Departamento de Informática, Faculdade de Ciências da Universidade de Lisboa

Sistemas Operativos
2021/22

Guião de aula
Ficheiros

1. Introdução

Nas últimas aulas a forma como comunicámos com os programas que desenvolvemos foi através do teclado, o *standard input* (`stdin`) e ecrã, o *standard output* (`stdout`). O `stdin` e o `stdout`[1] são assim abstrações pré-definidas pelo Sistema Operativo que servem como ligação de entrada e saída (I/O) de dados entre o programa e o ambiente (por exemplo, o utilizador). Nesta aula vamos falar de outra forma de comunicação dos programas com o ambiente envolvente: os ficheiros.

1.1. Abrir e fechar ficheiro

Antes de se poder ler de, ou escrever para um ficheiro é necessário que este seja previamente aberto. Para tal, utiliza-se a função `open()`, a qual é normalmente usada com dois argumentos:

```
f = open(filename, mode)
```

Esta função recebe o nome do ficheiro que vai ser aberto (*`filename`*) e informação que define o modo de abertura do ficheiro (*`mode`*). A função retorna um objeto do tipo `file`, ou seja, o seu descritor.

Há vários modos de abertura de um ficheiro:

- "r" – abre para leitura
- "w" – abre para escrita (se o ficheiro existir perde-se o conteúdo anterior)
- "a" – abre para acrescentar (escreve no fim do ficheiro)
- "r+" – abre para leitura e escrita

O argumento *`mode`* é opcional. O seu valor por omissão é "r".

Caso não seja possível abrir o ficheiro, é lançada a exceção **IOError**.

```
try:
      f = open(filename, mode)

except IOError as e:
      print >> sys.stderr, "open failed ",e.errno, "-", e.strerror
      sys.exit(1)
```

1 Aos quais deveríamos acrescentar o `stderr`, que é também usado para o programa passar informação para a consola, tal como o `stdout`, mas neste caso é usado para passar mensagens de erro ou de diagnóstico.

Quando já não for preciso usar um ficheiro `f`, deve chamar-se a função `f.close()`, de forma a fechar o ficheiro e libertar todos os recursos do sistema necessários para manter um ficheiro aberto. Como os Sistemas Operativos têm normalmente um limite para o número de ficheiros que podem estar abertos simultaneamente, é sempre aconselhável que se fechem os ficheiros que já não são necessários.

Precisamente para garantir que os ficheiros abertos são fechados quando já não são necessários, é boa prática usar a expressão `with` do python sempre que se utilizam objetos do tipo `file`. Isto pode ser feito da seguinte forma:

```
with open(filename, mode) as f:
        # Código que vai utilizar o ficheiro. Por exemplo:
        dados = f.read()
```

Após a execução do código no âmbito da expressão `with` o ficheiro já estará fechado, não havendo a necessidade de executar a instrução `f.close()` explicitamente. É de notar que a necessidade tratamento de exceções se mantém.

1.2. Leitura e escrita de ficheiros

Nesta e na próxima secção vamos assumir que foi criado um objeto do tipo ficheiro com o nome `f`.

Para ler o conteúdo do ficheiro, chama-se a função `f.read(size)`, que lê `size` bytes de um ficheiro e retorna esses dados como uma *string*. O argumento `size` é opcional. Por omissão, retorna todo o conteúdo do ficheiro. Quando se atinge o fim do ficheiro, a chamada `f.read()` retorna uma *string* vazia ("").
A chamada `f.readline()` lê apenas uma linha do ficheiro. O caracter `newline (\n)` é deixado no final da *string* e só é omitido na última linha do ficheiro (caso o ficheiro não termine numa `newline`). Assim, se `f.readline()` retornar uma *string* vazia, foi atingido o fim do ficheiro. Pelo contrário, uma linha vazia é representada por um `'\n'`.

Para ler várias linhas de um ficheiro, a forma mais eficiente, rápida e simples é fazer um ciclo sobre o objeto ficheiro:

```
for line in f:
     print line,
```

A chamada `f.write(string)` escreve os conteúdos de uma *string* para o ficheiro, retornando `None`. Num ficheiro de texto, se for necessário escrever algo diferente de uma *string* tem de se fazer a conversão para *string* (usando a função `str()`).

1.3. Outras funções úteis

Nesta secção apresentamos algumas funções úteis para manipulação de ficheiros e tratamento dos dados neles armazenados.

A chamada `f.tell()` retorna um inteiro que nos diz qual a posição atual do objeto `f` no ficheiro. Este valor é medido em bytes a partir do início do ficheiro. Se quisermos alterar a posição do objeto ficheiro, podemos usar a função `f.seek(offset, from_what)`. A nova posição é calculada adicionando o `offset` ao ponto de referência `from_what`. Se o valor do `from_what` for 0 a medida é feita deste o início do ficheiro, se for 1 usa-se a posição atual, se for 2 usa-se o fim do ficheiro como referência. O valor por omissão do argumento `from_what` é 0.

Uma função útil para fazer *parsing* de ficheiros é a função `split()`. Depois de o conteúdo de interesse de um ficheiro estar numa *string* `str`, a chamada `str.split(sep)` retorna uma lista de palavras da *string*, usando `sep` como delimitador para dividir a *string* original. Por omissão, o espaço em branco é o delimitador.

1.4. Exemplo de uso de ficheiros de texto

O objetivo do exemplo que vai ser apresentado nesta secção é ler alguns valores numéricos de um ficheiro de texto, `in.txt`, calcular a sua soma, e escrever o resultado num novo ficheiro de texto, `out.txt`. O conteúdo do ficheiro `in.txt` deste exemplo é o seguinte:

```
[Dados Soma]
Valores=10 12 20
```

O código que permite resolver este problema é apresentado de seguida.

```
with open("in.txt", "r") as inFile, open("out.txt","w") as outFile:
      for line in inFile:
            if line[0] != "[":
                  dados = line.split("=")
                  numeros = dados[1].split()

      soma = 0
      for numero in numeros:
            soma += int(numero)

      outFile.write(str(soma))
```

A vermelho destacamos as funções de abertura e fecho dos ficheiros, **a verde** as operações de leitura e escrita, e finalmente **a azul** as funções auxiliares para tratamento dos dados. Uma nota importante é relativa ao uso da função `split()`. A primeira chamada retorna uma lista com dois elementos, "Valores" e "`10 12 20`", já que o separador é o "=". O `split()` seguinte pega apenas no segundo elemento da lista e divide-o tendo por base o separador por omissão, o espaço em branco.

O resultado da execução deste programa é a criação de um novo ficheiro `out.txt` com o seguinte conteúdo:

```
42
```

1.5. Escrita de dados estruturados (serialização)

Como vimos, é muito fácil escrever e ler *strings* de um ficheiro. No exemplo anterior vimos que é um pouco mais complicado lidar com números: como o método `read()` só retorna *strings*, temos de usar a função `int()` para retornar o valor inteiro. Para gravar tipos de dados ainda mais complexos, como listas ou objetos, o *parsing* e a serialização (isto é, a forma como os dados são gravados) são ainda mais complexos.

O módulo `pickle` permite a serialização e desserialização de dados estruturados e objetos Python. Ao usar o *pickle*, esses dados ou objetos são convertidos numa *stream* de bytes que pode ser gravada num ficheiro (operação de "*pickling*"). A operação `pickle.dump(obj,f)` permite escrever uma representação *pickle* do objeto `obj` no ficheiro `f`.

Depois da leitura dessa *stream* de bytes do ficheiro, pode ser feita a operação inversa, o "*unpickling*", convertendo a *stream* gravada nos objetos Python originais. A operação:

```
obj = pickle.load(f)
```

lê a *string* com o conteúdo do ficheiro `f` e reconstrói o objeto original `obj`.

Os programas seguintes ilustram o uso deste módulo para gravar dados estruturados em ficheiros de texto.

Exemplo de *pickling*:

```
import pickle

class Pessoa(object):
      def __init__(self, nome):
            self.nome = nome

      def getNome(self):
            return self.nome

num = 11
tuplo = (2018, 11, 20, 21.5, "SO TP")
objecto = Pessoa("Aluno")

x = [num, tuplo, objecto]

with open("serial.txt","wb") as outFile:
      pickle.dump(x, outFile)
```

Ao executar este código, o conteúdo do ficheiro `serial.txt` vai conter, em formato *pickle* (serializado), os dados complexos que estão na lista `x`, incluindo um número, um tuplo, e ainda um objeto.

Exemplo de *unpickling*:

```
import pickle

class Pessoa(object):
      def __init__(self, nome):
            self.nome = nome

      def getNome(self):
            return self.nome

with open("serial.txt","rb") as inFile:
      x = pickle.load(inFile)

      print ("num = ", str(x[0]))
      print ("tuplo = ", str(x[1]))
      print ("nome = ", x[2].getNome())
```

Este código permite reconstruir os dados originais (ou seja, deserilizar), sendo o resultado da sua execução o seguinte:

```
num = 11
tuplo = (2018, 11, 20, 21.5, 'SO TP')
nome = Aluno
```

1.7. Ficheiros binários

Até agora discutimos a criação, leitura e escrita de ficheiros *de texto*. Nesta secção vamos falar de ficheiros binários. Ao contrário dos ficheiros de texto, estes ficheiros armazenam qualquer tipo de dados (e não somente texto, como os primeiros), o que por vezes é útil por razões de eficiência.

Quando se quer abrir um ficheiro binário deve incluir-se a letra "b" na definição do modo: por exemplo, "rb" no caso dum ficheiro binário para leitura.[2]

O exemplo seguinte ilustra a criação e manipulação de um ficheiro binário, `outbin`. No primeiro programa, o utilizador cria um ficheiro binário onde armazena uma lista de variáveis do tipo `int`.

Escrita binária

```
import struct

numeros = [0, 2, 4, 8, 16, 32]

with open("outbin","wb") as outFile:
     for num in numeros:
         outFile.write(struct.pack("i",num))
```

O uso da função `pack()` do módulo `struct` é necessário pois permite obter uma representação de cada um dos números no formato pretendido. Neste exemplo pretendemos guardar valores inteiros (com o tamanho normalizado de 32 bits), daí termos colocado a opção "i" (`integer`). Se quiséssemos guardar números reais podíamos usar a opção "f" (`float`).

O programa que permite a leitura do ficheiro binário é apresentado a seguir.

[2] Na realidade em sistemas Unix não é necessário acrescentar o "b" para distinguir um ficheiro binário de um ficheiro de texto, mas por razões de portabilidade tal é aconselhável.

Leitura binária

```
import struct

numerosLidos = []

with open("outbin", "rb") as inFile:
      for i in range(6):
            bytes = inFile.read(4)
            tuplo = struct.unpack("i",bytes)
            numerosLidos.append(tuplo[0])

print (numerosLidos)
```

Como cada número ocupa 4 bytes (32 bits), temos de ler esta quantidade de bytes para cada elemento da lista. Depois, executa-se a operação `unpack()` para fazer a conversão dos bytes recebidos para o formato apropriado (o retorno desta função é um tuplo). O resultado da execução deste código é o seguinte:

```
[0, 2, 4, 8, 16, 32]
```

1.8. Hex dump

Para visualizar um ficheiro de texto basta simplesmente usar-se um editor de texto, como o `gedit`. Visualizar o conteúdo de um ficheiro binário é mais complicado, pois a representação binária dos diferentes dados é diferente da representação de texto. Há, no entanto, algumas ferramentas que nos permitem visualizar os conteúdos de um ficheiro binário, como o utilitário `hexdump`.

Um *hex dump* permite uma vista hexadecimal dos dados armazenados num ficheiro binário. Cada byte é representado por dois dígitos hexadecimais. Recorde-se que há várias formas de representar um número. Por exemplo, o número 10 (em formato decimal) é:

1) em formato binário, 00001010_2
2) e em formato hexadecimal, $0A_{16}$

Um *hex dump* é organizado em linhas com 8 ou 16 bytes separados por espaços. Cada linha contém o endereço de memória do lado esquerdo e os dados guardados no centro. Para visualizarmos o ficheiro `outbin` criado na secção anterior podemos executar o seguinte comando:

```
$ hexdump outbin
```

O resultado da execução deste comando é o seguinte:

```
0000000 0000 0000 0002 0000 0004 0000 0008 0000
0000010 0010 0000 0020 0000
0000018
```

Como se pode verificar, no ficheiro aparece primeiro o **número 0**, depois **o número 2**,[3] depois o **número 4**, depois o **número 8**, e assim sucessivamente.

[3] Em formato *little endian*, ou seja, o byte menos significativo é armazenado primeiro, no endereço mais baixo.

2. Exercícios fundamentais

1. Escreva um programa em Python que escreva para dois ficheiros – um ficheiro binário e um ficheiro de texto – o número 1.5. Compare e discuta o tamanho dos dois ficheiros

2. Faça o mesmo exercício para o número 1.5555555555. Compare e discuta o tamanho dos dois ficheiros. Discuta também a precisão do número armazenado em cada um dos ficheiros.

3. Faça o mesmo exercício para o número n=1/3. Compare e discuta o tamanho dos dois ficheiros. Discuta também a precisão do número armazenado em cada um dos ficheiros. Sugestão: pode ler o número de ambos os ficheiros e compará-lo com x=1/3.

---

4. Considere o ficheiro `info.txt` disponível na página da disciplina. Este ficheiro contém informação relativa à idade e ao peso de um conjunto de pessoas. A informação encontra-se dividida em dois blocos, "Mulheres" e "Homens", e cada linha contém a seguinte informação:

```
Nome=Idade,Peso
```

4.1. Escreva um programa que leia os dados do ficheiro e calcule:

a) a média das idades
b) a média das idades das mulheres e dos homens (em separado)
c) a média dos pesos
d) a média dos pesos das mulheres e dos homens (em separado)

Os resultados devem ser arredondados para o inteiro mais próximo.

4.2. Escreva os resultados do programa num novo ficheiro `medias.txt` que tenha o seguinte formato:

```
-- Media de idades --
Global: <…>
Mulheres: <…>
Homens: <…>

-- Media de pesos --
Global: <…>
Mulheres: <…>
Homens: <…>
```

4.3. Armazene a informação existente no ficheiro `medias.txt` num novo ficheiro `medias2.txt`, mas com um formato diferente. Desta vez, a informação deve ser armazenada numa lista contendo dois tuplos, um para armazenar a média das idades e outro para armazenar a média dos pesos. Cada tuplo deve conter 4 elementos: uma string com o nome da informação armazenada no tuplo (idades ou pesos), e três inteiros com os valores das médias de idades/pesos globais, de mulheres e de homens.

4.4. Analise o conteúdo do ficheiro binário `ExemploBinario`, disponível na página da disciplina. Qual a informação contida nesse ficheiro?

4.5. Escreva um programa em Python que leia a informação desse ficheiro e a escreva para um novo ficheiro de texto: `ExemploBinario.txt`. Compare e discuta o tamanho dos dois ficheiros.

5. Escreva um programa que permita ler o ficheiro `medias2.txt` criado na questão 4.2 e apresente o seu conteúdo de forma mais amigável (e não em formato *pickle*), tal como no ficheiro `medias.txt`.

6. Desenvolva um programa que permita comparar o tempo que demora a copiar/escrever uma lista com 500 números inteiros (de 1 a 500) em quatro situações distintas:

a) escrita em memória
b) escrita numa zona de memória partilhada
c) escrita assíncrona num ficheiro binário
d) escrita síncrona num ficheiro binário

Antes de medirem os tempos das cópias/escritas, os alunos devem começar por preencher a lista original com os 500 números. Depois, caso a caso:

i. para testar o tempo que demora a escrita em memória, basta fazer a cópia da lista original para uma nova lista.

ii. para testar o tempo que demora a escrita numa zona de memória partilhada, basta inicializar um *array* com a lista original. Ver mais informação em
https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Array

iii. para testar o tempo que demora a escrita num ficheiro binário da forma mais precisa possível, essa escrita não deve ser colocada num ciclo, copiando 4 bytes de cada vez, como fizemos no exercício do tutorial. É preferível copiar toda a lista como uma única instrução usando a função pack do módulo `struct` da seguinte forma:

```
struct.pack("i"*len(dados_a_escrever),*dados_a_escrever))
```

iv. finalmente, uma escrita síncrona é uma em que se garante que, depois de se correr a função `write()`, os dados são mesmo copiados para disco (por razões de eficiência, numa escrita assíncrona é o sistema operativo que decide quando copiar para o disco). Para garantir que se escreve logo no disco, deve efetuar-se a escrita tal como no ponto anterior e executarem-se operações adicionais para garantir escrita síncrona, tal como explicado aqui:
https://docs.python.org/3/library/os.html#os.fsync

7. Discuta os resultados do exercício anterior. Se necessário, teste o programa com listas de tamanhos diferentes. Para aumentar o grau de confiança nos resultados, pode repetir cada uma das experiências 100 vezes, por exemplo, e retornar a média das 100 observações.

---

3. Bibliografia e outro material de apoio

[1] https://docs.python.org/3/tutorial/inputoutput.html

[2] https://docs.python.org/3/library/signal.html

[3] https://docs.python.org/3/library/time.html