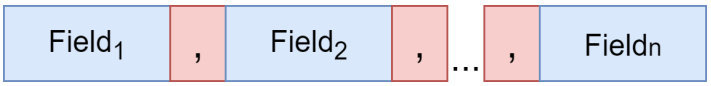

Figura 1: Estrutura básica de um registo.

Tendo por referência a plataforma *CVS Reader*[1], conseguimos obter algumas regras de construção de um registo e os campos que o compõem. É de notar que estas regras foram conseguidas com uma mistura da semântica do Excel e do UNIX.

1. Campos deverão estar separados por vírgulas;
2. Para vírgulas poderem estar incorporadas num campo, este deverá estar delimitado por aspas;
3. Registos deverão estar separados por caracteres de fim de linha (\n);
4. Caracteres de fim de linha nunca poderão estar dentro de um campo, mesmo que este esteja delimitado por aspas;
5. Espaços e tabulações antes ou depois de vírgulas ou delimitadores de registos serão cortados;
6. Para garantir a preservação desses caracteres, o campo deverá estar delimitado por aspas;
7. Espaços e tabulações serão aceites dentro dos campos, mesmo que estes não estejam a ser delimitados por aspas, podendo formar várias palavras;
8. Registos sem dados ou ocorrências de delimitadores de campos deverão ser ignorados;
9. Campos sem dados serão válidos;
10. O último registo poderá ou não ter um caractere de fim de linha.

Como é possível reparar, existe um padrão que se repete até à palavra n, sendo que a última poderá ou não conter uma vírgula, apesar de que na figura 1 tenha-se optado por não representar esse caractere. O objetivo desta fase é o de poder extrair as palavras contidas neste padrão.

2.1.1 Expressão Regular

Inicialmente optou-se por tentar descobrir uma expressão regular em que cada campo seria extraído através de grupos com nomes. De uma forma geral, a expressão regular teria este formato:

```
regex = spaces + textqualified + field  + is_textqualified +
        spaces + comma
```

Nesta expressão, um campo, representado pelo grupo `field`, poderia ser textualmente qualificado ou "cru", sendo assim então representado pelo seguinte grupo:

```
field = r'(?P<FIELD>(?(TEXTQUALIFIED).*?|' + raw_field + '))'
```

O que a expressão quer dizer é que se existe um grupo criado chamado "TEXTQUALIFIED", então aceitará qualquer caractere até à próxima aspa. Se não existir, então procurará um padrão de um campo "cru". A sua expressão regular é:

```
raw_field = r'[^,\n \t]+([ \t]*[^,\n \t]+)+'
```

Basicamente, um campo "cru"começará por um caractere que não pertença à classe dos caracteres brancos nem seja uma vírgula. Após uma palavra, poderão haver espaços pelo meio, garantindo que acabe sempre num caractere não branco, e nunca numa vírgula. Neste relatório diz-se campo "cru", um campo em que não se tem de aplicar regras especiais para qualificar o campo como texto.

Para se aplicar essas regras especiais, criou-se um grupo chamado `TEXTQUALIFIED`, demonstrado pela seguinte fórmula:

```
textqualified = r'(?P<TEXTQUALIFIED>")?'
```

Tudo o que este grupo faz é procurar por aspas. Posteriormente, na expressão regular, encontra-se uma expressão chamada `is_textqualified`. O seu objetivo é verificar se o grupo foi criado, e se sim, então irá procurar por aspas, de forma a acabar um campo textual. A sua expressão será:

```
is_textqualified = r'(?(TEXTQUALIFIED)")'
```

Por fim, foi preciso criar expressões que detetassem espaços entre as vírgulas e as palavras e expressões que detetam as próprias vírgulas, sendo que a última poderá ou não existir.

Estando todas estas expressões criadas, a expressão final seria algo do tipo:
Para se obter os campos, bastaria iterar a expressão regular ao longo de um registo, ou seja, ao longo de uma linha, e extrair o grupo `FIELD`.

```
                    spaces = r'[ \t]*'
                    comma = r',?'
REGEX = r'[ \t]*(?P<TEXTQUALIFIED>")?(?P<FIELD>(?(TEXTQUALIFIED).*?|
          [^,\n \t]+([ \t]*[^,\n \t]+)+))(?(TEXTQUALIFIED)")[ \t]*,?'
```

2.1.2 Estados e *Tokens*

Apesar do método anteriormente mencionado funcionar, não seria o mais indicado para a resolução do problema. É mais complicado de se manter em futuras versões do projeto e não é muito legível, pelo que se optou por explorar outros métodos. Para este novo método, utilizou-se a biblioteca PLY[2], como foi sugerido e fortemente recomendado pelos docentes.

Reutilizando um pouco o trabalho já feito no método anterior, concluiu-se que seria preciso um estado.

```
states = [("TEXTQUALIFIED", "exclusive")]
```

A este estado dar-se-á o nome de `TEXTQUALIFIED`, que será ou não ativado, caso um certo campo esteja ou não delimitado por aspas. De seguida, concluiu-se que seriam precisos três tipos de *tokens*.

```
tokens = ["FIELD", "QUOTES", "BLANK"]
```

Por cada *token*, criar-se-á uma ou mais funções, estando cada uma destas associadas a um estado. Em primeiro lugar, optou-se por fazer as funções associadas ao *token* `FIELD`:

```
def t_FIELD(t):
    r',?[\ \t]*(?P<FIELD>[^,\n\ \t{}]([\ \t]*[^,\n\ \t{}]+)*)'
    t.lexer.fields.append(str(t.lexer.lexmatch.group("FIELD")))

def t_TEXTQUALIFIED_FIELD(t):
    r'[^"]+'
    t.lexer.fields.append(t.value)
```

Um campo poderá ser capturado estando ou não qualificado como texto. Por este motivo, teve-se de fazer uma versão diferente para os dois estados possíveis. Ambos capturam um campo e guardam-no numa lista de campos. A única diferença está no que cada um captura. A primeira função captura o equivalente a um `raw_field` previamente mencionado na secção 2.1.1. Já o segundo capturará tudo o que estiver dentro de aspas. Como se analisa o documento **CSV** linha a linha, em nenhum dos campos, quer estejam ou não qualificados como texto, será aceite um caractere de nova linha.

De seguida, decidiu-se focar-se no *token* `QUOTES`, estando este presente quer no estado `TEXTQUALIFIED` ou não.
Caso esteja no estado inicial e encontrar aspas, entrará no estado `TEXTQUALIFIED`.

```
def t_QUOTES(t):
    r',?[\ \t]*"'
    t.lexer.begin("TEXTQUALIFIED")
```

Caso já esteja nesse estado, ao encontrar aspas de novo, completando o par, transicionará para o estado inicial.

```
def t_TEXTQUALIFIED_QUOTES(t):
    r'"'
    t.lexer.begin("INITIAL")
```

Sobrando apenas um *token*, sendo este o `BLANK`, tirar-se-á proveito de como o `ply.lex` funciona.

```
def t_BLANK(t):
    r',?[\ \t]*""|,[\ \t]*(?=(,))|,[\ \t]*\n'
    if t.lexer.mode == "HEADER":
        t.lexer.fields.append
            ("EMPTY_" + str(t.lexer.count_empties))
        t.lexer.count_empties += 1
    else:
        t.lexer.fields.append("")
```

É possível verificar que este *token* também captura aspas, entrando um pouco em conflito com o *token* `QUOTES`. Sabendo que `ply.lex` tentará capturar sempre o padrão maior, caso encontre duas aspas seguidas, o *token* pertencerá a `BLANK` e não a `QUOTES`.

> "Tokens defined by strings are added next by sorting them in order of decreasing regular expression length (longer expressions are added first)."
>
> (PLY Documentation [2])

Nesta função repara-se em dois novos elementos, sendo estes `mode` e `count_empties`. A razão pela qual foi criada uma variável que guarda o modo, podendo ser "HEADER"ou "RECORD", deve-se ao facto de que um cabeçalho não poderá ter campos com nome igual a outros campos. Por esta razão, optou-se por criar uma nomenclatura que guarde o nome do campo como "EMPTY_"mais a sua instância. Essa instância é guardada e atualizada pela segunda variável mencionada. Caso o modo seja "RECORD", juntar-se-á à lista de campos uma string vazia, indicando assim que este se encontra sem qualquer tipo de dados.

Concluindo assim esta fase, é preciso especificar que elementos se terá de ignorar e como o *lexer* comportar-se-á ao encontrar um *token* não esperado.

Caso esteja no estado inicial, irá ignorar todos os caracteres brancos. Caso esteja no `TEXTQUALIFIED`, então só ignorará novas linhas. Em ambos os casos, caso dê erro, irá imprimi-lo.

```
t_ignore = ' \t\n'
t_TEXTQUALIFIED_ignore = '\n'

def t_ANY_error(t):
    print("Error: " + str(t))
```

2.2 Segunda Fase - Cabeçalho Com Listas

Apesar de no enunciado separar listas com tamanho fixo de listas com tamanho variável, optou-se por fazer estas duas fases numa só, começando-se por descobrir como será a construção de uma lista num cabeçalho. Chegou-se à conclusão de que terá o seguinte formato:

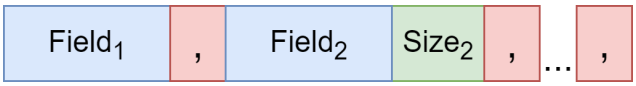

Figura 2: Estrutura básica de um cabeçalho com uma lista.

Uma lista poderá estar inserida no meio de outros campos, mas esta terá sempre de ter vírgulas à frente da secção do tamanho. O número de vírgulas dependerá do tamanho máximo que a lista poderá ter.

O tamanho poderá tomar dois formatos, sendo estes `{SIZE}` ou `{MIN_SIZE, MAX_SIZE}`. É possível verificar que em ambos, existem três elementos em comum: chavetas de abrir , um tamanho e chavetas de fechar, podendo esta informação ser aproveitada na construção da expressão regular.

2.2.1 Dicionário com Tamanhos

Sendo esta uma primeira versão do trabalho, inicialmente optou-se por utilizar um dicionário à parte para guardar os campos que constituem uma lista (chave) e o respetivo tamanho mínimo e máximo (valor em forma de tuplo), da seguinte maneira:

```
def t_LISTSIZE(t):
    r'{(?P<min_size>\d+)(,(?P<max_size>\d+))?}'

    field = t.lexer.fields[-1]
    min_size = int(t.lexer.lexmatch.group("min_size"))
    max_size_str = t.lexer.lexmatch.group("max_size")
    if max_size_str:
        max_size = int(max_size_str)
    else:
        max_size = min_size

    t.lexer.lists[field] = (min_size, max_size)
```

Este novo *token* `LISTSIZE` irá procurar por dois inteiros dentro de chavetas, sendo que o

segundo será opcional. Como se deu um nome a cada grupo de inteiros, de seguida foi extremamente fácil extraí-los do *token* capturado. Como o segundo é opcional, teve-se de verificar primeiro se ele existia. No final guarda-se um tuplo com o intervalo do tamanho da lista.

Posteriormente, ao interpretar uma linha do documento, simplesmente verificava-se se o campo correspondente estava nesse dicionário e, nesse caso, teríamos acesso às suas dimensões.

2.2.2 Cabeçalho com Listas Integradas

Mesmo que a versão anteriormente mencionada funcionasse, optou-se por seguir outra estratégia, em que as listas estivessem integradas juntamente com os outros campos. Começou-se por se adicionar um novo estado.

```
("LIST", "exclusive")
```

O estado `LIST` servirá para processar futuras funções e vírgulas consequentes, de forma a que a função `t_BLANK` não as processe como campos vazios.

De seguida, apenas se acrescentou um novo *token*, ficando a lista da seguinte forma.

```
tokens = ["FIELD", "QUOTES", "BLANK", "LSIZE"]
```

Sabendo que a estrutura de uma lista será um campo, seguido pelo tamanho, pode-se aproveitar o token do campo, sem alterar nada. Logo, para capturar uma lista, só precisaremos de capturar o que define o tamanho e as vírgulas consequentes. Começando pelo tamanho, definiu-se a seguinte função:

```
def t_LSIZE(t):
    r'{[\ \t]*(?P<MIN>\d+)[\ \t]*(,[\ \t]*(?P<MAX>\d+)[\ \t]*)?}'
    if t.lexer.mode == 'HEADER':
        t.lexer.begin('LIST')
        match = t.lexer.lexmatch
        min_size = int(match.group('MIN'))
        max_size = match.group('MAX')
        field = t.lexer.fields[-1]
        if max_size == None:
            global list_counter
            list_counter = str(min_size)
            t.lexer.fields[-1] = (field, int(min_size))
        else:
            list_counter = str(max_size)
            t.lexer.fields[-1] = (field, (min_size, int(max_size)))
```

Definitivamente, esta foi a maior função criada ao longo deste projecto, pelo que se vai tentar explicá-la em pedaços mais pequenos. Em comparação com a primeira solução, é possível encontrar vários elementos em comum, tais como os grupos e a verificação da existência do

segundo grupo. É de notar que se poderia trocar qual o grupo opcional, não sendo obrigatório ser o da direita. Simplesmente no final, a verificação seria do tamanho mínimo e não do máximo.

Onde esta função difere é no começo do estado de lista e na forma como guarda o tamanho. Esta irá transformar o campo num tuplo, sendo que na segunda posição estará o tamanho. Caso seja uma lista de tamanho fixo, irá guardar o tamanho diretamente, ficando o tuplo com o seguinte formato: `(FIELD, SIZE)`. Caso seja um intervalo, irá criar um tuplo: `(FIELD, (MIN_SIZE, MAX_SIZE))`.

Como após a definição do tamanho se terá um número variável de vírgulas, teve-se de definir uma variável que guarda esse valor, denominada por `list_counter`. Inicialmente colocou-se essa variável no lexer, podendo ser acedida de forma facilitada dentro das funções, mas como a própria expressão regular precisa dela, teve-se obrigatoriamente de torná-la global.

```
@lex.TOKEN(r'(,[\ \t]*){' + list_counter + r'}')
def t_LIST_BLANK(t):
    global list_counter
    t.lexer.pos = t.lexer.lexpos + int(list_counter)
    list_counter = str(1)
    t.lexer.begin('INITIAL')
```

Devido ao `ply.lex` não aceitar variáveis dentro de expressões regulares, teve-se de importar o decorador `@TOKEN` da própria biblioteca. Este decorador irá transformar uma string numa docstring, permitindo a função de um *token* imediatamente a seguir capturar essa expressão.

A partir da função acima, também se repara numa nova variável, sendo esta `lexer.pos`. O propósito desta variável, será para se guardar a posição do lexer no fim das vírgulas, pois ao passar para o estado inicial, verificou-se que o lexer iria processar as vírgulas outra vez, definindo campos vazios no cabeçalho. Com isto, teve-se de alterar o *token* `BLANK` no estado inicial.

```
def t_BLANK(t):
    r',?[\ \t]*""|,[\ \t]*(?=(,))|,[\ \t]*\n'
    if t.lexer.lexpos > t.lexer.pos:
        if t.lexer.mode == "HEADER":
            t.lexer.fields.append
                ("EMPTY_" + str(t.lexer.count_empties))
            t.lexer.count_empties += 1
        else:
            t.lexer.fields.append("")
```

Tudo o que se acrescentou foi apenas uma nova restrição, em que se capturar algo antes da posição guardada anteriormente, não se irá processar. Para esta restrição utilizou-se uma variável já pré-definida do lexer chamada `lex.lexpos`, em que esta representa a posição atual do mesmo.

Por fim, definiu-se os caracteres que o ply irá ignorar no estado `LIST`, chegando-se à conclusão de que seriam os mesmos que o estado inicial.

```
t_LIST_INITIAL_ignore = ' \t\n'
```

Quanto aos registos, apenas se captura os campos tal como na fase anterior (secção 2.1), retornando uma simples lista com valores. Nas secções 3.2.2 e 3.2.3 irá-se referir como a partir desses campos se aplica as regras das listas.

2.3 Terceira Fase - Cabeçalho Com Funções

Sendo esta a fase mais trivial, começou-se por definir o formato geral de como uma lista com funções se irá apresentar num ficheiro CSV.

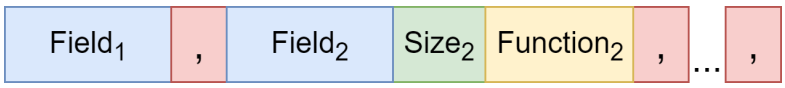

Figura 3: Estrutura básica de um registo.

Como é possível reparar, tudo o que se acrescenta é o módulo das funções imediatamente após a parte que especifica o tamanho das listas. A posição afortunada deste módulo, permite o aproveitamento do estado `LIST`, garantindo assim que num campo normal não se irá capturar o padrão associado a funções.

Começou-se por criar um novo *token*, alterando a lista de funções da seguinte forma:

```
tokens = ["FIELD", "QUOTES", "BLANK", "LSIZE", "FUNC"]
```

A este novo *token*, denominado por `FUNC`, apenas se associará ao estado `LIST`, pelas razões mencionadas acima, acabando por formar a seguinte função.

```
def t_LIST_FUNC(t):
    r'::(?P<OP>[a-zA-Z]+)'
    t.lexer.fields[-1] += (t.lexer.lexmatch.group("OP"),)
```

Tudo o que esta função faz, é capturar uma expressão regular que comece por um par de dois pontos, seguido por um conjunto de letras, sendo que podem ser tanto maiúsculas ou minúsculas. Esse conjunto de letras será a função que o programa irá analisar futuramente.

Após se obter a função, apenas se terá de adicionar no fim do tuplo do campo da lista, tomando o seguinte formato: `(FIELD, SIZE, FUNCTION)`.

As funções só terão algum efeito posteriormente, como se irá esclarecer na secção 3.2.4.

3 Implementação da Solução

Nesta secção, irá-se falar de como se efetivamente se utilizou todos os *tokens* e estados previamente mencionados.

3.1 Estrutura Geral do Programa

Em primeiro lugar, verifica-se se o utilizador forneceu argumentos suficientes (i.e. ficheiro de input e ficheiro de output), e, especificamente, se esses ficheiros têm extensão .csv e .json, respetivamente.

Depois, entra-se numa fase de pré-processamento, em que se abre os ficheiros do passo anterior, e se lê o *header* do ficheiro de input, extraindo informações relativas a nomes de campos, tamanhos em caso de listas, e funções se aplicável. Daqui resulta uma lista de elementos com os possíveis formatos:

1. `FIELD`
2. `(FIELD, SIZE)`
3. `(FIELD, (MIN_SIZE, MAX_SIZE))`
4. `(FIELD, SIZE, FUNCTION)`
5. `(FIELD, (MIN_SIZE, MAX_SIZE), FUNCTION)`

Finalmente, pode-se começar a ler o resto do ficheiro de input, 'fundindo' estes dados com as informações retiradas da fase anterior, resultando num lista de dicionários que poderá ser escrita no ficheiro de output no formato JSON.

3.2 Obtenção de Dicionários

Para cada linha do ficheiro de input, o programa itera sobre o *header*, que contém as informações mencionadas anteriormente:

```
i = 0
dic = {}
for category in header:
    ...
```

3.2.1 Campos Normais

No caso em que a categoria é uma simples string significa que há uma correspondência 1:1, ou seja, a essa chave vai corresponder um único campo.

```
if isinstance(category, str):
    dic[category] = fields[i]
    i += 1
```

3.2.2 Campos de Listas de Tamanho Fixo

Quando o segundo elemento da categoria é um inteiro N, está-se perante uma lista de tamanho fixo. A essa chave vão corresponder N campos. Caso a lista possua valores em branco, ou seja, se a lista não tiver os N elementos requeridos, a função retorna um dicionário vazio, que será interpretado como um entrada inválida e ignorada.

```
if not isinstance(category[1], tuple):
    min_size = category[1]
    cat_list = prepareList(fields[i:(i+min_size)])
    i += min_size
    size = len(cat_list)
    if size < min_size:
        return {}
    dic[category[0]] = cat_list
```

3.2.3 Campos de Listas de Tamanho Dinâmico

Se o segundo elemento da categoria é um tuplo (N,M), a lista terá entre N e M campos. O passo de verificação é muito semelhante ao das listas com tamanho fixo na medida em que, ao retirar os valores em branco, se o tamanho novo da lista for inferior ao tamanho mínimo estipulado, trata-se de uma entrada inválida.

```
if isinstance(category[1], tuple):
    (min_size, max_size) = category[1]
    cat_list = prepareList(fields[i:(i+max_size)])
    i += max_size
    size = len(cat_list)
    if size < min_size:
        return {}
    dic[category[0]] = cat_list
```

3.2.4 Campos de Listas com Funções

No caso em que a categoria tem 3 elementos (-,T,F), trata-se novamente de uma lista de tamanho T (fixo ou dinâmico), no entanto aplica-se a função F à lista. Executam-se os passos de verificação das listas, e, adicionalmente, verifica-se a validade da função F. Caso F não seja válida, ao invés de assumirmos uma entrada inválida, é utilizada a lista como se não tivesse sido aplicada F.

```
if category[2] == "sum":
    # SUM
    result = 0
    for value in cat_list:
        result += int(value)
    dic[category[0] + "_" + category[2]] = str(result)
```

```
elif category[2] == "avg":
    # AVERAGE
    result = 0
    for value in cat_list:
        result += int(value)
    dic[category[0] + "_" + category[2]] = str(result / size)

else:
    # Invalid function
    dic[category[0]] = cat_list
```

3.3 Conversão de Dicionários para um Ficheiro JSON

Através dos exemplos fornecidos no enunciado do projeto, facilmente retirou-se a estrutura requerida para o ficheiro **JSON**. Para tal, destacam-se cinco funções:

```
def dicToJson(fo, content):
    print("[", file=fo)
    for dic in content[:-1]:
        print("\t{", file=fo)
        printKeys(dic, fo)
        print("\t},", file=fo)
    else:
        print("\t{", file=fo)
        printKeys(content[-1], fo)
        print("\t}", file=fo)
    print("]", file=fo)
    return
```

Esta simples função imprime no ficheiro output **JSON** (`fo`) uma lista de dicionários, tendo o cuidado de que, na última iteração, não exista uma vírgula. O conteúdo recebido contém os dicionários previamente obtidos.

```
def printKeys(dic, fo):
    keys = list(dic.keys())
    for entry in keys[:-1]:
        printEntry(dic, entry, fo)
        print(",", file=fo)
    else:
        entry = keys[-1]
        printEntry(dic, entry, fo)
        print("", file=fo)
    return
```

Esta segunda função, utilizada na função previamente mencionada, itera sobre cada elemento do dicionário, imprimindo-o no ficheiro no formato adequado, tendo em conta, mais uma vez, que a última instância será tratada de modo diferente, ou seja, sem vírgula no final. É de

destacar que neste caso podem existir comportamentos diferentes nos diferentes elementos devido à possibilidade do aparecimento de uma lista, gerando-se aí um caso excecional, o qual é tratado adequadamente na função seguinte.

```
def printEntry(dic, entry, fo):
    value = dic[entry]
    if isinstance(value, list):
        printList(entry, value, fo)
    else:
        print ("\t\t\"" + str(entry) + "\": ", file=fo, end="")
        printElem(value, fo)
    return
```

Esta função tem como objetivo fundamental distinguir o tipo lista, imprimindo para ficheiro a chave e através da função `printElem` imprimir o valor da mesma.

```
def printList(entry, value, fo):
    print ("\t\t\"" + str(entry) + "\": [", file=fo, end ="")
    for elem in value[:-1]:
        printElem(elem, fo)
        print(", ", file=fo, end ="")
    else:
        printElem(value[-1], fo)
        print("]", file=fo, end="")
    return
```

Caso a entrada seja uma lista, existe já uma função especial para tratar deste caso, imprimindo-a para ficheiro com o formato requerido, mais uma vez usando a função `printElem` para imprimir o valor correspondente a cada chave de cada elemento da lista.

```
def printElem(elem, fo):
    try:
        print(int(elem), end = "", file=fo)
    except:
        try:
            print('\%.2f' \%float(elem), end = "", file=fo)
        except:
            print("\"" + str(elem) + "\"", end = "", file=fo)
    return
```

Finalmente, esta última função previamente mencionada tem como utilidade imprimir para ficheiro o valor recebido, tendo em atenção a conversão de tipo do elemento, tentando formatá-lo para o formato mais conveniente.

4 Testes

1. **CSV sem listas**

```
id_aluno,nome,curso,tpc1,tpc2,tpc3,tpc4
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",12,,18,20
"a3","Laurénio Narciso","ENGFIS",8,14,15,14,20
```

Figura 4: Ficheiro teste com cabeçalho sem listas


```
[
    {
        "id_aluno": "a1",
        "nome": "Aysha Melanie Gilberto",
        "curso": "LEI",
        "tpc1": 12,
        "tpc2": 8,
        "tpc3": 19,
        "tpc4": 8
    },
    {
        "id_aluno": "a2",
        "nome": "Igor André Cantanhede",
        "curso": "ENGFIS",
        "tpc1": 12,
        "tpc2": "",
        "tpc3": 18,
        "tpc4": 20
    },
    {
        "id_aluno": "a3",
        "nome": "Laurénio Narciso",
        "curso": "ENGFIS",
        "tpc1": 8,
        "tpc2": 14,
        "tpc3": 15,
        "tpc4": 14
    },
```


Figura 5: Resultado obtido com cabeçalho sem listas

2. **CSV com listas de tamanho fixo**

```
id_aluno,nome,curso,tpc{4},,,,
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",12,,18,20
"a3","Laurénio Narciso","ENGFIS",8,14,15,14,20
```

Figura 6: Ficheiro teste com listas de tamanho fixo no cabeçalho

```
[
    {
        "id_aluno": "a1",
        "nome": "Aysha Melanie Gilberto",
        "curso": "LEI",
        "tpc": [12, 8, 19, 8]
    },
    {
        "id_aluno": "a2",
        "nome": "Igor André Cantanhede",
        "curso": "ENGFIS",
        "tpc": [12, 18, 20]
    },
    {
        "id_aluno": "a3",
        "nome": "Laurénio Narciso",
        "curso": "ENGFIS",
        "tpc": [8, 14, 15, 14]
    },
```

Figura 7: Resultado obtido com listas de tamanho fixo no cabeçalho

3. **CSV com listas no meio do cabeçalho**

```
id_aluno_nome{2},curso,tpc{4},,,,
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",12,,18,20
"a3","Laurénio Narciso","ENGFIS",8,14,15,14,20
```

Figura 8: Ficheiro teste com duas listas de tamanho fixo no cabeçalho

```
[
    {
        "id_aluno_nome": ["a1", "Aysha Melanie Gilberto"],
        "curso": "LEI",
        "tpc": [12, 8, 19, 8]
    },
    {
        "id_aluno_nome": ["a2", "Igor André Cantanhede"],
        "curso": "ENGFIS",
        "tpc": [12, 18, 20]
    },
    {
        "id_aluno_nome": ["a3", "Laurénio Narciso"],
        "curso": "ENGFIS",
        "tpc": [8, 14, 15, 14]
    },
```

Figura 9: Resultado obtido com duas listas de tamanho fixo no cabeçalho

4. **CSV com listas de tamanho variável**

```
id_aluno,nome,curso,tpc{3,4}
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",,16,18,20
"a3","Laurénio Narciso","ENGFIS",,,15,14
"a4","Jasnoor Casegas","LCC",14,20,17,11,20
```

Figura 10: Ficheiro teste com listas de tamanho variável no cabeçalho


```
[
    {
        "id_aluno": "a1",
        "nome": "Aysha Melanie Gilberto",
        "curso": "LEI",
        "tpc": [12, 8, 19, 8]
    },
    {
        "id_aluno": "a2",
        "nome": "Igor André Cantanhede",
        "curso": "ENGFIS",
        "tpc": [16, 18, 20]
    },
    {
        "id_aluno": "a4",
        "nome": "Jasnoor Casegas",
        "curso": "LCC",
        "tpc": [14, 20, 17, 11]
    },
```


Figura 11: Resultado obtido com listas de tamanho variável no cabeçalho

5. **CSV com listas de tamanho variável no meio do cabeçalho**

```
id_aluno,nome_curso{1,2},tpc{3,4}
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",,16,18,20
"a3","Laurénio Narciso","ENGFIS",,,15,14
"a4","Jasnoor Casegas","LCC",14,20,17,11,20
```

Figura 12: Ficheiro teste com duas listas de tamanho variável no cabeçalho

```
[
    {
        "id_aluno": "a1",
        "nome_curso": ["Aysha Melanie Gilberto", "LEI"],
        "tpc": [12, 8, 19, 8]
    },
    {
        "id_aluno": "a2",
        "nome_curso": ["Igor André Cantanhede", "ENGFIS"],
        "tpc": [16, 18, 20]
    },
    {
        "id_aluno": "a4",
        "nome_curso": ["Jasnoor Casegas", "LCC"],
        "tpc": [14, 20, 17, 11]
    },
```

Figura 13: Resultado obtido com duas listas de tamanho variável no cabeçalho

6. **Listas com função de soma**

```
id_aluno,nome,curso,tpc{3,4}::sum
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",,16,18,20
"a3","Laurénio Narciso","ENGFIS",,,15,14
"a4","Jasnoor Casegas","LCC",14,20,17,11,20
```

Figura 14: Ficheiro teste com função soma na lista


```
[
    {
        "id_aluno": "a1",
        "nome": "Aysha Melanie Gilberto",
        "curso": "LEI",
        "tpc_sum": 47
    },
    {
        "id_aluno": "a2",
        "nome": "Igor André Cantanhede",
        "curso": "ENGFIS",
        "tpc_sum": 54
    },
    {
        "id_aluno": "a4",
        "nome": "Jasnoor Casegas",
        "curso": "LCC",
        "tpc_sum": 62
    },
```


Figura 15: Resultado obtido com função soma na lista

7. **Listas com função de média**

```
id_aluno,nome,curso,tpc{3,4}::avg
"a1","Aysha Melanie Gilberto","LEI",12,8,19,8
"a2","Igor André Cantanhede","ENGFIS",,16,18,20
"a3","Laurénio Narciso","ENGFIS",,,15,14
"a4","Jasnoor Casegas","LCC",14,20,17,11,20
```

Figura 16: Ficheiro teste com função média na lista


```
[
    {
        "id_aluno": "a1",
        "nome": "Aysha Melanie Gilberto",
        "curso": "LEI",
        "tpc_avg": 11.75
    },
    {
        "id_aluno": "a2",
        "nome": "Igor André Cantanhede",
        "curso": "ENGFIS",
        "tpc_avg": 18.00
    },
    {
        "id_aluno": "a4",
        "nome": "Jasnoor Casegas",
        "curso": "LCC",
        "tpc_avg": 15.50
    },
```


Figura 17: Resultado obtido com função média na lista

5 Conclusões e Trabalho Futuro

Chegando ao fim deste pequeno projeto, o grupo pode adquirir conhecimentos mais aprofundados relativamente a expressões regulares e duas bibliotecas da linguagem de programação python, sendo estas `re` e `ply.lex`.

Como extras, o grupo fez questão de pesquisar o que torna um campo válido, tentando aproximar-se dos interpretadores de CSV, como o excel. Isto permitiu o grupo poder explorar um pouco mais as expressões regulares, como a que se utiliza para a captura de campos, em vez de ser um simples '`\w+`'. Também se fez o projecto de forma modular, podendo-se acrescentar mais funcionalidades de forma mais facilitada, caso seja desejado.

Numa futura versão hipotética deste projeto propôs-se trabalhar com a biblioteca `ply.yacc`, já que, como foi demonstrado inúmeras vezes ao longo do trabalho, os campos têm uma estrutura bastante explícita. A essa estrutura pode-se dar o nome de gramática. Como essa matéria foi lecionada já numa fase bastante avançada do projeto, decidiu-se que não se iria aplicá-la.

Também existiu a hipótese de implementar mais funções, como máximo e mínimo de uma lista, mas o grupo decidiu que isso não iria acrescentar mais valor ao programa pois seria apenas "mais do mesmo". Como o foco do trabalho era expressões regulares e capturar diferentes grupos para futuro uso, o grupo achou que com o trabalho já feito pode-se demonstrar tudo o que era pretendido pelos docentes.

Por fim, pelas palavras de Fernando Pessoa, "A fé é o instinto da ação...". Por isso pode-nos dar um 20?

Referências

[1] CVS Reader Homepage, `https://www.csvreader.com/csv_format.php`

[2] PLY (Python Lex-Yacc), `https://ply.readthedocs.io/en/latest/ply.html`