Universidade do Minho
Escola de Engenharia

Universidade do Minho

Departamento de Informática

Processamento de Linguagens

Trabalho Prático nº 1

Processador de Registos de Exames Médicos Desportivos

Carlos Ferreira (A89509) José Peixoto (A89985)
Manuel Carvalho (A69856)

Março 2022

Resumo

O presente relatório descreve o trabalho prático realizado no âmbito da disciplina de *Processamento de Linguagens*, ao longo do segundo semestre do terceiro ano da Licenciatura em Engenharia Informática da Universidade do Minho.

O enunciado escolhido foi o "Processador de Registos de Exames Médicos Desportivos"

A realização deste trabalho prático tem como principal objetivo o processamento de um ficheiro CSV de forma a extrair todos os dados considerados relevantes, ficheiro esse que contém o registo de exames médicos desportivos. O processamento deste ficheiro é feito com recurso a Expressões Regulares identificando padrões no mesmo. A informação processada será posteriormente mostrada num website.

Neste documento descrevemos sucintamente o programa desenvolvido e discutimos as decisões tomadas durante a realização do trabalho prático.

Conteúdo

Estrutura do Relatório

O presente relatório encontra-se dividido em 3 partes.

No capítulo 1, Introdução, é feito um enquadramento e contextualização do trabalho prático e, de seguida, é feita uma descrição do problema.

No capítulo 2, Concepção da Solução, são expostas as estruturas de dados utilizadas e é feita uma descrição detalhada de todo o desenvolvimento do projeto até se obter a solução final.

Por fim, no capítulo 3, Conclusão, termina-se o relatório com uma síntese e análise crítica do trabalho desenvolvido.

1 Introdução

1.1 Enquadramento e Contexto

Tendo escolhido "Processador de Registos de Exames Médicos Desportivos"como tema do nosso projeto o trabalho apresentado tem como principal objetivo usar o módulo `re` da linguagem `Python` para reconhecer padrões num ficheiro input.

O ficheiro de input é um csv com registos de exames médicos desportivos de atletas. Cada um contém os campos, id, data, nome do atleta, idade, género, morada, modalidade, clube, email, estatuto de federado e resultado de aptidão.

A seguir apresenta-se um exemplo do conteúdo de um registo por campos:

```
id = 6045074cd77860ac9483d34e
index = 0
dataEMD = 2020-02-25
nome/primeiro = Delgado
nome/ultimo = Gay
idade = 28
genero = F
morada = Gloucester
modalidade = BTT
clube = ACRroriz
email = delgado.gay@acrroriz.biz
federado = true
resultado = true
```

1.2 Problema e Objetivo

Neste contexto de registos de exames médicos, pretende-se desenvolver um programa em `Python` capaz de:

- Calcular as datas extremas dos registos no dataset;
- Calcular a distribuição por género em cada ano e no total;
- Calcular a distribuição por modalidade em cada ano e no total;
- Calcular a distribuição por idade e género;
- Calcular a distribuição por morada;
- Calcular a distribuição por estatuto de federado em cada ano.
- Calcular a percentagem de aptos e não aptos por ano

Desta forma, numa primeira fase, o problema passa por analisar os registos de fornecidos. Em seguida, será necessário criar estruturas de dados adequadas para armazenar esta informação de forma a processá-la posteriormente.

Por fim, será criado um website para a apresentar toda a informação pedida de uma forma minimamente apresentável.

2 Conceção da Solução

2.1 Estratégia de Desenvolvimento

O grupo decidiu desenvolver o projeto em Python com tipagem e recorrendo ao mypy para verificação de tipos. Mypy é um *type checker* estático opcional para Python que aponta a combinar os benefícios de tipos de tipagem dinâmica e tipagem estática.

Optamos ainda por tentar seguir o paradigma funcional onde possível. Python é uma linguagem multi-paradigmática com capacidade para programação funcional, porém, não chega a ser pura neste sentido pelo que não seria realista seguir este paradigma muito estritamente.

Bibliotecas a ser usadas:

- matplotlib: Utilizada para gerar gráficos que permitem uma melhor visualização dos dados.
- jinja2: Utilizada para criar templates dos ficheiros html a ser criados com *placeholders* a preencher pelos dados calculados.

2.2 Expressões Regulares

As especificações dos campos não são dadas na descrição do projeto porém, o grupo optou por definir as suas especificações de forma a dar mais uso a regex no projeto. Chegamos então à seguinte tabela:

Field	Specification	Regex Pattern
_id	24 lowercase hex characters	[\da-z]{24}
index	Non negative integer	0|[1-9]\d*
dataEMD	ISO 8601 Complete and extended calendar date	\d{4}-\d{2}-\d{2}
nome/primeiro	Capitalized alphabetical word	[A-Z][a-z]*
nome/último	Capitalized alphabetical word	[A-Z][a-z]*
idade	Non negative integer	0|[1-9]\d{,2}
género	F or M	[FM]
morada	Capitalized alphabetical word	[A-Z][a-z]*
modalidade	Capitalized alphabetical word or uppercase acronym	[A-Z][A-Za-z]*
clube	Capitalized alphabetical word or uppercase acronym	[A-Z][A-Za-z]*
email	RFC 5322	*in the following code*
federado	true or false	(true|false)
resultado	true or false	(true|false)

2.3 Estruturas de Dados

De forma a armazenar a informação contida no dataset durante a execução do programa foi construída uma lista, `records`, com um registo por elemento e onde cada elemento é um dicionário que mapeia o nome de cada campo à informação correspondente do registo. Os seguintes tipos descrevem esta estrutura:

```
Record = Dict[str, str]
Records = List[Record]
```

Ao nível da implementação desta estrutura e da sua povoação foi criada a função `readCSV` responsável por ler o ficheiro CSV, e através de *named groups* e da função `groupdict` criar diretamente o dicionário acima referido que é anexado à lista `records`.

```
def readCSV(csv_file: str) -> Records:
    records: Records = []

    pattern = re.compile(r"""
        ^                                   # start of string
        (?P<id>         [\da-z]{24}),       # _id
        (?P<index>      0|[1-9]\d*),        # index
        (?P<date>       \d{4}-\d{2}-\d{2}), # dataEMD
        (?P<firstname>[A-Z][a-z]*),         # nome/primeiro
        (?P<lastname> [A-Z][a-z]*),         # nome/ultimo
        (?P<age>        0|[1-9]\d{,2}),     # idade
        (?P<gender>     [FM]),              # genero
        (?P<city>       [A-Z][a-z]*),       # morada
        (?P<sport>      [A-Z][A-Za-z]*),    # modalidade
        (?P<club>       [A-Z][A-Za-z]*),    # clube
        (?P<email>                          # email
            [a-z0-9!#%&'*+/=?^_`{|}~-]+             # local-part before fst dot
            (?:\.[a-z0-9!#%&'*+/=?^_`{|}~-]+)*      # local-part from fst dot
            @                                       # @ (at sign)
            (?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+  # domain name
            [a-z0-9](?:[a-z0-9-]*[a-z0-9])?),       # top-level domain name
        (?P<fed>        true|false),        # federado
        (?P<result>     true|false)         # resultado
        dollar                              # end of string
    """, re.X)

    with open(csv_file, 'r') as f:
        next(f)
        for line in f:
            if match := pattern.match(unidecode(line)):
                records.append(match.groupdict())

    return records
```

2.4 Algoritmos

Para responder às queries do enunciado foi utilizado um algoritmo de ordenação e agrupamento genérico que permitiu lidar com a maioria dos obstáculos encontrados. Um exemplo deste algoritmo está presente no seguinte excerto onde é evidente a utilização do paradigma funcional para obter soluções bastante expressivas:

```
def item_groups(records: Records, item: str) -> Dict[str, Records]:
    f = itemgetter(item)
    return {k: [*g] for k, g in groupby(sorted(records, key=f), key=f)}
```

Neste caso, a função devolve um dicionário de registos agrupados pelos campos selecionados na variável de entrada *item*. A partir da estrutura de retorno é simples calcular vários dados que possam ser necessários para responder à query, por exemplo:

- Quais os diferentes valores para um certo campo
- Quantos registos diferentes existem relativamente a um certo campo
- Registos ordenados e agrupados por ano

Para além deste tipo de algoritmo usamos também a seguinte função para calcular as data extremas:

```
def edge_dates(records: Records) -> Tuple[str, str]:
    date = itemgetter('date')
    return (min(records, key=date)['date'], max(records, key=date)['date'
    ])
```

Esta apenas usa as funções prédefinidas do python, *min* e *max*, para calcular as datas pretendidas.

Por fim usamos ainda outro algoritmo para calcular a distribuição dos registos por idade e género. Para este usámos ordenações e divisões sequenciais dos registos até obter estes estruturados de forma útil. Segue a função que implementa este algoritmo:

```
def age_gender(records: Records) -> Tuple[Tuple[Records, Records], Tuple[
    Records, Records]]:
    sorted_by_age = sorted(records, key=itemgetter('age'))
    split_idx = bisect_left(sorted_by_age, '35', key=itemgetter('age'))
    under35 = sorted_by_age[:split_idx]
    over35 = sorted_by_age[split_idx:]

    sorted_under35 = sorted(under35, key=itemgetter('gender'))
    split_idx = bisect_right(sorted_under35, 'Feminino', key=itemgetter('
    gender'))
    f_under35 = under35[:split_idx]
    m_under35 = under35[split_idx:]
```

```
11
12      sorted_over35 = sorted(over35, key=itemgetter('gender'))
13      split_idx = bisect_right(sorted_over35, 'Feminino', key=itemgetter('
        gender'))
14      f_over35 = over35[:split_idx]
15      m_over35 = over35[split_idx:]
16
17      return ((f_under35, m_under35), (f_over35, m_over35))
```

É de notar que aqui trocamos eficiência pelo que considerámos ser um solução mais natural do problema. Isto porque as divisões dos dados são sempre feitas com ordenações e divisões o que implica uma complexidade de $O(n \log n)$ onde se podia ter alcançado $O(n)$ através de iteração sobre as listas com atribuição condicional para outras listas.

3 Conclusão

Através do desenvolvimento deste projeto conseguimos aumentar a capacidade do grupo relativamente à escrita de expressões regulares como motor de reconhecimento de padrões textuais. Foi ainda interessante para a prática de desenvolvimento de software com a linguagem Python.

A utilização de Expressões Regulares para o reconhecimento de *strings* mostrou-se um mecanismo poderoso de *parsing*, permitindo simplificar esta tarefa.

Fazendo uma análise geral ao trabalho desenvolvido, considerámos ter atingido todos os objetivos de aprendizagem do mesmo e até implementar funcionalidades para lá das requeridas pelo enunciado.