Tema 4

Documentos

Objetivos:

- Documentos
- Comma Separated Values
- JavaScript Object Notation
- Extensible Markup Language

4.1 Introdução

A informação é frequentemente transferida entre máquinas, ou fornecida a aplicações para processamento, e estas atividades são vitais para o modo como os sistemas atuais funcionam e como nós utilizamos os recursos informáticos. Existe no entanto a necessidade das aplicações entenderem o formato utilizado para a codificação da informação. Para isso foram criados vários formatos, de acordo com as necessidades de cada caso de utilização.

Anteriormente foi abordado o formato HyperText Markup Language (HTML)[1] que permite a representação de documentos *Web*. Também foi abordado o formato LaTeX que permite a edição e geração de documentos principalmente de carácter técnico. Certamente também já utilizou ferramentas que operam sobre ficheiros `xls`, `docx`, ou `odt`.

Este tema irá focar-se sobre a manipulação de documentos em três formatos muito comuns, usados para a troca de informação entre aplicações e dispositivos (p.ex, através de *sockets*. Todos utilizam representações textuais para a codificação da informação, o que tem a vantagem de permitir que os dados sejam interpretados ou até gerados por humanos.

4.2 CSV

O formato Comma Separated Values (CSV)[2] é muito comum para a troca de informação, em especial quando se trata de séries temporais de valores medidos em sensores ou equipamentos laboratoriais, por exemplo. Teve a sua origem nos primórdios da computação, sendo um formato que exige muito pouca capacidade de processamento.

O seu nome deriva do uso de vírgulas para separar os diversos campos. Ele é estruturado em linhas de texto, terminadas por uma indicação de nova linha (`"\n"`), e que cada linha contém um ou mais valores relacionados entre si. Por outras palavras, cada linha representa um registo de dados composto por vários campos de informação. O formato pode possuir um cabeçalho, indicando qual o nome de cada coluna e frequentemente todas as linhas possuem um número igual de colunas.

O exemplo seguinte contém valores da temperatura medida dentro de um frigorífico num período de alguns minutos.

```
id,time,timestamp,value
1,15/03/2014 18:07:24,1394903244.0,2.3
1,15/03/2014 18:08:24,1394903304.0,1.8
1,15/03/2014 18:09:24,1394903364.0,1.2
1,15/03/2014 18:10:24,1394903424.0,1.6
1,15/03/2014 18:11:24,1394903484.0,2.1
1,15/03/2014 18:12:24,1394903544.0,2.5
1,15/03/2014 18:13:24,1394903604.0,2.9
1,15/03/2014 18:14:24,1394903664.0,3.3
1,15/03/2014 18:15:24,1394903724.0,3.0
1,15/03/2014 18:16:24,1394903784.0,2.8
1,15/03/2014 18:17:24,1394903844.0,2.4
```

Cada linha inclui um identificador do dispositivo que reportou os dados, uma data e hora em formato textual, um tempo em formato absoluto,[1] e finalmente o valor da temperatura. Este exemplo também demonstra que nem sempre a vírgula é um bom separador. Na língua Portuguesa utiliza-se a vírgula como separador decimal, pelo que qualquer número com parte decimal iria resultar numa linha com mais uma coluna.

Visto que o formato CSV não possui qualquer indicação de língua ou tabela de carateres, a escolha do separador deve ser feita com cuidado. Uma solução passa por delimitar com aspas todos os campos que potencialmente possam ter o caráter separador no seu interior. Utilizam-se também variações do formato CSV, tal com o formato Tabulation Separated Value (TSV). Neste último, o caráter de tabulação é utilizado para separar os diversos campos. É muito comum encontrarem-se ficheiros denominados CSV em que são utilizados os carateres `;`, `:` ou `|`.

[1] Este formato é muito comum e descreve o número de segundos desde 1 de Janeiro de 1970

Deste modo, deve-se considerar o formato CSV como uma família genérica de formatos e não apenas aqueles que utilizam vírgulas para separar valores.

Em *Python* e de uma forma geral na maioria das linguagens, o processamento deste tipo de ficheiros é simples e compatível com ferramentas existentes em qualquer sistema operativo. Por este motivo, o formato não morreu, encontrando ainda grande utilidade. Os ficheiros podem ser abertos usando o módulo **`csv`**, sendo que o módulo permite a iteração pelas linhas do ficheiro. Cada linha é apresentada como uma lista contendo um valor (da linha) em cada elemento da lista.

O exemplo seguinte abre um ficheiro CSV e imprime os seus valores.

```
import csv
import sys

def main(argv):
    fich_csv = open(argv[1], "r")
    csv_reader = csv.reader(fich_csv, delimiter=",")
    for row in csv_reader:
        print(row)
    fich_csv.close()

main(sys.argv)
```

Quando aplicado aos dados de temperatura o resultado deverá ser parecido com o seguinte.

```
["id", "time", "timestamp", "value"]
["1", "15/03/2014 18:07:24", "1394903244.0", "2.3"]
["1", "15/03/2014 18:08:24", "1394903304.0", "1.8"]
...
["1", "15/03/2014 18:17:24", "1394903844.0", "2.4"]
```

Desta forma, é possível aceder aos valores através de índices da lista, como por exemplo **`row[0]`**. Tenha em consideração que é possível especificar o delimitador a utilizar. De notar ainda que o cabeçalho do ficheiro é tratado como uma qualquer linha. Se por acaso pretender avançar sobre o cabeçalho do ficheiro e processar apenas as linhas de dados, ou seja neste exemplo imprimir somente as linhas de dados, pode usar a função **`next()`** na forma **`next(csv_reader)`** antes do ciclo de iteração. Experimente e veja a diferença.

No entanto, se o ficheiro for interpretado através do método **`csv.DictReader`**, o resultado será um dicionário que utiliza o cabeçalho para chave dos valores. Com este método o resultado será:

```
{"timestamp": "1394903244.0", "id": "1", "value": "2.3", "time": "15/03/2014 18:07:24"}
{"timestamp": "1394903304.0", "id": "1", "value": "1.8", "time": "15/03/2014 18:08:24"}
...
{"timestamp": "1394903844.0", "id": "1", "value": "2.4", "time": "15/03/2014 18:17:24"}
```

Exercício 4.1

Implemente um programa que leia os dados fornecidos e calcule o valor máximo, mínimo e médio da temperatura.

A criação de ficheiros CSV pode ser feita escrevendo os valores através da instrução **`print`**, mas o processo pode ser facilitado utilizando as estruturas e métodos adequados. Nomeadamente é possível criar uma lista com os valores e usar o módulo **`csv`** para a construção do ficheiro **`csv`**.

O exemplo seguinte cria um ficheiro chamado **`rand.csv`** com duas séries de valores: um valor incremental e um aleatório. O ficheiro também terá um cabeçalho com o nome das duas colunas.

```
import csv
import random

def main():
    fout = open("rand.csv", "w")
    writer = csv.DictWriter(fout, fieldnames=["time", "value"])

    writer.writeheader()

    for i in range(1,10):
        writer.writerow({"time": i, "value" : random.randint(1,100)} )

    fout.close()

main()
```

Exercício 4.2

Replique o exercício anterior e verifique o ficheiro criado. Modifique o delimitador especificando-o no construtor do objecto **`DictWriter`**.

A documentação desta classe pode ser consultada em `http://docs.python.org/3/library/csv.html#csv.DictWriter`.

Exercício 4.3

Considere o módulo **`time`** que permite acesso à hora atual e o módulo **`psutil`** que permite aceder a estatísticas do sistema,[a] em particular os seguintes métodos:

`time.time()`: Devolve o número de segundos desde o início da época (1/1/1970).

`psutil.cpu_percent(interval=x)`: Verifica qual a utilização do processador durante o intervalo especificado.

`psutil.net_io_counters()`: Verifica quantos pacotes/octetos foram enviados/recebidos por cada interface de rede. O resultado é um dicionário de tuplos. Por exemplo, o número de octetos enviados pela interface pode ser obtido acedendo à primeira posição, acedendo-se como a uma lista.

Construa um programa que registe o tempo em segundos, o número de octetos enviados e recebidos e a percentagem de ocupação do processador. O programa deve executar durante 60s, capturando valores a cada segundo. Execute o programa implementado, navegue por algumas páginas e verifique o resultado.

[a] Pode ser necessário instalar os módulos usando **`pip3 install nome-do-modulo`**.

4.3 JSON

Um outro formato bastante comum, especialmente no âmbito de aplicações *Web* é o formato JavaScript Object Notation (JSON)[3]. Sendo mais rico do que o formato CSV, mas mais compacto e menos rígido do que o formato XML (ver abaixo), é o formato utilizado em grande parte das transações de informação entre aplicações *Web*. A razão para isto reside no facto de ser um formato que é nativo para a linguagem *JavaScript* e muito bem suportado em muitas outras, como a linguagem *Python*.

O formato JSON é constituído pela descrição textual de pares chave-valor, sendo que cada valor pode ser uma *String*, um número, um *Array*, um valor booleano ou um outro objeto. Um exemplo de um documento JSON seria:

```
{
    "time" : 1394984189,
    "name" : "cpu",
    "value": 12
}
```

Este documento é composto por um objeto com três chaves, duas possuindo um valor inteiro e outra possuindo um valor *String*. Repare como a estrutura é semelhante à de um dicionário na linguagem *Python*.

De facto é possível converter qualquer estrutura de dicionário ou lista para o formato JSON e vice-versa, o que é extremamente útil para transmitir informação sobre *Sockets*.

O documento anterior pode ser gerado na linguagem *Python* através da criação e posterior conversão de um dicionário. O exemplo seguinte demonstra como uma lista de valores poderia ser convertida para o formato JSON.

```
import json

def main():
    data = [
               {"time": 1394984189, "name": "cpu", "value": 12},
               {"time": 1394984190, "name": "cpu", "value": 19}
           ]

    print(json.dumps(data, indent=4))

main()
```

Exercício 4.4

Verifique o resultado do exemplo anterior e altere-o de forma à variável **data** conter várias listas e dicionários dentro da lista principal.

Exercício 4.5

Altere o Exercício 3 de forma a que o seu resultado seja um ficheiro no formato JSON com o seguinte conteúdo:

```
{
    "stats" : [
                  {"time": timestamp, "cpu": value, "network": value},
              ]
}
```

A leitura de documentos do tipo JSON pode ser realizada através dos métodos: **json.loads**, que converte uma *String* em formato JSON e devolve uma lista ou um dicionário; e **json.load**, que lê um documento em formato JSON de um ficheiro.

Exercício 4.6

Implemente um programa que leia os dados armazenados no ficheiro criado pelo exercício anterior e os imprima no monitor.

4.4 XML

Outro formato bastante popular nos sistemas informáticos é o Extensible Markup Language (XML). Tal como o nome indica (ML = *Markup Language*), o formato XML é uma linguagem em si, o que significa que é bastante mais completo do que o formato CSV. O formato XML partilha muitas características com o HTML, mas tem em âmbito de aplicação muito mais variado e uma sintaxe mais rígida e uniforme.

Um documento XML é um texto que inclui *marcas* e *conteúdo*. As marcas seguem o padrão `<texto-da-marca>`, enquanto o conteúdo encontra-se entre marcas. As marcas usam-se para organizar o conteúdo do documento como um conjunto de *elementos* estruturais.

Cada elemento é indicado por uma *marca de início*, algum *conteúdo* e termina com uma *marca de fim* correspondente. O conteúdo de um elemento pode incluir outros elementos, formando uma estrututra hierárquica. As marcas de início e fim têm o formato `<tipo-de-marca>` e `</tipo-de-marca>`, repetivamente.

Também há elementos vazios, sem conteúdo, indicados por um marca com o formato `<tipo-vazio/>`. Além do tipo, o texto da marca também pode incluir atributos. O exemplo abaixo mostra um elemento XML de tipo **`foo`** cujo conteúdo inclui três elementos vazios de tipo **`bar`**. Todos os elementos têm atributos definidos.

```
<foo attrib1="x" attrib2="y">
    <bar attrib1="z1" />
    <bar attrib1="z2" />
    <bar attrib1="z3" />
</foo>
```

Repare que este formato é em tudo semelhante ao do formato HTML, mas no XML as marcas utilizadas não estão restringidas ao necessário para construir páginas. Pelo contrário, podem codificar uma grande variedade de conteúdos. Por exemplo, os formatos **`docx`** e **`odf`** utilizados pelas aplicações *Microsoft Office* e *LibreOffice* são baseados em XML. É um formato muito popular para codificar informação de documentos, para ficheiros de configuração e mesmo para transmitir séries de dados.

Um ficheiro contendo XML deve ser iniciado por um cabeçalho que indica algumas características do ficheiro nomeadamente: a codificação utilizada e por vezes o *Schema*. A codificação é importante para identificar corretamente os carateres utilizados, enquanto o *Schema* define que marcas podem ser encontradas no documento, o seu tipo e como se relacionam entre si. Um *Schema* é uma forma poderosa de definir a sintaxe específica de certo tipo de documento baseado em XML. É geralmente definido num documento separado, que também pode estar em formato XML.

Os documentos XML possuem uma estrutura hierárquica, o que significa que existe um elemento raíz e vários sub-elementos. Relembre o caso do HTML em que o elemento **<html>** é a raíz de qualquer documento deste tipo.

4.4.1 Leitura de Ficheiros XML

De uma forma ad-hoc, o formato XML é vulgarmente utilizado para armazenar configurações, ou para construir documentos que sejam interoperáveis entre várias plataformas, particularmente quando a informação a armazenar ou a enviar tem uma estrutura complexa.

Considere o exemplo seguinte, que poderia ser um ficheiro de configuração (que vamos designar por **conf.xml**) para o programa do Exercício 3. Em particular, este ficheiro define o intervalo de atualização, o formato de saída e quais os valores que o programa deve monitorizar.

```
<?xml version="1.0" encoding="utf-8"?>
<conf>
    <interval>1</interval>
    <output>
        <csv active="true" separator="," />
        <xml active="false" />
    </output>
    <monitor>
        <cpu active="true" />
        <network active="true" />
        <ram active="false" />
    </monitor>
</conf>
```

Para ler este ficheiro no programa podemos usar funções fornecidas no módulo **lxml.etree**, que permitem aceder ao conteúdo do ficheiro na forma de uma árvore. O código seguinte processa o ficheiro **conf.xml** e dá acesso aos seus elementos. Cada elemento da árvore possui um nome de marca (**tag**), attributos (**attrib**) e conteúdo (**text**).

```
from lxml import etree

def main():
    xml = etree.parse("conf.xml")
    root = xml.getroot()
    print(root.tag)
    for child in root:
        print(child.tag, child.attrib, child.text)

main()
```

O exemplo deverá imprimir a raiz do documento (**conf**) e todos os elementos contidos dentro da raíz, também chamados os elementos filhos.

Exercício 4.7

Altere o exemplo anterior de forma a imprimir todos os atributos e valores presentes no ficheiro **conf.xml**. Note que cada elemento filho poderá conter também outros filhos, que é preciso mostrar recursivamente.

Também é possível procurar entradas num ficheiro XML de forma a obterem-se rapidamente os valores pretendidos. Neste caso, isto seria interessante para que o programa pudesse obter os valores das configurações que irão condicionar a sua operação. Para isto utiliza-se o método **findall**, como exemplificado no excerto seguinte.

```
...
    monitor_cpu = root.findall("./monitor/cpu")
    monitor_ram = root.findall("./monitor/ram")

    print(monitor_cpu[0].attrib["active"])
    print(monitor_ram[0].attrib["active"])
```

O método **findall** devolve uma lista com todos os elementos encontrados. Se não for encontrado nenhum elemento, devolve uma lista vazia. De seguida é possível aceder ao atributo **active** de forma a saber se se deve monitorizar a ocupação de CPU e a utilização de RAM.

Exercício 4.8

Altere o programa criado no Exercício 3 de forma a que ele tenha em consideração o ficheiro **conf.xml**. Para obter a informação da memória, use o método **psutil.virtual_memory()**, que indica a memória disponível no segundo elemento do tuplo devolvido (ver **https://github.com/giampaolo/psutil**).

4.4.2 Escrita de valores

Os documentos XML, sendo baseados num formato textual, são facilmente editáveis por humanos. No entanto, estes documentos são também utilizados para a comunicação entre sistemas, pois a codificação textual também uniformiza o processamento do documento numa multiplicidade de sistemas, evitando complicações decorrentes de detalhes de codificações binárias como a *endianness* da representação.

Voltando ao exemplo anterior, gostaríamos que o programa pudesse escrever o seu resultado como um ficheiro XML em vez de CSV. Esta preferência poderá ser indicada através do ficheiro de configuração **conf.xml**. O processo de escrita de XML inclui 3 fases: 1) criação da raíz do documento e elementos principais, 2) adição de valores ao documento, 3) escrita do documento para um ficheiro de texto.

O exemplo seguinte cria uma raíz para utilizar num documento XML, cria depois um sub-elemento, adiciona-o à raíz e escreve o resultado para o ecrã.

```
from lxml import etree
import time

def main():
    root = etree.Element("stats")

    for i in range(1,10):
        value = etree.SubElement(root,"value")
        value.set("time", str(int(time.time())))
        value.text = str(i)
        time.sleep(1)

        print(etree.tostring(root, xml_declaration=True,
                             encoding="utf-8",pretty_print=True).decode("utf-8"))
main()
```

Neste caso a raíz terá um sub-elemento chamado **value** com um atributo chamado **time** que contém a hora atual. Depois de impresso, o resultado será o seguinte:

```
<?xml version="1.0" encoding="UTF-8"?>
<stats>
  <value time="1394984189"> 0 </value>
  ...
</stats>
```

Exercício 4.9

Implemente o exemplo anterior e verifique o resultado obtido.

Comparando com o formato CSV, pode-se verificar que o XML é muito menos compacto, ocupando muito mais espaço de armazenamento para conter a mesma informação. No entanto, a informação está mais estruturada e claramente identificada.

Se fosse necessário adicionar mais elementos ao elemento **value**, seguir-se-ia a mesma metodologia de criar um sub-elemento, tal como demonstrado no exemplo seguinte.

```
...
        value = etree.SubElement(root,"value")
        value.set("time", str(int(time.time())))

        cpu = etree.SubElement(value,"cpu")
        cpu.text = 10
...
```

Exercício 4.10

Considere o Exercício 3 e escreva o resultado para um ficheiro XML no formato indicado de seguida.

```
<stats>
    <value time="timestamp-atual">
        <cpu> valor </cpu>
        <ram> valor <ram>
        <network> valor </network>
    </value>
</stats>
```

4.4.3 Validação de documentos

Um aspeto importante no processamento de documentos, independentemente do formato utilizado, é a validação da sua estrutura e conteúdo. No caso de XML, um erro pode ocorrer devido a carateres inválidos ou em falta (ex, falta de um caráter `/`), mas também pode ser devido à colocação de marcas numa posição incorreta, porque alguns elementos só admitem conter elementos de certos tipos. Outra situação importante é a validação das marcas utilizadas no documento. Por exemplo, em HTML, a marca **`h1`** existe mas a marca **`hh1`** não faz qualquer sentido, sendo assim considerada como errada.

O exemplo seguinte considera uma pequena *Playlist* de música no formato XML Shareable Playlist Format (XSPF). Este formato, baseado em XML, é utilizado para armazenar e partilhar *Playlists* com músicas.

```
<?xml version="1.0" encoding="utf-8"?>
<playlist version="1" xmlns="http://xspf.org/ns/0/">

  <title>My playlist</title>
  <creator>Jane Doe</creator>
  <annotation>My favorite songs</annotation>
  <info>http://example.com/myplaylists</info>

  <trackList>
    <track>
      <location>http://example.com/my.mp3</location>
      <title>My Way</title>
      <creator>Frank Sinatra</creator>
      <annotation>This is my theme song.</annotation>
      <album>Frank Sinatra's Greatest Hits</album>
      <trackNum>3</trackNum>
      <duration>19200</duration>
    </track>
  </trackList>
</playlist>
```

Neste exemplo, a primeira linha identifica o ficheiro como sendo XML usando uma codificação **`UTF-8`**. A segunda linha identifica qual o elemento raíz do documento. Tendo em consideração um dado documento e o seu *Schema* é possível validar conteúdo e estrutura, detectando erros no documento. Pode reparar que com a excepção da marca **`title`**, nenhuma outra está presente num documento HTML, sendo que a estrutura é bastante semelhante.

O exemplo seguinte demonstra como é possível validar a *Playlist* anterior. O ficheiro **`paylist.xspf`** contém o exemplo demonstrado anteriormente, enquanto que o ficheiro **`xspf-1_0.7.rng`** pode ser encontrado no endereço **`https://svn.xiph.org/websites/xspf.org/validation/`** e contém o *Schema* relativo ao formato XSPF.

```
from lxml import etree

def main():
    doc = etree.parse("playlist.xspf")

    schema = etree.parse("xspf-1_0.7.rng")
    validator = etree.RelaxNG(schema)

    print(validator.validate(doc))
    print(validator.error_log.last_error)

main()
```

Neste exemplo, a primeira impressão irá apresentar o resultado da validação, enquanto a segunda impressão irá apresentar o erro encontrado (se algum). Ao acto de se ler um formato estruturado dá-se o nome de *Parsing* e como pode deduzir, este método de validação permite que aplicação (ex, o *LibreOffice*, ou um navegador) verifique se um dado documento foi construído corretamente ou possui erros.

Exercício 4.11

Replique o exemplo anterior e verifique se o documento apresentado é XML válido para uma *Playlist*.

Introduza uma qualquer pequena modificação ao ficheiro e volte a validar o documento.

Exercício 4.12

Altere o programa anterior de forma a escrever o resultado em XML.

Glossário

CPU Central Processing Unit

CSV Comma Separated Values

HTML HyperText Markup Language

JSON JavaScript Object Notation

RAM Random Access Memory

TSV Tabulation Separated Value

XML Extensible Markup Language

XSPF XML Shareable Playlist Format

Referências

[1] W3C. «HTML 4.01 Specification». (1999), URL: `http://www.w3.org/TR/1999/REC-html401-19991224/`.

[2] Y. Shafranovich, *Common Format and MIME Type for Comma-Separated Values (CSV) Files*, RFC 4180 (Informational), Internet Engineering Task Force, out. de 2005.

[3] E. T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, Internet Engineering Task Force, mar. de 2014.