Laboratórios de Informática 2014/2015

DETI - Universidade de Aveiro

Docentes
João Paulo Barraca <jpbarraca@ua.pt>
Diogo Gomes <dgomes@ua.pt>
João Manuel Rodrigues <jmr@ua.pt>
Mário Antunes <mario.antunes@ua.pt>

TEMA 16

Documentos

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

16.1 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 que as aplicações entendam o formato utilizado para a codificação da informação, sendo que foram vários os formatos criados, de acordo com as necessidades de cada caso de utilização. Num tema anterior 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 de carácter principalmente 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, constituindo pilares para a troca de informação entre aplicações e dispositivos.

Todos utilizam representações textuais para a codificação da informação, possuindo um valor muito grande quando é pretendido que os dados sejam passíveis de ser gerados ou interpretados por humanos.

16.2 CSV

O formato Comma Separated Values (CSV)[2] é muito comum para a troca de informação, em especial quando considerando séries temporais e equipamentos laboratoriais. 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"`), em que cada linha contém um ou mais valores partilhando alguma relação. Se se considerar a aquisição de valores efetuada por um equipamento de laboratório, cada linha irá conter o valor e outros campos presentes na altura da aquisição, tal como a hora atual. O formato pode possuir um cabeçalho, indicando qual o nome de cada coluna e frequentemente todas as linhas possuem um número igual de valores (colunas).

Como exemplo, podem-se considerar as linhas seguintes, relativas à temperatura encontrada dentro de um frigorífico.

```
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
```

Como se pode verificar, cada linha indica um identificador (ex, o número do dispositivo que reportou os dados), uma data e hora em formato textual, um tempo em formato absoluto[1], e finalmente o valor. Este exemplo também demonstra que nem sempre a vírgula é um bom separador. Na língua Portuguesa utiliza-se a vírgula para separar os valores inteiros dos decimais, pelo que qualquer número com componente 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

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

com cuidado. Uma solução passa por delimitar todos os campos que potencialmente possam ter o caráter separador no seu interior por aspas. Utilizam-se também variações do formato CSV, tal com o formato Tabulation Separated Value (TSV). Neste último formato 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 `|`. 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 formado 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], "rb")
    csv_reader = csv.reader(fich_csv, delimiter=',')
    for row in csv_reader:
        print row

main(sys.argv)
```

Quando aplicado aos dados de temperatura o resultado deverá ser composto por linhas no formato:

```
['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. 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 16.1

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

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. De notar que o programa cria uma lista chamada **data** e depois essa lista é escrita para um ficheiro. O ficheiro também terá um cabeçalho com o nome das duas colunas.

```
import csv
import random

def main():
    fout = open('rand.csv', 'wb')
    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 16.2

Replique o exercício anterior e verifique o ficheiro criado. Modifique o delimitador especificando-o a quando da construção do objecto `DictWriter`. A documentação desta classe pode ser consultada em `http://docs.python.org/2/library/csv.html#csv.DictWriter`.

Exercício 16.3

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 Janeiro 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. No caso do interface `eth0`, o número de octetos enviados pode ser verificado por interpretando o resultado da forma `res['eth0'][0]`.

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 1 segundo. Execute o programa implementado, navegue por algumas páginas e verifique o resultado.

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

16.3 JSON

Um outro formato bastante comum, especialmente no âmbito de aplicações *Web* é o formato JavaScript Object Notation (JSON)[3]. Mais rico do que o formato CSV mas sendo mais compacto e menos rígido do que o formato Extensible Markup Language (XML)**xmlstd** , 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, de onde se inclui a linguagem *Python*.

Estruturalmente o formato JSON é construído através da descrição textual de pares chave-valor, sendo que o objecto 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 uma 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 e de o formato textual JSON, 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 16.4

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

A leitura de documentos do tipo JSON pode ser realizada através do método **`json.loads`**, sendo que será devolvida uma lista ou dicionário com o conteúdo da } indicada no parâmetro da função.

Exercício 16.5

Altere o Exercício 3 de forma a que o seu resultado seja um ficheiro no formato JSON, seguindo o formato:

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

}
```

16.4 XML

Finalmente um formato bastante popular nos sistemas informáticos é o XML. Tal como o nome indica, o formato XML é uma linguagem em si, o que significa que é bastante mais completo do que o formato CSV. Pode considerar que o formato HTML é um caso especial do formato XML, partilhando no entanto muitas características.

Este formato é baseado em texto organizado por marcas que podem conter atributos. Uma marca possui um início e um fim, podendo igualmente conter algum conteúdo. O conteúdo pode ser uma outra marca. Um exemplo de uma marca contendo conteúdo e atributos pode ser encontrado de seguida:

```
<marca attrib1='x' attrib2='y'>
    <outra-marca attrib1='z1' />
    <outra-marca attrib1='z2' />
    <outra-marca attrib1='z3' />
</marca>
```

Repare que este formato é em tudo semelhante ao já estudado a quando do formato HTML. No caso do formato XML considera-se que as marcas utilizadas não estão restritas ao necessário para construir páginas, podendo ser utilizado para a codificação de uma grande variedade de conteúdos. Não está portanto limitado à representação de séries de valores. Como exemplo, os formatos **`docx`** ou **`odf`** utilizados pelas plataformas *Microsoft Office* e *LibreOffice* são construídos utilizando XML. No âmbito da informática este formato é muito popular para codificar informação de documentos ou 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 útil para identificar corretamente os carateres utilizados, enquanto o *Schema* define que marcas pode ser encontradas e como estas se relacionam. 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 a marca **<html>** é a raíz de qualquer documento deste tipo.

16.4.1 Leitura de Ficheiros XML

De uma forma ad-hoc, o formato XML é vulgarmente utilizado para armazenar configurações, ou construir documentos que sejam interoperaveis entre várias plataformas. Em particular quando a informação a armazenar ou enviar é estruturada.

Considere o exemplo seguinte, que descreve um possível ficheiro de configuração para definir como o programa do Exercício 3 poderia funcionar. Em particular define o intervalo de atualização, o formato de saída e quais os valores a 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>
```

A incorporação desde ficheiro no programa requer apenas a importação do módulo correto (**lxml**), sendo que a árvore que representa o documento está disponível para processamento. O código seguinte processa um ficheiro **conf.xml** e permite acesso aos seus membros. Considere que cada elemento da árvore possui um nome (**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 as marcas contidas dentro da raíz, também chamadas de marcas filhas.

Exercício 16.6

Altere o exemplo anterior de forma a imprimir todos os atributos e valores presentes no ficheiro **conf.xml**. Considere que cada marca filha terá também outras marcas filhas que é preciso iterar.

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** tal 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** irá devolver 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 16.7

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, considere o método **psutil.virtual_memory()**, sendo que a memória disponível corresponde ao segundo índice do tuplo devolvido (ver **https://code.google.com/p/psutil/**).

16.4.2 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 sua codificação textual também facilita que um dado documento seja facilmente processado numa multiplicidade de sistemas.

No caso do exemplo anterior, seria útil que o programa pudesse escrever o seu resultado como um ficheiro XML em vez de CSV. Sendo que esta preferência pode ser definida através do ficheiro de configuração **`conf.xml`**. Para isto tem de se considerar que existem 3 fases do processo: 1) criação a 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))

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 16.8

Implemente o exemplo anterior e verifique o resultado obtido.

Relativamente ao formato CSV, pode-se verificar que este formato é muito menos compacto, ocupando muito mais espaço de armazenamento para conter a mesma informação. No entanto a informação considera-se estar mais estruturada e claramente identificada.

Se fosse necessário adicionar mais elementos ao elemento **value**, seguiria-se a mesma metodologia de criar um sub-elemento, tal como demonstrada no exemplo seguinte:

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

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

Exercício 16.9

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>
```

16.4.3 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 caratere `/`), mas

também pode ser devido à colocação de marcas numa posição incorreta. A título de exemplo, a pode considerar um documento HTML onde dentro da marca **<head>** apenas são permitidas algumas outras marcas. Outra situação importante é a validação das marcas utilizadas no documento. Mais uma vez, considerando HTML, a marca **h1** existe mas a marca **hh1** não faz qualquer sentido, sendo assim considerada como errada se for encontrada num documento HTML.

O exemplo seguinte considera uma pequena *Playlist* de música no formato XML Shareable Playlist Format (XSPF). Este formato 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 o ficheiro **xspf-draft8.rng** pode ser encontrado no endereço **http://www.xspf.org/schema/** e contém o *Schema* dos dados.

```
from lxml import etree

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

    schema = etree.parse('xspf-draft8.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, *LibreOffice*, ou um navegador) verifique se um dado documento foi construído corretamente ou possui erros.

Exercício 16.10

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 16.11

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

16.5 Para Aprofundar

Exercício 16.12

Considere o ficheiro disponível em `http://api.openweathermap.org/data/2.5/weather?q=NOME-DA-CIDADE&mode=xml`, contendo os dados meteorológicos observados na cidade indicada. Obtenha-o para o seu computador e construa um programa que aceite um argumento como nome da cidade e imprime os dados observados.

Exercício 16.13

Considere o ficheiro disponível em `http://api.web.ua.pt/pt/services/universidade_de_aveiro/lotacao_parques`, contendo os dados relativos à capacidade dos parques da Universidade de Aveiro.
Implementa um programa que apresente a disponibilidade do parque mais próximo do utilizador. Considere que a localização é fornecida como argumento, no formato latitude e longitude. Para calcular a distância entre duas coordenadas, recorra à fórmula da distância de Haversine.

Exercício 16.14

Considere um outro serviço disponível no endereço `http://api.web.ua.pt` e apresente os dados fornecidos. Neste site pode encontrar serviços que fornecem informação tal como as ementas ou as senhas da secretaria.

Pode deixar um programa a monitorizar com frequência um dado serviço. Por exemplo, será que a ementa depende do estado do tempo? E a hora da manhã a partir da qual os parques enchem?

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. (1999). Html 4.01 specification, endereço: `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.