

# Programação em Python

## Objetivos:

- A linguagem Python
- Variáveis
- Condições, Ciclos e Funções
- Listas e Dicionários

### 12.1 Linguagem Python

A linguagem *Python* [1] é uma das linguagens de uso geral mais populares e tem uma extensa documentação [2]. É considerada uma linguagem de alto nível por abstrair os conceitos fundamentais do computador, focando-se no desenvolvimento rápido de algoritmos, na fácil compreensão do código produzido e na sua estrutura. O programador foca-se na expressão do algoritmo sem necessidade de compreender aspectos de baixo nível.

Os programas escritos em *Python* são tipicamente bastante compactos (considerando o número de linhas), especialmente em comparação com outras linguagens como *Java* ou *C*. Estas características tornam a linguagem *Python* ideal para a prototipagem rápida de sistemas, para o desenvolvimento de sistemas complexos e, através dos módulos que possui, como ferramenta para cálculo científico.

*Python* é uma linguagem que suporta múltiplos paradigmas (ou estilos) de programação. Enquanto *Java* requer obrigatoriamente a utilização de classes, uma característica da programação orientada a objetos pura, a linguagem *Python* suporta esse paradigma, mas não o impõe, permitindo outros como a programação funcional ou a programação imperativa procedural.

Visto ser uma linguagem interpretada, não são necessários os passos de compilação e ligação a bibliotecas, como noutras linguagens. Isto possibilita que o mesmo código seja

facilmente executado em múltiplos sistemas operativos e que partes de um sistema sejam alterados dinamicamente durante a execução de uma aplicação.<sup>1</sup>

Em Laboratórios de Informática, *Python* será a linguagem principal para a exploração de vários conceitos do domínio da informática. As secções seguintes são o primeiro passo nessa direção, introduzindo *Python* para a resolução de problemas de programação comuns.

## 12.2 Características Básicas

*Python* é uma linguagem com um conjunto de características particulares, que a distingue de outras linguagens vulgarmente utilizadas:

**Uso geral:** Pode ser utilizada para o desenvolvimento de qualquer tipo de aplicações ou serviços, não sendo uma linguagem para um nicho específico.

**Interpretada:** Os programas são processados à medida que é necessário executar cada pedaço de código. Não é necessário compilar o programa antes de ser utilizado.

**Alto nível:** Não são expostos detalhes da plataforma de computação, como registos, ponteiros, ou endereços de memória. O programador foca-se na implementação de um algoritmo e não nos detalhes do *hardware*.

**Tipos dinâmicos:** As variáveis não têm um tipo fixo e por isso não precisam de ser pré-declaradas. O tipo da variável está sempre associado ao valor que tem armazenado e pode modificar-se dinamicamente, bastando para isso atribuir-lhe um valor de tipo diferente.

**Tipagem forte:** Em todas as operações, os tipos dos operandos são verificados. Não há conversões implícitas entre tipos de dados que não sejam naturalmente compatíveis. Isto contrasta com a tipagem fraca do Javascript, por exemplo.

**Gestão automática de memória:** A alocação e libertação de memória dinâmica é gerida de forma automática, não sendo este detalhe exposto ao programador.

Os programas *Python* são ficheiros de texto, tipicamente com a extensão `.py`, que são executados através de um interpretador de *Python*. O exemplo seguinte demonstra o conteúdo de um programa *Python* (`hello.py`) que imprime um texto curto no ecrã:

---

```
print('Gosto de LabI')
```

---

<sup>1</sup>A alteração de um ficheiro não implica qualquer recompilação.

Uma forma alternativa de o fazer seria através do módulo **sys**:

---

```
import sys
sys.stdout.write('Gosto de LABI\n')
```

---

Note que neste caso é necessário especificar de forma explícita o final de linha através do carácter "**\n**". No entanto permite maior controlo sobre a escrita para a consola.

Em qualquer dos casos, o programa executa-se através do comando:

---

```
python3 hello.py
```

---

O interpretador de *Python* executa programas em ficheiros, mas também pode ser usado interativamente. Quando se invoca o interpretador sem argumentos, ele processa as instruções que vierem do dispositivo de entrada (o terminal). Neste modo de funcionamento aparece um *prompt* com o formato **>>>** e o utilizador pode introduzir instruções ou expressões de *Python*, que são executadas linha a linha e produzem resultados imediatos. Desta forma, o utilizador interage com uma interface de linha de comandos que interpreta *Python*, o que é muito útil para explorar a linguagem e testar ideias ou resolver pequenos problemas. Para terminar o interpretador pode ser utilizada a sequência **CTRL+D** (em Windows usa-se **CTRL-Z**), que sinaliza o fim do “ficheiro” de entrada de dados. O exemplo abaixo demonstra uma breve interação com o interpretador de *Python*.

---

```
user@host:~$ python3
Python 3.5.3 (default, Jan 19 2017, 14:11:04)
[GCC 6.3.0 20170118] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Gosto de LABI')
Gosto de LABI
>>> ^D
```

---

Em modo interativo, o interpretador tem um comportamento ligeiramente diferente do modo *offline*. Em particular, quando se avalia uma expressão em modo interativo, o seu resultado é apresentado imediatamente. Assim, pode usar-se o interpretador como uma calculadora avançada.

---

```
>>> 3+5
8
>>> 'cara' + 'pau'
'carapau'
```

---

```
>>> print('cara' + 'pau')
carapau
```

---

Note como a representação interna de um valor pode diferir do seu valor impresso.

### Exercício 12.1

Faça um programa com as 3 linhas do exemplo anterior e execute-o. Repare que os resultados das duas primeiras linhas não são impressos, apenas a terceira linha produz um resultado visível. Esse é o comportamento esperado em modo não interativo.

Um ponto importante é que existem duas versões em utilização da linguagem: 2.x e 3.x. Estas duas versões são maioritariamente compatíveis entre si, apenas com algumas diferenças relevantes para estes laboratórios. Os exercícios aqui propostos são apresentados utilizando a versão 3.x, mas podem ser facilmente convertidos para a versão 2.x.

As diferenças relevantes são:

---

```
#Impressão de valores
#Python 2
print 'Hello World'

# Python 3
print('Hello World')

#Entrada de valores
#Python 2
x = raw_input('Insira X')

#Python 3
x = input('Insira X')
```

---

#### 12.2.1 Estrutura de um programa

Um programa em *Python* é composto por uma zona inicial onde são declaradas as importações de módulos, a que se seguem as definições de variáveis, de funções e/ou de classes, com a implementação do algoritmo necessário. Um aspecto importante é que não existem delimitadores de blocos explícitos como na linguagem *Java*. Em vez disso, cada instrução ocupa uma linha e é a indentação dessa linha que determina a que bloco pertence a instrução. O código a seguir demonstra esta característica da linguagem.

```
1 # encoding=codificação
2
3 # Zona de importação de módulos
4 import modulo1
5 import modulo2
6
7 # Funções
8 def funcao(a):
9     print(a)
10    if a:
11        print('SIM')
12    return False
13
14 # Outras instruções
15 funcao(True)
16 print('d')
```

---

A linha 8 do código define uma função (isto será abordado na Secção 12.2.6), e a linha 9 pertence a essa função apenas porque está mais indentada. A linha 11 possui uma indentação ainda maior, indicando que pertence a um sub-bloco, neste caso de uma instrução condicional. Por sua vez a instrução na linha 12 já não pertence à instrução condicional.

Note ainda que as linhas 8 a 12 definem uma função, mas esta só é executada quando invocada, na linha 15. Também não existe qualquer indicador de fim de instrução, sendo que esta corresponde a toda uma linha.

Sequências iniciadas pelo carácter `#` são comentários, que se estendem até ao fim de linha. Em *Python* não existem comentários de múltiplas linhas, pelo que cada linha de comentário tem de começar com o carácter `#`.

Em *Python* 3 também é facilitado o processamento de texto com caracteres acentuados, suportanto *unicode* de forma nativa. Em *Python* 2 é importante definir explicitamente a codificação dos ficheiros (ver primeiro comentário do exemplo anterior).

### 12.2.2 Valores, variáveis e tipos

As variáveis são declaradas no momento em que lhes é atribuído um dado valor. Em *Python* todas as variáveis são referências, não existindo a distinção entre variáveis primitivas e não primitivas que existe na linguagem *Java*. O exemplo seguinte define duas variáveis `a` e `b` e imprime a soma de ambas.

---

```
a = 3
b = 5.0
print(a + b)
```

---

De notar que o valor de **a** é do tipo inteiro (**int**) e o de **b** é real (**float**), que são dois (sub)tipos numéricos compatíveis em operações aritméticas. A adição de inteiros produz um inteiro, mas uma adição com um **float** produz um **float**, como seria de esperar. Pelo contrário, o operador de divisão em Python 2, tal como em Java ou C, tem um comportamento inesperado para um leigo, como se pode verificar no exemplo seguinte,

---

```
a = 3
b = 5.0
print a / b
b = 5
print a / b
```

---

que produz o resultado abaixo.

---

```
0.6
0
```

---

A primeira divisão, entre um inteiro e um real produziu o quociente real, mas na segunda, entre dois inteiros, o resultado é o quociente da divisão inteira. Este comportamento ambíguo do operador **/** é propício a erros de programação e por isso recomenda-se que se use sempre o operador **//** quando se pretende a divisão inteira. Na versão 3 do Python a ambiguidade foi mesmo eliminada: o operador **/** dá sempre o quociente real, mesmo quando usado entre inteiros. O exemplo anterior também demonstra a variação dinâmica do tipo da variável **b**.

## Exercício 12.2

Crie um ficheiro *Python* e verifique o resultado de adicionar, subtrair, multiplicar, dividir (**/** e **//**) e achar o resto da divisão inteira (operador **%**) de diferentes valores. Use valores positivos, negativos e nulos, inteiros e reais.

Se puder, execute o programa com o Python 2 e com o Python 3.

Em *Python*, os operadores de quociente (**//**) e resto (**%**) de divisão inteira têm resultados diferentes dos equivalentes em Java ou C quando o dividendo é negativo. Nenhuma das linguagens está errada. Tratam-se apenas de interpretações diferentes, mas ambas matematicamente coerentes, da operação de divisão inteira. Na opinião dos autores, a interpretação adotada no Python tem vantagens na maioria das aplicações práticas.

## Strings

Um dos tipos mais utilizados para além dos tipos numéricos é talvez o tipo *String*. Em *Python* ele define uma sequência com zero ou mais caracteres. Não existe um tipo específico para caracteres isolados como o **char** do Java. Um carácter é tratado como um caso especial de uma *String* com comprimento um. Isto vai de encontro a um dos princípios da linguagem *Python* que diz *Special cases aren't special enough to break the rules*.

Valores do tipo string indicam-se entre aspas ('**string**') ou entre plicas ('**string**').

---

```
# encoding=utf-8

a = 'Laboratórios'
b = ' de '
c = "Informática"
print(len(c)) # qual o resultado?
print(a+b+c) # concatenação de strings
```

---

O exemplo seguinte demonstra como se pode aceder a partes de uma string. Note que a sintaxe é semelhante ao modo como são acedidos os arrays em Java, mas permitindo também selecionar subsequências.

---

```
# encoding=utf-8

a = 'Laboratórios'
b = ' de '
c = 'Informática'

print(a[0:3]+'-' +c[0]+ ' '+str(2018)) # Imprime Lab-I 2018
```

---

### Exercício 12.3

Reita os exemplos anteriores com outras variáveis do tipo *String*. Experimente operações matemáticas como a adição e multiplicação com inteiros (sem utilizar **str**).

Existem várias funções auxiliares que permitem manipular variáveis do tipo *String*. Nominalmente, há funções para a determinar o comprimento da string, para converter para maiúsculas ou minúsculas, para retirar espaços, etc. Para uma lista completa, consultar <http://docs.python.org/3/library/stdtypes.html#string-methods>.

---

```
# encoding=utf-8

a = ' Laboratórios de Informática '

print(len(a))      # Comprimento de uma string
print(a.lower())    # Converte para minúsculas
print(a.upper())    # Converte para maiúsculas
print(a.title())    # Converte para título
print(a.find('t'))  # Primeira posição de 't'
print(a.isalpha())   # Verifica se só tem letras
print(a.isdigit())   # Verifica se é um número
print(a.islower())   # Verifica se tem só minúsculas
print(a.strip())     # Remove espaços nos extremos
print(a.split(' '))  # Divide por espaços
```

---

### Exercício 12.4

Implemente um pequeno programa que experimente as funções apresentadas e outras que encontre na documentação da linguagem Python.

Não existe uma função **printf**, mas é possível criar uma string com uma dada formatação e imprimi-la. Para isso pode usar-se o operador de formatação **%** ou o método **format**, que inserem valores em locais assinalados numa string de especificação de formato. A sintaxe da especificação de formato é semelhante à usada em C e Java. O exemplo seguinte produz a linha **Benfica 4, Porto 0** a partir de uma formatação de *Strings*.

---

```
equipa1 = 'Benfica'
equipa2 = 'Porto'
golos1 = 4
golos2 = 0
print('{:s} {:d}, {:s} {:d}'.format(equipa1, golos1, equipa2, golos2))
print('%s %d, %s %d' % (equipa1, golos1, equipa2, golos2))
```

---

### Exercício 12.5

Implemente um exemplo semelhante ao anterior, mas referente a cursos da Universidade.

Por vezes é necessário converter *Strings* em valores numéricos e vice versa. A conversão de valores numéricos para *String* é conseguida através da função **str()**, enquanto que

a conversão inversa é conseguida através das funções `float()` ou `int()`. O exemplo seguinte converte valores de e para *String*, imprimindo-os de seguida.

---

```
a = 3
sa = str(3)
b = int(sa)
c = float(sa) * 1.2
print('{:d}, {:s}, {:d}, {:.2f}'.format(a, sa, b, c))
```

---

## Listas

Em *Python*, as listas são as estruturas de dados nativas com maiores semelhanças aos *arrays* usados noutras linguagens, mas são bastante mais versáteis. Estas estruturas são compostas por uma sequência de valores, que podem ser accedidos através de um índice. O primeiro valor corresponde ao índice 0 e o final ao índice `len(array)-1`. O exemplo seguinte demonstra a definição de uma lista com os cursos do DETI e a sua impressão de várias formas. A primeira forma imprime toda a lista, a segunda forma imprime apenas o primeiro elemento, enquanto a terceira forma imprime todos os valores entre o primeiro e o terceiro (exclusive).

---

```
l = ['MIECT', 'LEI', 'MEI', 'MSI', 'MIEET']
print(l)
print(l[0])
print(l[0:2])
```

---

Uma vantagem das listas é o facto de a sua dimensão ser dinâmica, sendo possível adicionar mais elementos a uma lista através dos métodos `append` e `extend`. O método `append` acrescenta um novo elemento ao fim da lista, enquanto o método `extend` permite estender uma lista com outra. A lista criada no exemplo anterior poderia ser refeita através destes métodos da seguinte forma:<sup>2</sup>

---

```
l = []
l1 = []
l2 = []

l1.append('MIECT')
l1.append('LEI')
l1.append('MEI')
l2.append('MSI')
```

---

<sup>2</sup>Embora estes métodos sejam úteis, não existe qualquer vantagem nesta implementação, sendo preferido o método anterior.

```
l2.append('MIEET')

l.extend(l1)
l.extend(l2)

print(len(l))
```

---

### Exercício 12.6

Crie um programa que declare uma lista e imprima o seu conteúdo. Imprima partes, toda a lista e estenda o seu conteúdo.

### Exercício 12.7

Verifique qual o resultado de aplicar a função `sorted` a uma dada lista.

Uma lista importante é a que é utilizada para fornecer ao programa os argumentos na linha de comandos. Pode ser acedida através da variável `sys.argv` definida no módulo `sys`. Esta lista contém todos os argumentos passados ao programa, incluindo o próprio nome do programa. O programa seguinte imprima os valores dos argumentos que lhe são passados. Experimente executá-lo com e sem argumentos.

---

```
import sys
print(sys.argv)
```

---

### Exercício 12.8

Utilizando a lista `sys.argv`, implemente um programa que calcule a soma do primeiro e segundo argumento.

## Dicionários

Os dicionários são semelhantes às listas, no sentido em que estabelecem uma associação entre uma chave (índice no caso da lista) e um valor. No entanto, os dicionários permitem que se possa definir tanto a chave como o valor. Os dicionários são vantajosos em determinadas aplicações pois permitem ter uma estrutura em que o acesso é efetuado através de uma chave com significado para o programador. A sintaxe básica para criar um dicionário é a seguinte:

---

```
nome = {'chave1': valor1, 'chave2': valor2, ... }
```

---

Para aceder a um elemento usa-se a forma `nome['chave']`.

---

```
nome = {'chave1': 0} # Cria um dicionário com uma chave

nome['chave1'] = 1 # Redefinição do valor
nome['chave2'] = 2 # Definição de um novo par <chave, valor>

print(nome['chave1'])
print(nome['chave2'])
```

---

O exemplo seguinte mostra a criação de uma lista de alunos, a impressão do nome do primeiro elemento, seguida da impressão de toda a lista. Cada aluno é um dicionário com nome e número mecanográfico.

---

```
l = []

l.append( {'nome': 'Catarina', 'mec': 4534} )
l.append( {'nome': 'Pedro', 'mec': 1234} )
l.append( {'nome': 'Joana', 'mec': 5354} )
l.append( {'nome': 'Miguel', 'mec': 6543} )

print(l[0]['nome'])
print(l)
```

---

### Exercício 12.9

Crie um dicionário que permita armazenar as pontuações de um jogo de futebol entre duas equipas.

#### 12.2.3 Entrada de dados da consola

Em *Python*, a leitura de dados do teclado pode ser efetuada através da função `raw_input('mensagem')`. Esta função imprime a mensagem passada como parâmetro (um prompt) e espera pela introdução de uma linha de texto. Essa linha é devolvida como string e poderá depois ser convertida para um valor inteiro, real ou de outro tipo através das funções `int()`, `float()` ou outra.

No exemplo seguinte é implementada uma calculadora simples que multiplica 2 valores inseridos pelo teclado.

---

```
# encoding=utf-8

valor1 = float(input('Primeiro Valor: '))
valor2 = float(input('Segundo Valor: '))

print('Resultado: {} * {} = {:.10.6f}'.format(valor1, valor2, valor1 * valor2))
```

---

### Exercício 12.10

Implemente um programa que repita o texto inserido mas convertendo todos os caracteres para maiúsculas.

#### 12.2.4 Instruções condicionais

As instruções condicionais permitem executar blocos de código alternativos dependendo do valor lógico de uma ou mais condições. Em *Python* estas instruções são indicadas pelas palavras chave **if**, **else** e **elif**, e têm um modo de funcionamento semelhante ao de outras linguagens de programação, como pode constatar no exemplo abaixo. A diferença mais significativa é meramente sintática: decorre do uso de indentação para definir se um conjunto de instruções pertence a um bloco ou não.

---

```
if cond:
    instruções a executar em caso positivo
else:
    instruções a executar em caso negativo
```

---

A condição **cond** é geralmente uma expressão booleana que pode resultar de uma comparação, de um valor de uma variável ou de uma combinação de múltiplas expressões com operadores lógicos como **and**, **or**, ou **not**. O exemplo seguinte imprime “Sim” se um valor for divisível por 2 ou por 3:

---

```
a = ... #Valor
if a % 3 == 0 or a % 2 == 0:
    print('Sim')
else:
    print('Não')
```

---

### Exercício 12.11

Implemente um pequeno programa que calcule se um dado ano é bissexto ou não.

Um aspecto algo peculiar das condições em *Python* é que não têm de ser estritamente do tipo **bool** (o tipo de dados booleanos). Por exemplo, um valor numérico usado como condição é considerado verdadeiro sse for diferente de zero; uma string ou outro tipo de sequência é considerado verdadeiro sse tiver um tamanho superior a 0; o valor **None** é sempre considerado falso. O exemplo seguinte demonstra como seria possível imprimir a palavra “Par” caso um dado valor seja divisível por dois.

---

```
a = 3 # Ou outro valor
if not (a % 2):
    print('Par')
else:
    print('Impar')
```

---

### Exercício 12.12

Use esta funcionalidade para determinar se uma *String* é vazia, sem utilizar a função `len()`.

O mérito desta característica da linguagem é, no mínimo, questionável. A vantagem de poupar dois ou três carateres a expressar algumas condições não parece compensar o que se perde em termos de legibilidade e comprehensibilidade e simplicidade.

#### 12.2.5 Instruções de repetição

As instruções de repetição (ou ciclos) permitem executar blocos de instruções repetidamente. Existem duas palavras chave reservadas na linguagem para implementar ciclos: **for** e **while**. Noutras linguagens é comum estas duas instruções serem usadas sem grande diferenciação. Isto **não** é verdade na linguagem *Python*. Isto resulta da aplicação da regra “*There should be one—and preferably only one—obvious way to do it*”, impedindo que existam múltiplas instruções com a mesma funcionalidade.

- **for** - Percorre os elementos de uma sequência, repetindo um conjunto de instruções por cada um deles. Por exemplo, pode iterar sobre os carateres de uma *String*, ou sobre todos os valores entre 1 e 10.
- **while** - Executa repetidamente um conjunto de instruções enquanto uma condição for verdadeira.

O exemplo seguinte demonstra um ciclo **for**. Note que não há qualquer operação aritmética explícita para incrementar valores (ex, `i+=1`). Usou-se a função **range**, que

gera uma sequência de valores em progressão aritmética  $[0, 1, \dots, 9]$ , e o ciclo repete a instrução `print(i)` após atribuir cada um desses valores à variável `i`.

---

```
for i in range(0, 10):
    print(i)
```

---

A execução da função `range` pode ser analisada em detalhe se se executar o exemplo seguinte. O resultado deverá ser uma lista de valores entre 0 e 9.

---

```
print(range(0, 10))
```

---

### Exercício 12.13

Utilizando um ciclo `for`, implemente um programa que imprima uma sequência de Fibonacci. Uma sequência de Fibonacci é composta por números em que cada número é composto pela soma dos 2 anteriores. Uma sequência típica de 10 elementos será 1,1,2,3,5,8,13,21,34,55.

Aplicando um ciclo `for` a uma sequência do tipo *String* irá iterar sobre os seus caracteres. O exemplo seguinte imprime cada um dos caracteres de um texto.

---

```
a = 'Laboratórios de Informática'
for i in a:
    print(i)
```

---

### Exercício 12.14

Implemente um ciclo `for` que determine quantos dígitos existem numa frase. Pode recorrer à função `isdigit()` de uma *String*.

### Exercício 12.15

Implemente um ciclo `for` que permita contar quantas palavras existem numa frase. Recorra à função `split(' ')` para criar uma lista com a frase dividida pelos espaços.

### Exercício 12.16

Implemente um ciclo `for` que permita criar uma *String* que seja o inverso de outra ('abcde' -> 'edcba').

O ciclo **while** difere do ciclo **for** pois não percorre um conjunto de valores. Em vez disso, repete instruções enquanto uma condição for verdadeira. O exemplo abaixo usa um ciclo **while** para apresentar os valores entre 0 e 9. Neste caso é necessário inicializar e incrementar explicitamente a variável **i** de forma a esta variar o seu valor.

---

```
i = 0
while i < 10:
    print(i)
    i = i + 1
```

---

### Exercício 12.17

Inverta uma string, mas agora utilizando um ciclo **while**.

---

### Exercício 12.18

A utilização do ciclo **while** é mais adequada a cálculos com o índice, ao invés de iterar por um conjunto. Desta forma, é mais adequada a utilização deste ciclo em cálculos como o factorial de um número.

Use o ciclo **while** para calcular o factorial de um valor.

---

## 12.2.6 Funções

A utilização de funções permite reutilizar instruções, sem que exista duplicação de blocos de código, assim como isolar instruções que desempenhem operações específicas. Esta característica é parte integrante do conceito de modularidade, essencial para o desenvolvimento de aplicações com mais de umas dezenas de linhas.

Na linguagem *Python* as funções podem possuir parâmetros, tal como podem devolver valores por retorno. No entanto, não existe lugar à definição de tipos de valores. A função presente no exemplo seguinte permite calcular o número de valores pares entre **a** e **b - 1**. Para definição da função utiliza-se a palavra chave **def**, seguida do nome da função e dos seus parâmetros.

---

```
def pares(a, b):
    c = 0
    i = a
    while i < b:
        if i % 2 == 0:
            c = c + 1
        i = i + 1
    return c
```

---

Neste caso, a função é declarada mas não é realmente utilizada no programa. Para isso é necessário que seja invocada e só são automaticamente executadas instruções que não possuam indentação. De forma a executar esta função para os valores 1 e 10, poderia ser adicionada a linha `print(pares(1, 10))` ao final do ficheiro. O ficheiro resultante seria o apresentado de seguida.

---

```
def pares(a, b):
    ...
print(pares(1, 10))
```

---

### Exercício 12.19

Crie um programa que contenha uma função que calculo o número de múltiplos de 3 entre dois valores.

## 12.3 Ficheiros

Python permite aceder a ficheiros em modos de escrita e leitura, através das funções `open()`, `read()`, `readline()` e `write()`.

Em primeiro lugar é necessário criar uma representação do ficheiro que se pretende aceder. O exemplo seguinte abre o ficheiro “texto.txt” em modo de leitura.

---

```
f = open('texto.txt', 'r')
```

---

Para se abrir um ficheiro noutras modos de acesso, seria necessário utilizar os especificadores:

- ‘r’ - Modo de leitura. Não é possível escrever.

- 'w' - Modo de escrita. Se o ficheiro não existir, é criado. Se existir, é truncado.
- 'a' - Modo de adição. Escritas são adicionadas ao final do ficheiro.
- 'r+' - Modo de leitura e escrita. Se o ficheiro não existir, dá erro.

A partir deste momento a variável `f` terá uma representação do ficheiro e permite acesso ao mesmo. Para se efetuar uma leitura é necessário usar o método `read(tamanho)`. O parâmetro tamanho indica quantos *bytes* se devem ler. Se o valor for menor ou igual que 0, ou omitido, todo o ficheiro é lido para memória. Uma alternativa é utilizar o método `readline()` que obtém apenas uma linha.

**A leitura de um ficheiro completo para a memória deve ser usada com parcimónia. Pode justificar-se para ficheiros curtos ou quando se tenha de fazer acessos aleatórios ao conteúdo. Em situações de processamento sequencial, deve optar-se pela leitura incremental por blocos ou linhas.**

No final da leitura ou escrita, deve-se sempre fechar o ficheiro que se abriu. Um exemplo completo que imprime para o ecrã o conteúdo de um ficheiro, lendo uma linha de cada vez seria:

---

```
f = open('texto.txt', 'r')

while True:
    linha = f.readline()
    if linha == '':
        break
    print(linha)

f.close()
```

---

O ciclo `for` permite iterar sobre um grande número de estruturas de dados incluindo ficheiros. Neste caso, o ciclo `for` itera por ficheiros linha a linha. Assim o exemplo anterior poderia ser reescrito da seguinte forma:

---

```
f = open('texto.txt', 'r')

for linha in f:
    print(linha)

f.close()
```

---

### **Exercício 12.20**

Implemente um programa que imprima o número de caracteres, palavras e linhas de um ficheiro de texto.

### **Exercício 12.21**

Implemente um programa que imprima o conteúdo de um ficheiro, invertendo cada palavra.

As funcionalidades que permitem verificar se um ficheiro existe, se é um ficheiro ou um diretório, etc..., estão disponíveis através do módulo `os.path`. Usando programação defensiva, a verificação de existência de um dado ficheiro pode ser implementada através das seguintes linhas:

```
import os.path
import sys

fname = 'Ficheiro.txt'
if not os.path.exists(fname):
    sys.exit('Não existe')

if os.path.isdir(fname):
    sys.exit('É diretório')

if not os.path.isfile(fname):
    sys.exit('Não é ficheiro')

f = open(fname, 'r')
```

### **Exercício 12.22**

Melhore os 2 exercícios anteriores de forma a verificar se o ficheiro realmente existe e se pode ser acedido.

## 12.4 Para aprofundar o tema

### Exercício 12.23

Escreva um programa que determine a nota na época normal de um aluno de Laboratórios de Informática e que indique se o aluno está aprovado ou reprovado. Para esse fim o programa deve pedir as notas necessárias (MT1, MT2, MT3, MT4, AP1, AP2, P1 e P2) e calcular a nota final.

### Exercício 12.24

Escreva um programa que indique se um número (inteiro positivo) é primo.

### Exercício 12.25

Na terra do Alberto Alexandre (localmente conhecido por Auexande Aubeto), o dialecto local é semelhante ao português com duas exceções:

- Não dizem os Rs
- Trocam os Ls por Us

Implemente um tradutor de português para o dialecto do Alberto. Por exemplo “lar doce lar” deve ser traduzido para “ua doce ua”. A tradução deve ser feita linha a linha, até que surja uma linha vazia.

Para este exercício, considere a função `replace` (ver <http://docs.python.org/2/library/string.html#string.replace>).

### Exercício 12.26

Escreva um programa que leia uma lista de números e imprima a sua soma e a sua média. O fim da lista é indicado pela leitura do número zero, que não deve ser considerado parte da lista. (Note que se a lista for vazia, a soma será zero, mas a média não pode ser calculada.)

### **Exercício 12.27**

Escreva um programa que implemente o jogo “Adivinha o número!”.

Neste jogo, o programa deve escolher um número aleatório no intervalo [0; 100],<sup>a</sup> dando depois a possibilidade de o utilizador ir tentando descobrir o número escolhido. Para cada tentativa, o programa deve indicar se o número escolhido é maior, menor ou igual à tentativa feita. O jogo termina quando o número correcto for indicado, sendo a pontuação do jogador o número de tentativas feito (portanto o valor 1 será a pontuação máxima).

---

```
aimport random  
random.randint(0,100)
```

---

## **Glossário**

## **Referências**

- [1] Python Software Foundation, *Python Programming Language*, <http://www.python.org/>, [Online; acedido em 6 de Fevereiro de 2019], 2014.
- [2] ——, *Python Documentation*, <http://www.python.org/doc>, [Online; acedido em 6 de Fevereiro de 2019], 2014.



# Criptografia em Python

## Objetivos:

- Módulos e bibliotecas Python para lidar com criptografia
- Cálculo de sínteses de ficheiros
- Cálculo de sínteses de senhas
- Cifra de ficheiros com criptografia simétrica e assimétrica

## 13.1 Introdução

A criptografia é um conceito antigo de transformação de conteúdos, que até há cerca de duas décadas era sobretudo usado em ambientes onde a segurança é um elemento fundamental (ambientes militares, serviços de informações, etc.). Hoje em dia, em virtude da massificação do uso da informática e da Internet para os mais variados fins, a segurança criptográfica faz parte do dia-a-dia do cidadão comum, mesmo que disso ele não se aperceba (por exemplo, quando usa comunicações seguras usando HTTPS).

O objetivo desta aula é o de mostrar como se conseguem realizar as transformações criptográficas mais comuns usando módulos Python.

Para uma referência mais profunda sobre a segurança e as suas aplicações recomenda-se a consulta da bibliografia existe **1996-schneier2013-zuquete**

## 13.2 Funções de síntese

As funções de síntese (*digest*) não são funções de cifra convencionais, como as que veremos nas demais secções, mas são funções que usam princípios relacionados com a criptografia no seu modelo de operação. Para além disso, são muitas vezes usadas em conjugação com funções de cifra para as completar de alguma forma (por exemplo, para calcular chaves de cifra de dimensão fixa a partir de senhas de dimensão arbitrária).

O objetivo de uma função de síntese é o de calcular um valor de dimensão fixa (em número de bits) a partir de conteúdos constituídos por conjunto arbitrários de bits. Normalmente diz-se que estas funções permitem criar *impressões digitais informáticas* (*digital fingerprints*) de conteúdos, porque é muito difícil (por requisito) encontrar dois conteúdos que tenham a mesma síntese, assim como é difícil encontrar dois humanos com as mesmas impressões digitais. Também é comum designar o valor calculado por estas funções como *soma de controlo* (*checksum*), muito embora existam inúmeras funções de cálculo de somas de controlo que não possuem as propriedades (criptográficas) das funções de síntese.<sup>1</sup>

As funções de síntese mais usadas são a MD5, a SHA-1, SHA-256 e SHA-512, mas há muitas mais. Estas funções usam-se de forma similar, mas produzem resultados de dimensão diferente (128, 160, 256 e 512 bits, respetivamente).

A forma usual de aplicar uma função de síntese num programa consiste sem seguir os 4 passos seguintes:

1. Iniciar o seu contexto interno;
2. Adicionar dados para serem processados pela função;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar;
4. Calcular a síntese resultante de todos os dados fornecidos à função.

Em Python o módulo `hashlib`<sup>2</sup> possui as funções de síntese mais usuais. O exemplo seguinte mostra a utilização da função MD5 para calcular a síntese de uma frase longa dividida em duas partes.

---

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()                                     # Iniciar contexto
>>> h.update("A long sentence ".encode('utf-8'))        # Adicionar dados
>>> h.update("broken in two halves".encode('utf-8'))    # Adicionar mais dados
>>> print(h.hexdigest())                                # Calcular síntese
f2b8308b3e84e032f5d7e6dee84e647a
```

---

O resultado apresentado por este programa é uma sequência de 32 algarismos hexadecimais. Cada um desses algarismos representa 4 bits ( $0 \rightarrow 0000$ ,  $1 \rightarrow 0001$ ,  $2 \rightarrow 0010$ ,  $\dots$ ,

<sup>1</sup>Por outras palavras, as funções de síntese podem ser consideradas como funções de cálculo de somas de controlo, enquanto o inverso não é verdade em geral.

<sup>2</sup>Disponível em <https://docs.python.org/3/library/hashlib.html>

$8 \rightarrow 1000$ ,  $9 \rightarrow 1001$ ,  $a \rightarrow 1010$ ,  $\dots$ ,  $d \rightarrow 1101$ ,  $e \rightarrow 1110$ ,  $f \rightarrow 1111$ ). Logo, o resultado possui  $32 \times 4$  bits, ou seja, 128 bits.

O resultado apresentado seria o mesmo se a frase tivesse sido fornecida apenas de uma vez e não dividida em duas metades:

---

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update( "A long sentence broken in two halves".encode('utf-8') )
>>> print(h.hexdigest())
f2b8308b3e84e032f5d7e6dee84e647a
```

---

Porém, qualquer pequena alteração do texto processado pela função de síntese muda o resultado de forma radical:

---

```
$ python3
>>> import hashlib
>>>
>>> h = hashlib.md5()
>>> h.update( "A long sentence".encode('utf-8') )      # trailing space removed!
>>> h.update( "broken in two halves".encode('utf-8') )
>>> print(h.hexdigest())
1d0b93b21eb945593abab4b1a04456d6
```

---

### Exercício 13.1

Faça um programa que calcule e apresente a síntese de ficheiros usando a função de síntese SHA-1 (cujo nome, no módulo `hashlib`, é `sha1`). Os nomes dos ficheiros deverão ser passados como argumentos ao programa (acessíveis através da variável `sys.argv`). Confirme os resultados apresentados pelo seu programa confrontando-os com os apresentados pelo comando `sha1sum` disponível na linha de comando UNIX.

## Exercício 13.2

Reescreva o programa anterior para calcular a síntese de cada ficheiro usando blocos 512 octetos de cada vez. Use, para esse fim, a função `read` para ler octetos dos ficheiros<sup>a</sup>:

---

```
f = open(name, "rb")
buffer = f.read(512)

# len(buffer) == 0 --> End-of-file reached
# len(buffer) > 0 --> buffer has len(buffer) bytes

while len(buffer) > 0:
    ...
    buffer = f.read(512)
```

---

Compare os resultados obtidos neste exercício com os do exercício anterior (não deverão mudar!).

---

<sup>a</sup>Torna-se útil abrir o ficheiro em modo de leitura binária usando "rb"

---

## 13.3 Biblioteca `pycrypto`

A biblioteca (*toolkit*) `pycrypto`<sup>3</sup> possui uma coleção muito interessante de funcionalidades relacionadas com criptografia, incluindo funções de síntese. Daqui em diante iremos usar esta biblioteca.

### 13.3.1 Instalação

O interpretador de Python tipicamente é acompanhado por duas ferramentas que auxiliam a instalação de módulos adicionais, necessários aos programas. Estes programas são respetivamente o `easy_install` e `pip`. Para esta aula será necessário instalar a biblioteca `pycrypto`, que fornece mecanismos para a utilização de métodos criptográficos.

Para instalar a biblioteca, no caso de se utilizar um sistema pessoal com Ubuntu ou Debian, será necessário executar:

---

```
sudo apt install python3-crypto
```

---

ou em alternativa, usando um método mais demorado

---

<sup>3</sup><https://www.dlitz.net/software/pycrypto/>

```
sudo apt-get install build-essential python3-dev python3-pip
```

---

seguido de:

```
pip3 install --user pycrypto
```

---

Repare na utilização da opção `--user`, que indica que as bibliotecas adicionais deverão ser instaladas apenas para o utilizador atual (e não para todo o sistema). Isto é importante pois os utilizadores comuns não possuem permissões para instalar bibliotecas no sistema.

Pode-se confirmar a existência da biblioteca executando os seguintes comandos:

```
$ python3
>>> from Crypto.Hash import MD5
>>>
>>> h = MD5.new()
>>> h.update( "A long sentence ".encode('utf-8') )
>>> h.update( "broken in two halves".encode('utf-8') )
>>> print(h.hexdigest())
f2b8308b3e84e032f5d7e6dee84e647a
```

---

### 13.3.2 Utilização

O programa antes indicado que usava MD5 agora escrever-se-á assim para usar a função MD5 da biblioteca `pycrypto`:

```
$ python3
>>> from Crypto.Hash import MD5
>>>
>>> h = MD5.new()
>>> h.update( "A long sentence ".encode('utf-8') )
>>> h.update( "broken in two halves".encode('utf-8') )
>>> print(h.hexdigest())
f2b8308b3e84e032f5d7e6dee84e647a
```

---

### Exercício 13.3

Altere o programa que antes desenvolveu com a função SHA-1 do módulo `hashlib` para usar a função SHA-256 da biblioteca `pycrypto`. Confirme os resultados confrontando-os com os produzidos pelo comando `sha256sum`.

## 13.4 Cifras simétricas

As cifras simétricas são funções de transformação (reversível) de conteúdos que usam duas chaves iguais na cifra e na decifra. Ou seja, se se cifrar um conteúdo original  $T$  com a função de cifra  $E$  e a chave  $K$ , produzindo o criptograma  $C$ , poder-se-á recuperar  $T$  a partir de  $C$  com a função de decifra  $D$  e a mesma chave  $K$ .

As cifras simétricas subdividem-se em duas grandes famílias: as contínuas (*stream*) e as por blocos.

### 13.4.1 Cifras contínuas (*stream cyphers*)

As cifras contínuas produzem um criptograma  $C$  por mistura de um conteúdo original  $T$  com uma chave contínua (*keystream*)  $KS$ . A decifra consiste em retirar do criptograma a componente  $KS$  que lhe foi misturada usando uma função inversa da de mistura. Por simplicidade, a função de mistura e a sua inversa são exatamente a mesma: a adição módulo 2 de bits, vulgarmente designada por **XOR** (de *eXclusive OR*, cujo símbolo matemático é  $\oplus$ ).

Se  $A$  e  $B$  forem bits, que podem tomar os valores 0 e 1, a operação  $\oplus$  é a seguinte:

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

É fácil de ver que sempre que um operando é 1, o resultado é o inverso do outro operando; pelo contrário, sempre que um operando é 0, o resultado é o valor exato do outro operando. Daqui resulta a seguinte propriedade, para qualquer valor de  $X$ :

$$A \oplus X \oplus X = A$$

Logo, é fácil de mostrar que a cifra e decifra se podem fazer usando a operação **XOR** e a mesma chave contínua  $KS$ :

$$\begin{aligned} C &= T \oplus KS \\ T &= C \oplus KS = T \oplus KS \oplus KS = T \end{aligned}$$

A diferença entre as várias cifras contínuas está na forma como é produzido o valor de  $KS$ . A grande maioria das cifras contínuas usa um gerador pseudoaleatório, controlado por uma chave de dimensão fixa, para gerar  $KS$  (tanto para a operação de cifra como para a de decifra).

As cifras contínuas são muito usadas em comunicações rádio envolvendo equipamentos móveis, porque existem geradores muito simples de realizar em hardware e de baixo consumo. É o caso dos geradores A5 (usado nas comunicações GSM) e RC4 (usado inicialmente nas comunicações WiFi). Neste trabalho iremos usar esta última. O seu nome, porém, é diferente: ARC4 (de *Alleged ARC4*). A razão de ser desta diferença de nomes vem do facto do algoritmo RC4 ainda ser oficialmente secreto, sendo o ARC4 uma sua versão que alegadamente (e comprovadamente) é compatível com o RC4 (e que, por isso, é obviamente igual).

A forma usual de usar uma função de cifra/decifra num programa consiste sem seguir os 3 passos seguintes:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Adicionar dados para serem processados pela função e recolher o resultado da sua operação;
3. Repetir o passo anterior até que tenham acabado todos os dados do conteúdo a processar.

Note-se que com uma cifra contínua o processamento dos dados (na cifra ou na decifra) é por omissão feito sequencialmente, não se podendo cifrar ou decifrar zonas de dados por uma ordem arbitrária. Há cifras contínuas que permitem essa liberdade, mas a sua utilização é diferente. Neste guião não vamos explorar essa faceta.

### Exercício 13.4

Faça um programa em Python (`cifraComRC4.py`) que use a cifra RC4 (ou ARC4, na biblioteca `pycrypto`). O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual.

O RC4 suporta chaves com uma dimensão entre 40 e 2048 bits (5 a 256 octetos), pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável. Sugere-se a seguinte política: se a chave tiver menos do que 5 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-1). Se tiver mais dos que 256 octetos, deverão ser usados apenas os primeiros 256. Caso contrário, deverão ser usados exatamente os octetos fornecidos.

O programa deverá escrever o criptograma para o `stdout` (por omissão, a consola), o qual poderá ser redirigido para um ficheiro usando os mecanismos do interpretador de comandos:

```
python3 cifraComRC4.py ficheiro chave > criptograma
```

Em Python a escrita para o `stdout` pode ser feita da seguinte maneira:

```
import os
from Crypto.Cipher import ARC4

cipher = ARC4.new("chave")
cryptogram = cipher.encrypt("Text")
os.write(1, cryptogram)

decipher = ARC4.new("chave")
decrypted = decipher.decrypt(cryptogram)
print( decrypted.decode('utf-8') )

# se o conteúdo for binário, utiliza-se:
# os.write(1, decrypted)
```

### Exercício 13.5

Cifre vários ficheiros executando várias vezes o seu programa e verifique se o comprimento dos ficheiros resultantes, contendo os criptogramas, têm a mesma dimensão dos originais (têm de ter!).

### Exercício 13.6

Verifique que o programa está a funcionar corretamente decifrando o criptograma usando novamente a mesma chave:

```
python3 decifraComRC4.py criptograma chave > textoDecifrado
```

Após executar este comando deverá surgir, listado no ecrã, o conteúdo do ficheiro original (**ficheiro**).

### Exercício 13.7

Execute o comando anterior usando uma chave diferente e veja o resultado. Explique o sucedido.

#### 13.4.2 Cifras por blocos

As cifras por blocos consideram que os dados a transformar (e o resultado da sua transformação) são constituídos por blocos contíguos de dimensão constante; esta dimensão é imposta pela cifra. O modo mais simples de usar uma cifra (ou decifra) por blocos, denominado EBC (*Electronic Code Book*) consiste em realizar os seguintes passos:

1. Iniciar o seu contexto interno, normalmente indicando uma chave;
2. Selecionar o primeiro bloco dos dados a transformar para serem processados pela função e recolher o bloco resultante da sua operação;
3. Repetir o passo anterior para os blocos seguintes até que tenham acabado todos os dados do conteúdo a processar.

Este processo implica que a dimensão total dos dados a transformar seja múltipla da dimensão do bloco (diz-se que estão alinhados ao bloco). Porém, é natural que nem sempre assim aconteça, o que implica que se tenha de forçar esse alinhamento acrescentando dados extra, denominados excipiente (*padding*). Estes dados extra são acrescentados na cifra e removidos na decifra. Há várias maneiras de lidar com os excipientes, mas independentemente do método usado, é preciso indicar ao decifrador a sua presença e dimensão. Uma forma padrão de o fazer, denominada PKCS #7 **rfc5652**, consiste em fazer o seguinte:

- Acrescentar sempre excipiente, mesmo quando à partida não é necessário (por os dados a cifrar já estarem alinhados);

- Cada octeto do alinhamento tem um valor igual ao comprimento desse alinhamento.

Neste exercício vamos usar a cifra por blocos que é atualmente o padrão, denominada AES (*Advanced Encryption Standard*), que foi a vencedora de um concurso de cifras que terminou pouco depois do início do atual milénio. Esta cifra processa blocos de 128 bits (16 octetos) usando para o efeito chaves de 128, 192 ou 256 bits (16, 24 ou 32 octetos). Quando se usa a biblioteca **pycrypto**, a dimensão do bloco de uma cifra por blocos pode ser obtida através da variável **blocksize** de um objeto de cifra:

---

```
$ python3
>>> from Crypto.Cipher import AES
>>>
>>> key = '1234567890abcdef'      # Must provide a valid key (with 16, 24 or 32 bytes)
>>> cipher = AES.new( key )
>>> print(cipher.block_size)      # Prints the number of bytes in each block
16
>>> print(cipher.mode)           # Prints the cipher mode (1 for ECB)
1
```

---

Para se cifrar ou decifrar dados devem-se usar os métodos **encrypt** ou **decrypt**, respetivamente, do objeto de cifra:

---

```
$ python3
>>> from Crypto.Cipher import AES
>>>
>>> key = '1234567890abcdef'
>>> cipher = AES.new( key )
>>> x = cipher.encrypt( "texto para cifrar" )
>>> print(cipher.decrypt( x ))
b'texto para cifrar'
```

---

### Exercício 13.8

Faça um programa em Python (`cifraComAES.py`) que use a cifra AES. O programa deverá receber como argumentos o nome de um ficheiro a cifrar e uma chave textual.

O AES suporta chaves com uma dimensão exata de 16, 24 ou 32 octetos, pelo que deverão ser tomados cuidados para adaptar a chave fornecida pelo utilizador a algo que seja aceitável. Sugere-se a seguinte política: se a chave tiver menos do que 16 octetos (letras), deverá ser usada uma síntese da mesma (calculada, por exemplo, com SHA-1), de cujo resultado serão usados apenas os 16 primeiros octetos. Caso contrário, deverão ser usados apenas os primeiros 16 octetos da senha fornecida.

O programa deverá escrever o criptograma para o `stdout` (por omissão, a consola), o qual poderá ser redirigido para um ficheiro usando os mecanismos do interpretador de comandos

### Exercício 13.9

Faça o programa correspondente de decifra (`decifraComAES.py`). Note que o programa será fundamentalmente igual ao de cifra, mas deverá ter os seguintes cuidados:

- Como os criptogramas estão necessariamente alinhados, não deverá aceitar fazer a decifra de ficheiros que não tenham um comprimento alinhado à dimensão do bloco de cifra. A dimensão de um ficheiro pode ser obtida com a função `os.path.getsize(nome_do_ficheiro)`.
- Não se esqueça de retirar (não escrever) o excipiente colocado durante a cifra.  
Não se esqueça de que, se usou o método de colocação de excipiente descrito, existe sempre excipiente no último bloco do ficheiro cifrado!

## 13.5 Cifras Assimétricas

As cifras assimétricas são cifras que usam **duas chaves**, uma para cifrar e outra para decifrar (designadas por par de chaves). Uma destas chaves designa-se por privada e a outra por pública. A privada só é conhecida por uma entidade, que é dona do respetivo par de chaves; a pública pode ser universalmente conhecida. O conhecimento da chave pública não permite a dedução da correspondente chave privada. Estas cifras também são por vezes designadas por *cifras de chave pública*.

As cifras assimétricas, ou de chave pública, são historicamente muito recentes. Enquanto as cifras simétricas são tão antigas quanto a própria escrita, as assimétricas só existem desde meados de década de 1970. Neste exercício iremos usar a primeira cifra assimétrica

que foi publicada, denominada RSA, que é atualmente a mais usada.

### 13.5.1 RSA

Como se disse, as cifras assimétricas usam pares de chaves, uma privada e outra pública. O RSA permite cifrar com a pública e decifrar com a privada (para garantir confidencialidade) ou o inverso, cifrar com a privada e decifrar com a pública (para garantir autenticidade, o conceito que está na base das assinaturas digitais).

O RSA opera através da realização de operações matemáticas com números inteiros de grande dimensão (centenas ou milhares de bits). As operações são a exponenciação e o resto da divisão por um número inteiro. À combinação destas duas operações dá-se o nome de exponenciação modular.

Um par de chaves RSA possui 3 elementos:

- Um módulo,  $n$ , comum às componentes privada e pública;
- Um expoente,  $d$ , pertencente à componente privada;
- Um expoente,  $e$ , pertencente à componente pública.

Assim, a chave privada é formada pelo par de valores  $(d, n)$ , enquanto a chave pública é formada pelo par de valores  $(e, n)$ . As operações de transformação de dados usando estas chaves são as seguintes:

$$\begin{array}{ll} C = T^e \mod n & \text{Cifra com a chave pública (confidencialidade)} \\ T = C^d \mod n & \text{Decifra com a chave privada} \end{array}$$

$$\begin{array}{ll} C = T^d \mod n & \text{Cifra com a chave privada (autenticidade)} \\ T = C^e \mod n & \text{Decifra com a chave pública} \end{array}$$

onde a expressão  $x \mod n$  representa o resto da divisão inteira de  $x$  por  $n$  (em Python seria calculado com a expressão `x % n`).

Ao contrário das cifras anteriores, as cifras assimétricas não usam chaves indicadas por uma pessoa, nem uma pessoa é capaz de memorizar um par de chaves. Os pares de chaves são gerados por programas, usando para o efeito geradores aleatórios de bits, e as chaves geradas por esses programas têm de ser guardadas algures (por exemplo, em ficheiros) para poderem ser usadas mais tarde.

```

$ python3
>>> from Crypto.PublicKey import RSA
>>>
>>> keypair = RSA.generate( 1024 )
>>> fout = open( "keypair.pem", "wb" )
>>> kp = keypair.exportKey( "PEM", "senha" )
>>> fout.write(kp)
>>> fout.close()
...
>>> fin = open( "keypair.pem", "rb" )
>>> keypair = RSA.importKey( fin.read(), "senha" )
>>> fin.close()

```

---

O exemplo acima mostra como se gera um par de chaves RSA com um módulo de 1024 bits e se guarda o mesmo num ficheiro (**keypair.pem**) codificando o seu conteúdo em PEM (um formato textual) e protegendo a chave privada da observação de terceiros através da cifra com a senha **senha**. O conteúdo do ficheiro terá alguma semelhança com o seguinte:

---

```

-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: DES-EDE3-CBC,A8751314737B2A42

5JbCg5KVTxYUfheJBVS11G7VkJFf90M0+Q/1FgKRluV5DJWRfw9IzYgy1HBX8VL8
JdmqHo9HRmxdvLUZWSg3nn0UpVaRjiRzFjC+rxtHEWto9o8izmxBhozeaN1MWGEe
Kmt6B2+rrm/KFSrDonuz3r+GitfzGRM2nT+09D1aYQQsS+L7ZZBk/nZ5hLB082S
SWTEQZtrUEQho8o872GVANeA9M2A5urjdPKdav0Dw/CS8a/pD7s96cDEwFOV2Kyw
eG+TMeDDSG7SR+uzQGuEWpzuxciRmwxmlB8C4pxdWArhqu+/2feTt1uDH+PeBQS1
J9/WreLJVwNleo/ZXKhVbp+zL9+IT08b72NmzK1FsF0zk6heh9GIEFJNL6oGVRn8
eD2b1bs6KWVp0XoZrdMdLPDmbjbfaAn+RY15Rvgn0m2Jfs8g+iD1UeLhha05VTIq
3fYOL+QU3scaRUMvkyNbbVZ9/6Tj7GNgWZRMFn5oiKhEtkH9Hsa6esSmAoLmYUol
vYhs1eRk1LnhfiMlg8itfEaX1RKTi5BNne7GxgW63picPf9ryeKnITifpJS+G19+j
3uYOMqCohJqJwk+smYT/QSua7hRXjHLqwZAubhQ4iAVis1WMBUVOfTybF4K6Hub1
jQa/Mgf9hh1jynxohORUvAv+OkuNSICkQcyB2LGf4iKcHbj/5lf5xhpc1NoZVYCw
fARDGXFFHFnURBJ12XjKv8TIDuRFPeA6UxaKhEeD8QDWHc6GGQ7qApMpj8f60Dbk
cPfLAheh9yx7i+xU9rAfHeyu43rHaF4m3XonstPsNxtQR0p586SAQ==
-----END RSA PRIVATE KEY-----

```

---

### Exercício 13.10

Faça um programa em Python que gere um par de chaves e que o guarde num ficheiro. O programa deverá ter como parâmetros o nome do ficheiro para o par de chaves, a chave de cifra e o número de bits da chave.

A cifra RSA não é usada diretamente tal como indicado anteriormente nas expressões matemáticas. Com efeito, muito embora se usem as operações de exponenciação modular referidas, os valores que as mesmas processam na operações de cifra são pré-processados para obter algumas funcionalidades adicionais (controlo de erros e randomização). Há duas formas fundamentais de fazer esse pré-processamento, designadas por PKCS #1 v1.5 e PKCS #1 OAEP (*Optimal Asymmetric Encryption Padding*). Neste guião vamos considerar apenas esta última pois é a mais correta de ser utilizada em novas implementações.

---

```
$ python3
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import PKCS1_OAEP
>>>
>>> f = open( "keypair.pem", "r" )
>>> keypair = RSA.importKey( f.read(), "senha" )
>>> cipher = PKCS1_OAEP.new( keypair )
>>> # Encryption w/ public key
>>> x = cipher.encrypt( "The quick brown fox jumps over the lazy dog".encode('utf-8') )
>>> # Decryption with private key
>>> print(cipher.decrypt( x ).decode('utf-8'))
The quick brown fox jumps over the lazy dog
```

---

### Exercício 13.11

Altere os programas que usam AES para usarem também o RSA. O objetivo genérico é fazer a cifra do ficheiro com uma chave pública, recorrendo à privada para fazer a sua decifra. Na prática, vai-se recorrer à chamada cifra mista que tem um resultado semelhante mas um custo muito menor em termos de desempenho. O processo é o seguinte:

- Gera-se uma chave simétrica aleatória para cifrar os dados do ficheiro;
- Cifra-se a chave simétrica com a chave pública do destinatário e acrescenta-se o resultado ao ficheiro cifrado.

Para gerar a chave simétrica aleatória use a função `os.urandom()` com um parâmetro que indique o número de octetos aleatórios desejados. Use a senha fornecida pelo utilizador para decifrar a chave privada do par de chaves RSA.

## 13.6 Para Explorar

### Exercício 13.12

Avalie a performance das diferentes cifras e sínteses que foram exploradas. Para isto, use o módulo `time` para medir quantas operações são efetuadas em 10 segundos.

### Exercício 13.13

Sabe que pode enviar mensagens cifradas mesmo sobre canais não seguros, como um chat? Experimente cifrar mensagens de texto e, antes da impressão, converter o criptograma para Base64 usando o módulo `base64`. O resultado final será um texto com caracteres compatíveis com aplicações de comunicação por mensagens. Antes de as decifrar terá de reverter a codificação de Base64.

Consegue comunicar no canal de LABI com outro colega com quem tenha partilhado uma chave?

## Glossário

<b>AES</b>	Advanced Encryption Standard, cifra simétrica por blocos
<b>MD5</b>	Message Digest 5
<b>OAEP</b>	Optimal Asymmetric Encryption Padding
<b>RSA</b>	Cifra assimétrica, acrônimo dos nomes dos criadores (Rivest, Shamir, Adleman)
<b>SHA-1</b>	Secure Hashing Algorithm (versão 1)
<b>SHA-256</b>	Secure Hashing Algorithm (versão 2 com resultado de 256 bits)
<b>SHA-512</b>	Secure Hashing Algorithm (versão 2 com resultado de 512 bits)

# Testes e Depuração

## Objetivos:

- Test Driven Development
- Testes Unitários
- Testes Funcionais
- Depuração

## 14.1 Introdução

A validação do software é vital para que as equipas consigam realizar entregas dos componentes que desenvolvem, com alguma garantia do funcionamento do código em causa. Ao realizar testes de forma continuada e logo desde o início do desenvolvimento, é possível construir aplicações mais robustas e previsíveis. A existência de testes sistemáticos permite aferir o progresso e detetar regressões no desenvolvimento de uma aplicação. Quando se detetam problemas, é necessário encontrar a razão da anomalia, o que normalmente se faz através de depuração interativa e revisão do código desenvolvido. Este guiaão irá abordar ambos os temas, na perspectiva de pequenas equipas, ou programadores isolados, que pretendem desenvolver aplicações funcionais.

## 14.2 Desenvolvimento guiado por testes

A metodologia Test Driven Development (TDD) tem ganho adeptos nos últimos anos. Adota uma abordagem alternativa perante o desenvolvimento de aplicações, dando um ênfase especial à definição e execução de testes ao software. Considera-se que não é possível determinar qual o estado de uma aplicação se ela não for testada em todos os seus componentes. Sem testes sistemáticos é possível que alguns problemas passem despercebidos e venham a provocar problemas no futuro.

Segundo a TDD, o desenvolvimento de uma nova funcionalidade deve começar pela definição de testes à funcionalidade desejada, ainda antes de existir qualquer código. Claro que inicialmente todos os testes irão falhar, pois as funcionalidades não estão implementadas. A este estado dá-se o nome de *RED* (vermelho). À medida que se implementa cada funcionalidade e o teste respetivo é satisfeito, diz-se que estado da funcionalidade passa para *GREEN* (verde). De forma a tornar a aplicação consistente e evitar que se torne uma colagem de funcionalidades, depois de cada teste deverá ser realizada uma análise do que foi produzido e harmonização da solução, a fase *REFACTORING*. Este processo está representado na Figura 14.1. Quando todos os testes forem bem sucedidos, considera-se que a aplicação possui a funcionalidade desejada, para todos os casos de utilização previstos.

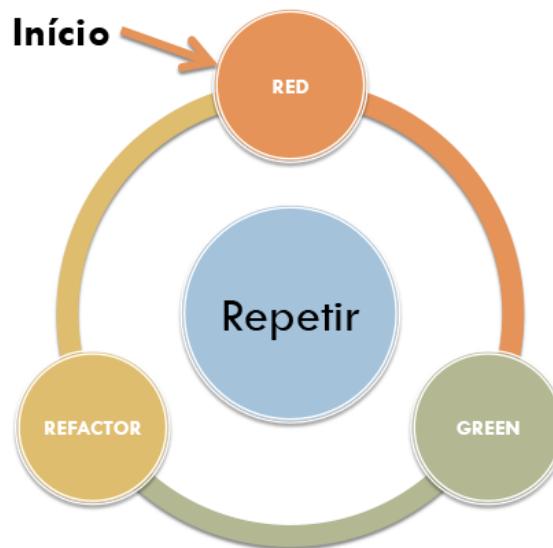


Figura 14.1: Fluxo de processos na metodologia TDD.

Esta metodologia é completamente independente da linguagem de programação, e pode implementar-se sem qualquer ferramenta específica. No entanto, na prática é comum utilizar ferramentas como *Jenkins*,<sup>1</sup> que de forma pontual executa testes a aplicações, permitindo acompanhar de forma detalhada qual o estado de desenvolvimento da aplicação. Permite igualmente que os programadores evitem um erro muito comum: começar a implementar código da primeira solução, ignorando todos os outros casos que o algoritmo também tem de considerar. Permite igualmente que se pense no problema de forma objetiva e em como será implementado, de uma forma mais distante e considerando o que seria a arquitetura ótima da solução. Começar imediatamente a programar implica que a solução final irá apenas realizar parte do que é pedido, os módulos irão comportar-se da

---

<sup>1</sup><http://jenkins-ci.org/>

maneira que dá mais jeito ao programador enquanto desenvolve, e nunca se terá uma caracterização completa da solução implementada.

É importante realçar que existem vários tipos de testes aos quais se pode sujeitar uma solução. Este guia irá focar-se no teste individual dos seus componentes (unidades) e no teste das funcionalidades necessárias da aplicação. Aplicações mais complexas irão necessitar de muitos outros testes.

### 14.3 Testes Unitários

Os testes unitários são testes aplicados pelos programadores às unidades que compõem os programas que desenvolvem. Uma unidade é um pequeno trecho de código que pode ser testado de forma independente, tal como: parte de uma função, uma função ou uma pequena classe. Podem e devem ser aplicados diversos testes a uma unidade, de forma a cobrir todos os casos de interesse, normais e excepcionais. Se um dado caso não for coberto por um dos testes, esse caso é considerado indeterminado, não se podendo presumir que a solução cobre a situação de forma correta.

Foram criadas diversas ferramentas, adequadas a cada linguagem, que permitem criar testes e automatizar a sua validação. Em *Python* pode-se encontrar o módulo **unittest** ou a ferramenta **py.test**, que iremos usar de seguida. Em *Java* a classe **JUnit** apresenta funcionalidades semelhantes. Muitas outras plataformas permitem realizar o mesmo. Encontra uma lista no endereço [http://en.wikipedia.org/wiki/List\\_of\\_unit\\_testing\\_frameworks](http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks).

O programa **py.test** permite a execução e validação de testes em programas Python, de forma simplificada.

#### Exercício 14.1

Verifique se o comando **py.test** está disponível no seu computador, executando:

```
$ py.test -3
```

Se aparecer uma mensagem parecida com esta:

```
platform linux -- Python 3.5.3, pytest-3.4.1, py-1.5.2, pluggy-0.6.0
rootdir: /home/debian/Desktop, inifile:
collected 0 items

===== no tests ran in 0.00 seconds =====
```

então já está instalado.

Se aparecer uma mensagem de erro, terá de instalar o pacote **pytest**. Para isso é necessário realizar **uma** das seguintes operações.

Se tiver permissões para a instalação de pacotes, num sistema *Ubuntu* ou *Debian*, pode executar como administrador:

---

```
$ apt install python3-pytest
```

---

Se não tiver permissões, pode instalar a aplicação na área pessoal usando um instalador de pacotes python:

---

```
$ pip3 install --user pytest      # instala o pacote
$ export PATH=~/local/bin:$PATH   # indica à shell onde encontrar o comando
```

---

Isto irá instalar a aplicação em `~/.local/bin/py.test`.

Vejamos um exemplo de utilização da ferramenta **py.test** para o desenvolvimento de uma função segundo a metodologia TDD.

Considere que pretendemos criar uma função chamada **fibonacci**, que determine os **n** primeiros valores da sequência de Fibonacci. Antes de iniciar a implementação devem especificar-se os testes, e estes devem cobrir tanto os casos considerados normais, como os casos especiais. Decidimos que a função irá aceitar um argumento e irá devolver depois uma lista com a sequência, em que o tamanho da lista deverá ser igual ao valor especificado no argumento.

Começamos por enunciar os seguintes testes:

- Para valores de  $n$  inferiores a 0 a função deverá devolver uma lista vazia.
- Para  $n$  igual a 0 a função deverá devolver `[0]`.
- Para  $n$  igual a 1 a função deverá devolver `[0, 1]`.
- Para  $n$  igual a 2 a função deverá devolver `[0, 1, 1]`.
- Para  $n$  igual a 5 a função deverá devolver `[0, 1, 1, 2, 3, 5]`.
- Para qualquer  $n$  a função deverá devolver uma lista com  $n + 1$  elementos.

Após a definição destes testes podemos iniciar o desenvolvimento, tratando cada um dos testes de forma isolada. Isto envolve implementar o código, validar o teste e re-arranjar o código de forma a continuar consistente. Neste caso, o primeiro teste requer que a função tenha a seguinte estrutura:

---

```
def fibonacci(n):
    res = []
    if n < 0:
        return res
```

---

Como é óbvio, esta implementação não pode ser considerada correta pois caso `n` seja superior ou igual a 0 a função não irá devolver nada. No entanto, do ponto de vista do primeiro teste ele será concretizado com sucesso. A implementação pode então continuar, procurando satisfazer os testes seguintes, **um de cada vez**, até que todos sejam bem sucedidos.

A execução dos testes pode ser feita de forma bastante simples, recorrendo a sequências de instruções que validam o valor devolvido pelas funções. O primeiro teste poderia ser implementado fazendo:

---

```
def teste1():
    if fibonacci(0) == [0] and fibonacci(-1) == []:
        print("Teste OK")
    else:
        print("Teste Falhou")
```

---

Esta função irá imprimir **Teste OK** ou **Teste Falhou** dependendo do valor devolvido pela função. No entanto, isto não é suficiente para, de uma forma sistemática, executar testes e avaliar a situação do desenvolvimento da aplicação. Por isso recomenda-se a utilização da ferramenta `pytest`.

Para usar esta ferramenta, os testes têm de ser codificados em funções com nomes começados por `test_`. Normalmente essas funções são definidas em ficheiros auxiliares com nomes começados por `test_` e com extensão `.py`. Por exemplo, o ficheiro `test_fibonacci.py` poderia começar por definir uma função para o primeiro teste que enunciámos:

---

```
# test_fibonacci.py
import pytest
from fibonacci import fibonacci

def test_inferior_1():
    print("Testa comportamento com n < 1")
    assert fibonacci(0) == [0]
    assert fibonacci(-1) == []
```

---

De notar que é necessário importar a função `fibonacci`, que deverá estar num ficheiro chamado `fibonacci.py`. A palavra reservada `assert` tem o significado de *afirmar* e usa-

se para verificar *asserções* ou seja, condições que se presume serem sempre verdadeiras. Num programa normal, se a condição avaliada num `assert` for falsa, é gerada uma exceção que interrompe o programa com uma mensagem de erro informativa. Pelo contrário, quando executada pelo `py.test`, uma asserção falsa é reportada como uma falha do teste, mas a verificação avança para os testes seguintes.

### Exercício 14.2

Implemente os testes que definimos acima no ficheiro `test_fibonacci.py` e verifique que executam através da ferramenta `py.test`. Depois de todos os testes estarem implementados, implemente progressivamente o código num ficheiro `fibonacci.py` de forma a cumprir cada um dos testes.

### Exercício 14.3

Defina testes unitários para seis funções que realizem as operações aritméticas de soma, subtração, multiplicação, divisão, resto da divisão inteira e raiz quadrada sobre valores reais.

Após a implementação dos testes, implemente as funções de forma a cumprirem os testes desenvolvidos. Tenha em consideração os valores aceitáveis (o domínio) para cada uma das operações.

Tenha igualmente em atenção que os valores reais possuem limite de precisão. Pode verificar isto se utilizar a linguagem Python para somar 1.3 com 1.6. Poderá ter de utilizar a função `round` para limitar a precisão e evitar estes erros de aproximação.

## 14.4 Testes Funcionais

Os testes unitários dedicam-se à verificação de componentes do software tão pequenos quanto possível. Mas uma aplicação é formada por muitos componentes que interagem de forma complexa e a correção dos componentes isolados não basta para garantir a correção da aplicação. Para verificar a funcionalidade de uma aplicação é necessário considerar aspectos de execução e de interface com o utilizador, tomando a aplicação como um todo. Esse é objetivo dos testes funcionais.

Os testes funcionais podem ser encarados da mesma forma que os unitários, mas focam-se sobre aspectos mais amplos da aplicação. Eles devem ser desenvolvidos de forma única para cada aplicação, visto que cada aplicação possui funcionalidades distintas. No caso dos exemplos tratados anteriormente, faz sentido avaliar os resultados debitados pelo programa (o seu *output*), quando se lhe fornece certos dados (o seu *input*).

A ferramenta `py.test` também pode ser utilizada para testes funcionais, pois é possível executar uma aplicação, capturar o que escreve para o ecrã e comparar o resultado obtido com o esperado. A chave para isto é o módulo `subprocess` e a classe `Popen`. Usando estas classes é possível executar uma aplicação da seguinte forma:

---

```
from subprocess import Popen
from subprocess import PIPE

proc = Popen("comando a executar", stdout=PIPE, shell=True)

return_code = proc.wait()
output = proc.stdout.read().decode('utf-8')
```

---

A linha `proc.stdout.read()` obtém tudo o que a aplicação escreve para o ecrã. Para se iterar sobre cada linha em tempo real seria possível fazer:

---

```
...
for line in iter(proc.stdout.readline, b''):
    print( line.decode('utf-8'), end=' '
```

---

#### Exercício 14.4

Implemente um programa que execute o comando `"ls -la "+sys.argv[1]` e imprima o seu resultado, mas descartando qualquer linha que contenha um termo fornecido no segundo argumento ao programa (`sys.argv[2]`). Pode verificar se uma linha contém um termo usando `if termo in line:`.

O exemplo anterior pode ser combinado de forma a ser parte integrante de um teste. Pode-se comparar a impressão da execução de um comando e o seu código de retorno,

bastando para isso que o código pertença a uma função com nome iniciado por `test_`.

### Exercício 14.5

Defina um conjunto de testes **funcionais** para as aplicações consideradas anteriormente (Fibonacci e Calculadora). Considere a introdução de argumentos inválidos (*Strings*), a total ausência de argumentos, ou a introdução de valores inválidos para operações específicas. Em cada caso a aplicação deverá devolver mensagens de erro específicas.

Implemente os testes usando o formato necessário pela ferramenta `py.test`.

**Não implemente qualquer funcionalidade na aplicação que vá de acordo aos testes.**

### Exercício 14.6

Ordene os testes e, um de cada vez, implemente as funcionalidades necessárias para a aplicação os passar com sucesso.

## 14.5 Depuração

Um programa pode apresentar uma miríade de erros que impedem o seu funcionamento correto. No contexto da linguagem *Python* podemos encontrar:

**Erros de sintaxe:** Encontrados pelo interpretador de *Python* quando converte o código fonte para instruções. Estes erros são detetados imediatamente após a tentativa de execução de uma aplicação, resultando numa mensagem de `SyntaxError: invalid syntax`. Parêntesis ou outro carácter em falta, indentação incorreta, ou erros na escrita das palavras reservadas levam a que seja produzido este erro.

**Erros de execução:** Encontrados pelo interpretador quando uma situação excepcional é encontrada durante a execução. São situações como uma divisão por zero, ou uma operação entre tipos incompatíveis, que ocorrem devido ao fluxo de execução particular, não sendo possível prever esta situação quando o ficheiro é carregado.

**Erros semânticos:** O programa executa sem detetar qualquer erro, mas o resultado não é o esperado. Os testes funcionais e unitários podem detetar estes erros.

O primeiro tipo de erros é detetado facilmente pelo interpretador, que os localiza e identifica. Os erros de execução também são detetados pelo interpretador, mas não é

detetada a sua causa. O terceiro tipo de erros apenas pode ser detetado com testes, mas mais uma vez, apenas se saberá que o erro existe dentro de uma unidade ou de uma funcionalidade, não se sabendo em concreto qual o problema.

Para estes casos existem mecanismos nas linguagens e ferramentas que permitem executar de forma interativa uma aplicação, sendo possível inspecionar cada ponto da execução. Estas ferramentas, denominadas por depuradores (*debuggers*), executam os programas fornecendo ao programador muito mais controlo sobre a sua execução. São exemplos o módulo `pdb`, o `pydbgr` ou o `ipdb`, todos eles disponíveis para instalação com `pip` ou `easy_install`. Alguns Integrated Development Environment (IDE) já possuem esta funcionalidade incluída, como é o caso do Eclipse<sup>2</sup>, do Microsoft Visual Studio<sup>3</sup> ou do PyCharm<sup>4</sup>, entre outros.

Considere o seguinte programa que, de uma forma simples (e incorreta) tenta verificar se um dado número é primo:

---

```
import sys

def main(argv):
    num = int(argv[1])
    for x in range( num//2 ):
        if num % x != 0:
            print("False")
    print("True")

main(sys.argv)
```

---

Este programa não apresenta nenhum erro sintático. No entanto apresenta um erro de execução e um semântico. Neste caso é simples de detetar por revisão do código, mas poderia não ser, servindo mesmo assim como exemplo para utilização de um depurador.

Para instalar o depurador `ipdb` poderá usar o gestor de pacotes da sua distribuição Linux, no caso da VM da disciplina (Debian) execute:

---

```
$ apt install python3-ipdb
```

---

Alternativamente, poderá usar o gestor de pacotes do Python3, executando:

---

```
$ pip3 install --user ipdb
```

---

<sup>2</sup><https://eclipse.org/>

<sup>3</sup><http://www.visualstudio.com/>

<sup>4</sup><https://www.jetbrains.com/pycharm/>

Depois, pode-se iniciar o depurador para um dado programa executando:

---

```
$ python3 -m ipdb prime.py 10
```

---

Se tiver problemas a instalar este depurador, pode utilizar o depurador integrado **pdb**.

De notar que os argumentos, neste caso 10, são passados para o programa tal como se não estivesse a ser depurado.

A partir deste momento o programa é carregado e está pronto a ser depurado.

---

```
> prime.py(1)<module>()
----> 1 import sys
      2
      3 def main(argv):
ipdb>
```

---

O depurador indica a instrução que está pronta para ser executada, apresenta um prompt e aguarda por comandos.

Alguns comandos bastante úteis que se podem utilizar são os seguintes:

**continue:** Continua a execução;

**run:** Volta a executar o programa;

**break <nome-da-funcao>:** Define que a execução deve ser parada na função indicada;

**break <numero-da-linha>:** Define que a execução deve ser parada na linha de código indicada;

**print( <nome-da-variavel> ):** Mostra o valor de uma variável;

**list:** Lista o código fonte;

**next:** Executa a próxima instrução;

**step:** Executa a próxima instrução.

Se a instrução chamar uma função, a execução interativa entra dentro da função, de forma a que a próxima instrução a executar interactivamente seja a primeira instrução da função.

Neste caso interessa definir um *breakpoint* na linha 4 e depois executar cada instrução passo a passo até encontrar um erro.

O resultado seria:

---

```
> prime.py(1)<module>()
----> 1 import sys
      2
      3 def main(argv):
          ipdb> break 4
Breakpoint 1 at prime.py:4
ipdb> continue
> prime.py(4)main()
      3 def main(argv):
1---> 4     num = int(argv[1])
      5     for x in range( num//2 ):
          ipdb> n
> prime.py(5)main()
1     4     num = int(argv[1])
----> 5     for x in range( num//2 ):
      6         if num % x != 0:
          ipdb> n
> prime.py(6)main()
      5     for x in range( num//2 ):
----> 6         if num % x != 0:
      7             print("False")
          ipdb> n
ZeroDivisionError: integer division or modulo by zero
> prime.py(6)main()
      5     for x in range( num//2 ):
----> 6         if num % x != 0:
      7             print("False")
          ipdb> print(num)
10
ipdb> print(x)
0
ipdb>
```

---

Após a ocorrência do erro, pode-se verificar em que linha aconteceu e inspecionar o valor das variáveis atuais. Neste caso, verifica-se que a variável `x` é igual a 0, o que gera uma divisão por 0.

Sem o depurador seria possível obter informação do erro, mas não poderíamos inspecionar a memória no momento do erro. Um aspeto interessante do depurador é que ele permite

inspecionar a memória de uma aplicação quando são encontrados problemas, mesmo que não tenhamos definido um *breakpoint* previamente.

### Exercício 14.7

Volte a executar o programa no depurador e não defina nenhum *breakpoint* nem execute o programa de forma interativa. Simplesmente forneça a instrução de **continue**. Repare que o programa é interrompido na ocorrência de um erro.

Verifique que pode inspecionar o estado de todas as variáveis e pode mesmo alterar o seu conteúdo através de sintaxe *Python* (p. ex. `x = 1`).

### Exercício 14.8

Corrija o erro em causa e voltando a utilizar o depurador, verifique a correta execução do programa. Caso encontre um problema, corrija-o e volte a iniciar uma sessão de depuração.

### Exercício 14.9

Crie e implemente testes funcionais de forma a validar o programa acima referido. Ele deverá aceitar um argumento inteiro superior a 0 na linha de comandos e determinar se é primo ou não. O resultado é apresentado através da impressão das palavras **False** ou **True** e definição do código de execução: 1 para primo, 0 para não primo.

## 14.6 Para Aprofundar

### Exercício 14.10

Construa testes unitários para os programas do segundo guião de Programação 2. Utilize para isso a linguagem *Java*. Em *Java* existe a classe *JUnit* que pode facilitar a sua especificação e que possui funcionalidades semelhantes à ferramenta **py.test**. Pode encontrar exemplos de como utilizar a class *Junit* no endereço:  
<https://github.com/junit-team/junit/wiki/Getting-started>

### **Exercício 14.11**

Construa testes funcionais para os programas do segundo guião de Programação 2 e valide a sua execução. Utilize para isso a linguagem *Java* ou *Python*.

## Glossário

**IDE** Integrated Development Environment

**TDD** Test Driven Development

# Comunicação entre aplicações

## Objetivos:

- Comunicação entre aplicações
- Sockets UDP
- Sockets TCP
- Acesso Assíncrono

## 15.1 Introdução

As aplicações informáticas realizam trabalho sobre dados que lhes são fornecidos, o que tem sido feito através de argumentos, introdução de informação pelo teclado, ou de ficheiros. No entanto também é possível e extremamente útil as aplicações trocarem informação diretamente entre si. Um exemplo muito comum é a consulta de páginas na Internet. Segundo a indicação do utilizador, um navegador Web irá consultar os diversos servidores de forma a obter páginas. Tanto o navegador, como o(s) servidores Web são aplicações que possuem a capacidade de trocarem informação, o que é feito através de mensagens que são transmitidas através de uma rede. A construção e transmissão das mensagens foi já abordado anteriormente. Este guião aborda a geração, envio e recepção das mensagens pelas aplicações.

## 15.2 Conceitos de comunicação

### 15.2.1 Modelo Cliente-Servidor

Nas comunicações entre aplicações é usual distinguir-se dois tipos de intervenientes: o cliente e o servidor. O cliente é aquele que inicia a comunicação e faz um pedido, enquanto o servidor é aquele que aceita um pedido e realiza uma ação, podendo emitir uma resposta. Por exemplo, o navegador Web é um cliente que emite pedidos enquanto

os servidores Web são aqueles que aceitam pedidos dos clientes, fornecendo depois as páginas como resposta. A Figura 15.1 representa uma simples interacção entre vários clientes e um servidor.

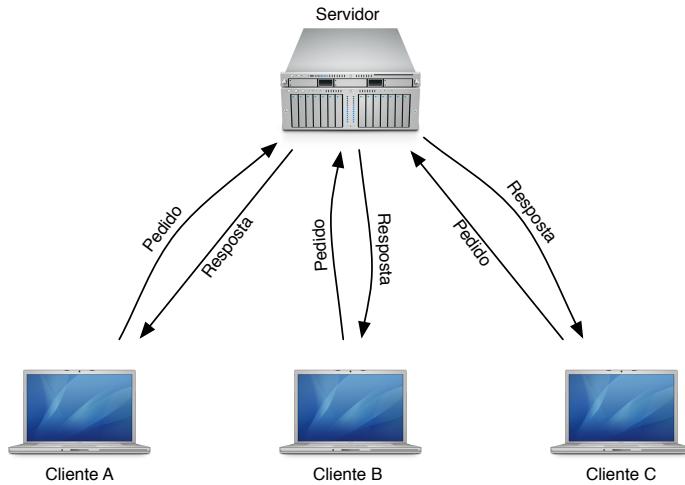


Figura 15.1: Modelo Cliente-Servidor

Esta separação de funções é importante porque implica que as aplicações do cliente e do servidor são distintas: o fluxo lógico (algoritmo) do cliente é diferente do fluxo lógico do servidor. Considerando o mesmo exemplo, os servidores Web têm de fornecer páginas e outros ficheiros, enquanto os clientes têm de processar HyperText Markup Language (HTML)[1], JavaScript (JS)[2], Cascading Style Sheets (CSS)[3], construir a página e interagir com o cliente. Outra característica deste modelo é que vários clientes se podem ligar a um único servidor.

Existem aplicações, como as utilizadas nas redes Peer to Peer (P2P) que são construídas de forma a funcionarem ao mesmo tempo como cliente e servidor. Isto não vai contra o dito anteriormente relativamente a existirem dois tipos de aplicações, pois para suportar esta modalidade, estas aplicações têm de possuir código para ambas as funções. Quando um aplicação P2P comunica com outra, durante esta transação essa age como servidor e a outra como cliente.

### 15.2.2 Sockets

As aplicações na consola podem interagir com o utilizador através de três dispositivos básicos: `stdin`, `stdout` e `stderr`. Estes dispositivos têm sido extensivamente utilizados quando se lê texto do teclado ou se escreve para a consola. As aplicações também podem criar representações internas de ficheiros e diretórios, que depois utilizam para ler,

escrever, criar ou apagar estes elementos. Para permitir a comunicação entre processos, ou Inter Process Communication (IPC), existem outros mecanismos como o *Socket*. O termo inglês *Socket* designa uma tomada onde se pode ligar uma ficha, tal como uma tomada de corrente eléctrica ou uma tomada de rede ethernet. No âmbito do software, um *Socket* é um mecanismo pelo qual as aplicações podem criar tomadas virtuais onde outras aplicações se podem “ligar”. Uma aplicação pode criar quantos *Sockets* quiser, tal como uma casa pode ter quantas tomadas forem necessárias. A Figura 15.2 demonstra uma aplicação com vários pontos de comunicação, sendo que alguns são *Sockets*.

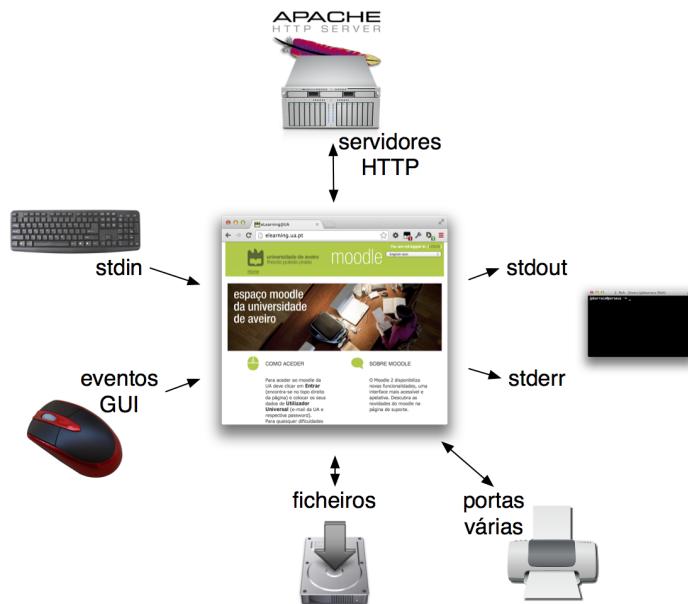


Figura 15.2: Vários pontos de comunicação com um navegador Web.

Tal como uma tomada possui um dada especificação, quanto à sua forma e contactos, também um *Socket* possui algumas características distintivas, das quais se destacam a *família* e o *tipo*. Um *Socket* de uma dada família e tipo só aceita ligações com essas características e não com outras. Existem várias famílias de *Socket*. As principais são:

**AF\_UNIX:** Indica um *Socket* para ser utilizado em comunicações entre aplicações locais (na mesma máquina).

**AF\_INET:** Indica um *Socket* para ser utilizado sobre endereços Internet Protocol v4 (IPv4)[4]. Pode ser utilizado para comunicações entre aplicações locais ou remotas.

**AF\_INET6:** Indica um *Socket* para ser utilizado sobre endereços Internet Protocol v6 (IPv6)[5]. Pode ser utilizado para comunicações entre aplicações locais ou remotas.

Neste caso, um *Socket* da família **AF\_INET** pode ser utilizado para trocar mensagens com aplicações que estejam em sistemas com um IPv4 disponível, mas não permite trocar mensagens com aplicações que estejam em sistemas que só possuam IPv6.

Além da família, o tipo do *Socket* irá indicar como as mensagens devem ser encapsuladas antes de serem enviadas. Há dois tipos mais relevantes:

**SOCK\_DGRAM:** As comunicações deverão utilizar o protocolo User Datagram Protocol (UDP)[6]. Com este tipo de *Socket*, é possível que as mensagens se percam ou cheguem fora de ordem.

**SOCK\_STREAM:** As comunicações deverão utilizar o protocolo Transmission Control Protocol (TCP)[7]. Com este tipo de *Socket*, em caso de perda, o protocolo TCP irá retransmitir as mensagens. Este também garante que a ordem de envio é mantida à chegada.

### 15.3 Sockets não orientados à ligação (UDP)

Um tipo de *Sockets* permite o envio de mensagens entre aplicações sem que estas estabeleçam uma ligação explícita persistente. Entre muitos outros cenários, são indispensáveis para envio de mensagens de *broadcast*, para sistemas com baixas capacidades computacionais, ou para comunicações com restrições de tempo real. Este tipo de *Sockets* são normalmente utilizados nas redes IP TeleVision (IPTV) que temos em casa, nas aplicações Voice Over IP (VoIP) como o *Skype*, ou no processo de atribuição de endereços dinâmicos (Dynamic Host Configuration Protocol (DHCP)[8]). Este tipo de *Socket* utiliza o protocolo UDP para o transporte das mensagens.

Um *Socket*, seja ele orientado à ligação ou não, não realiza comunicações de forma imediata só porque existe. Comunicar com um *Socket* requer uma sequência de acções, nomeadamente:

**Criação:** Um *Socket* é criado, definindo-se a sua família e tipo. Utiliza-se para isso a instrução **socket**.

**Nomeação:** É necessário dar um nome ao *Socket*, usando-se a primitiva **bind**. O nome permite identificar **unicamente** um dado *Socket* num dado sistema que pode ter inúmeros *Sockets* de várias aplicações. As aplicações trocam informação a partir de e para um *Socket* em particular. O nome é composto por um caminho, normalmente um endereço IPv4 e um porto (ou porta).

**Enviar Informação:** A aplicação emissora usa a acção **send** para enviar informação para o *Socket*. O sistema de destino armazena essa informação numa memória associada ao *Socket*, temporariamente.

**Receber Informação:** A aplicação recetora usa a acção **receive** para recolher a informação recebida e armazenada pelo *Socket*.

**Fechar:** Tal como um ficheiro, o *Socket* deve ser fechado quando a comunicação termina.

A Figura 15.3 representa uma utilização destas primitivas numa comunicação entre uma aplicação cliente e uma aplicação servidora. Note que não há nenhuma diferença formal entre um *Socket* no servidor ou no cliente, mas a lógica das aplicações é diferenciada. Uma espera por pedidos, a outra efetua pedidos.

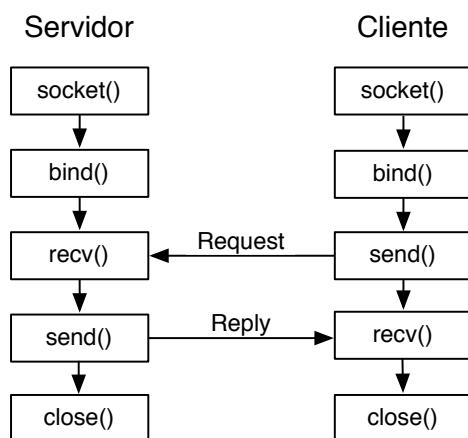


Figura 15.3: Sequência de primitivas utilizadas num *Socket UDP*.

Em *Python* é possível utilizar *Sockets* através da inclusão do módulo **socket**. Assim, um *Socket* pode ser criado e nomeado através das seguintes instruções:

---

```
import socket

def main():
    udp_s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    udp_s.bind( ("127.0.0.1", 1234) )

main()
```

---

Neste exemplo o *Socket* é criado de forma a comunicar na porta **1234** para aplicações no próprio computador (**127.0.0.1**). Este endereço determina em que interface de rede o *Socket* vai comunicar, sendo que o valor especial **0.0.0.0** indica que irá comunicar através de todos os interfaces. Se o valor da porta for igual a 0, o sistema operativo irá escolher uma porta aleatória.

**Atenção:** Num dado sistema, não é possível existirem dois *Sockets* com o mesmo nome (endereço e porta)!

Para a troca de informação, é agora necessário receber e enviar informação. O exemplo seguinte espera por uma mensagem e responde enviando a mesma mensagem em maiúsculas. De notar que se utiliza o método **recvfrom** que possui como parâmetro o número máximo de octetos a receber e devolve 2 valores: uma string com os dados recebidos e o endereço do socket que os enviou.

---

```
...
def main():
    ...
    while 1:
        b_data, addr = udp_s.recvfrom(4096)
        udp_s.sendto(b_data.upper(), addr)
    udp_s.close()
...
```

---

### Exercício 15.1

Utilizando os exemplos anteriores, implemente um servidor de mensagens UDP.  
Pode testar o servidor utilizando o comando **nc -u localhost 1234**.

O cliente para comunicar com esta aplicação seria programado de maneira muito semelhante. Apenas teria as instruções de envio e recepção por ordem inversa: primeiro envia e depois recebe a resposta. O trecho seguinte demonstra um cliente que lê uma frase do teclado, envia-a para o servidor, espera uma resposta e imprime-a.

```
...
def main():
    ...
    udp_s.bind(("127.0.0.1", 0))
    server_addr = ("127.0.0.1", 1234)
    while 1:
        str_data = input("<-:")
        b_data = str_data.encode("utf-8")
        udp_s.sendto(b_data, server_addr)
        # ---
        b_data, addr = udp_s.recvfrom(4096)
        str_data = b_data.decode("utf-8")
        print("->: %s \n" % str_data)

    udp_s.close()
...

```

### Exercício 15.2

Utilizando os exemplos fornecidos implemente um cliente que permita a troca de mensagens. Tenha em consideração que, por residirem no mesmo sistema, o cliente não pode criar um *Socket* na mesma porta que o servidor.

### Exercício 15.3

Coordene com o grupo ao seu lado de forma a executarem um servidor com um *Socket* no endereço 0.0.0.0. Deverá conseguir enviar mensagens para este servidor se o seu cliente se ligar ao endereço do computador onde reside o servidor.

## 15.4 Sockets orientados à ligação (TCP)

Além das primitivas já vistas, um *Socket* do tipo **SOCK\_STREAM** necessita de mais três. Isto deve-se ao facto dos *Sockets* deste tipo serem orientados à ligação, existindo a necessidade de se estabelecer uma ligação (ou sessão) entre as duas aplicações antes da transmissão de informação.

**Aceitar Ligações:** Um *Socket* do servidor irá ser definido como aceitando ligações de novos clientes, realizando-se esta ação através da instrução **listen**.

**Estabelecer ligação:** O cliente necessita de se ligar ao servidor antes de se poder trocar informação, usando-se para isso a instrução **connect**.

**Aceitar Clientes:** O servidor, após receber o pedido de ligação, pode aceitá-lo através da instrução `accept`.

Estas primitivas são utilizadas da forma descrita na Figura 15.4. Como pode ser visto, neste tipo de Sockets, **existe** uma declaração formal de que *Socket* é servidor ou cliente e a lógica das aplicações é também diferenciada. Uma aplicação espera por pedidos, outra efetua pedidos. As primitivas adicionais, necessárias ao estabelecimento da sessão, estão realçadas na figura.

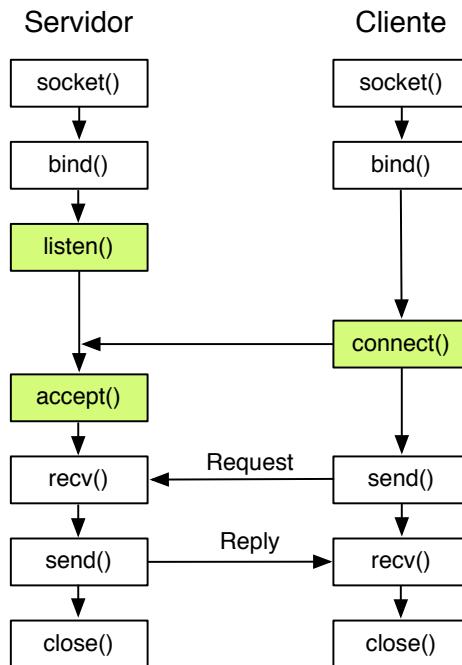


Figura 15.4: Sequência de primitivas utilizadas num *Socket* TCP.

Utilizando *Python* é possível utilizar Sockets orientados à ligação de uma forma semelhante aos anteriores, mas agora considerando a necessidade de estabelecimento da ligação antes da troca de informação. Assim, um *Socket* TCP pode ser criado e nomeado através das seguintes instruções:

---

```
# encoding=utf-8
import socket

def main():
    tcp_s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind( ("127.0.0.1", 1234) )

main()
```

---

Para o estabelecimento da sessão é agora necessário que o servidor defina o *Socket* como aceitando ligações e que espere por novos clientes, aceitando-os depois. Note que quando se aceita uma ligação de um novo cliente é criado um novo **Socket**, que é utilizado para identificar a ligação entre o servidor e o cliente.

---

```
...
def main():
    ...
    # máximo de 1 cliente à espera de
    # aceitação
    tcp_s.listen(1)

    # esperar por novos clientes
    client_s, client_addr = tcp_s.accept()

    while 1:
        b_data = client_s.recv(4096)
        client_s.send(b_data.upper())

    client_s.close()
    tcp_s.close()
...
```

---

De notar que neste caso utiliza-se o *Socket* chamado **client\_s** para todas as futuras comunicações. Também se utilizam os métodos **recv** em vez de **recvfrom** e **send** em vez de **sendto**. Isto porque não é necessário saber de onde chegou ou para onde vai a informação: toda a informação que seja lida ou escrita no **client\_s** é trocada entre o servidor e o cliente específicos.

#### Exercício 15.4

Implemente um servidor TCP utilizando o exemplo fornecido. Pode testar o servidor utilizando o comando **telnet localhost 1234**, ou o comando **nc localhost 1234**.

O cliente será bastante semelhante ao caso dos Sockets não orientados à ligação, mas neste caso, é necessário estabelecer previamente a sessão.

---

```
...
def main():
    ...
    # Ligar ao servidor
    tcp_s.connect( "127.0.0.1", 1234 )
    while 1:
        str_data = input("Mensagem: ")
        b_data = str_data.encode("utf-8")
        tcp_s.send(b_data)
        # ---
        b_data = tcp_s.recv(4096)
        str_data = b_data.decode("utf-8")
        print(str_data)

    tcp_s.close()
...

```

---

Tal como acontece com a implementação do servidor, a troca de informação é feita através dos métodos `recv` e `send`, não existindo necessidade de especificar sempre qual o destino das mensagens, ou de obter informação relativa à sua origem.

### Exercício 15.5

Implemente um cliente que utilize Sockets orientados à ligação. Mais uma vez, tenha em atenção que os clientes devem utilizar portas aleatórias de forma a não colidirem com serviços existentes.

## 15.5 Servidor de Mensagens Instantâneas

De uma forma simples é possível implementar um servidor que actue como uma ferramenta de *chat*, permitindo a troca de mensagens entre utilizadores. Este exemplo é útil pois permite demonstrar com é possível interagir com múltiplos clientes e, no caso do cliente, ler de forma alternada de duas fontes de informação. De notar que o servidor e o cliente podem ser implementados utilizando qualquer um dos tipos de Socket abordados. De forma a simplificar a explicação, o exemplo irá focar-se no caso de comunicações através de UDP deixando-se a implementação com TCP para exercício.

A implementação do servidor é simples, bastando que possua uma lista de Sockets conhecidos e envie as mensagens recebidas de um Socket para todos.

---

```
...
def main():
    ...
    # Lista de sockets conhecidos
    addr_list = []

    while 1:
        b_data, addr = udp_s.recvfrom(4096)
        print(b_data.decode("utf-8"))

        # Adicionar o nome do socket à lista de sockets conhecidos
        if not addr in addr_list:
            addr_list.append(addr)

        # Enviar a mensagem para todos
        for dst_addr in addr_list:
            udp_s.sendto(b_data.upper(), dst_addr)
```

---

### Exercício 15.6

Implemente um servidor como o indicado no exemplo. Este deverá funcionar com os clientes UDP criados anteriormente. No entanto o cliente não irá suportar a nova lógica, pelo que não serão apresentadas as mensagens de todos os clientes.

Um cliente de *chat* também necessitaria de pequenas alterações de forma a permanentemente ouvir novos dados do teclado e do *Socket*. O teclado permite ao utilizador enviar mensagens, o *Socket* permite receber as mensagens dos outros clientes. Isto é possível através da utilização da instrução **select**, que basicamente permite ficar à escuta de informação de múltiplas fontes, indicando depois qual das fontes possui informação para ser consumida.

O método **select** aceita três parâmetros: a lista de *Sockets* (ou outros dispositivos) onde se espera por dados, a lista de *Sockets* onde recentemente foram escritos dados e se espera que estes sejam transmitidos e a lista de *Sockets* onde se querem receber notificações de exceções (p.ex, *Socket* fechado). Irá devolver igualmente três listas com os *Sockets* que tiveram os eventos respetivos. Aqui apenas nos interessa a primeira das listas, a que indica os *Sockets* com informação pronta a ser consumida.

---

```
import select
...
def main():
    ...
    while 1:
        rsocks = select.select([udp_s, sys.stdin, ], [], [])[0]

        for sock in rsocks:
            if sock == udp_s:
                # Informação recebida no socket
                b_data, addr = udp_s.recvfrom(4096)
                sys.stdout.write("%s\n" % b_data.decode("utf-8"))
            elif sock == sys.stdin:
                # Informação recebida do teclado
                str_data = sys.stdin.readline()
                udp_s.sendto(str_data.encode("utf-8"), server_addr)
        ...

```

---

### Exercício 15.7

Implemente um cliente de *chat* de forma a que possa dialogar com os outros utilizadores ligados ao mesmo servidor.

### Exercício 15.8

Implemente o mesmo cliente e servidor mas utilizando TCP. Neste caso tenha em consideração que o servidor irá possuir um *Socket* por cliente, armazenando estes *Sockets* na lista de clientes (e não o endereço e porta utilizados pelo cliente).

## 15.6 Para Aprofundar

### Exercício 15.9

Implemente um programa que dado um endereço HyperText Transfer Protocol (HTTP)[9] como primeiro argumento, e um nome de ficheiro como segundo argumento, crie uma ligação TCP e envie os cabeçalhos HTTP de forma a obter o recurso (ficheiro) indicado. A resposta deverá ser escrita para o ficheiro indicado no segundo argumento. (Este programa é um cliente HTTP simples.)

### **Exercício 15.10**

Implemente um programa que escute por mensagens no *Socket TCP* no formato:

```
GET /dir/fname
```

O programa deve responder com o conteúdo do ficheiro que se encontra no local indicado. Considere que os ficheiros pedidos encontram-se em diretórios e sub-diretórios de um diretório raiz específico. A título de exemplo, se o diretório raiz for **/tmp** e for pedido o ficheiro **/labi/aula/a.txt**, o programa deve fornecer o conteúdo do ficheiro **/tmp/labi/aula/a.txt**. (Este programa é um servidor HTTP simples.)

## Glossário

<b>CSS</b>	Cascading Style Sheets
<b>DHCP</b>	Dynamic Host Configuration Protocol
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	HyperText Transfer Protocol
<b>IPC</b>	Inter Process Communication
<b>IPTV</b>	IP TeleVision
<b>IPv4</b>	Internet Protocol v4
<b>IPv6</b>	Internet Protocol v6
<b>JS</b>	JavaScript
<b>P2P</b>	Peer to Peer
<b>TCP</b>	Transmission Control Protocol
<b>UDP</b>	User Datagram Protocol
<b>VoIP</b>	Voice Over IP

## Referências

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

- [2] ECMA International, *Standard ECMA-262 – ECMAScript Language Specification*, Padrão, dez. de 1999. URL: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.
- [3] W3C. (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, URL: <http://www.w3.org/TR/2011/REC-CSS2-20110607/>.
- [4] J. Postel, *Internet Protocol*, RFC 791 (Standard), Updated by RFC 1349, Internet Engineering Task Force, set. de 1981.
- [5] S. Deering e R. Hinden, *Internet Protocol, Version 6 (IPv6) Specification*, RFC 2460 (Draft Standard), Updated by RFCs 5095, 5722, 5871, Internet Engineering Task Force, dez. de 1998.
- [6] J. Postel, *User Datagram Protocol*, RFC 768 (Standard), Internet Engineering Task Force, ago. de 1980.
- [7] ———, *Transmission Control Protocol*, RFC 793 (Standard), Updated by RFCs 1122, 3168, 6093, Internet Engineering Task Force, set. de 1981.
- [8] R. Droms, *Dynamic Host Configuration Protocol*, RFC 2131 (Draft Standard), Updated by RFCs 3396, 4361, 5494, Internet Engineering Task Force, mar. de 1997.
- [9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 (Draft Standard), Updated by RFCs 2817, 5785, 6266, Internet Engineering Task Force, jun. de 1999.



# 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 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. Num tema anterior foi abordado o formato HyperText Markup Language (HTML)[1] que permite a representação de documentos Web. Também foi abordado o formato L<sup>A</sup>T<sub>E</sub>X 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 *sокеты*). 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.

## 16.2 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,<sup>1</sup> 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 caracteres, a escolha do separador deve ser feita com cuidado. Uma solução passa por delimitar com aspas todos os campos que potencialmente possam ter o carácter separador no seu interior. Utilizam-se também variações do formato CSV, tal com o formato Tabulation Separated Value (TSV). Neste último, o carácter de tabulação é utilizado para separar os diversos campos. É muito comum encontrarem-se ficheiros denominados CSV em que são utilizados os caracteres ;, : ou |. Deste modo, deve-se considerar o formato CSV

---

<sup>1</sup>Este formato é muito comum e descreve o número de segundos desde 1 de Janeiro de 1970

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)

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. 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 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. 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', '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 16.2

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 16.3

Considere o módulo `time` que permite acesso à hora atual e o módulo `psutil` que permite aceder a estatísticas do sistema,<sup>a</sup> 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. 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.

<sup>a</sup>Pode ser necessário instalar os módulos usando `pip3 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]. 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 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**, que recebe uma *String* em formato JSON e devolve uma lista ou dicionário.

#### Exercício 16.5

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},
    ]
}
```

---

## 16.4 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 estrutura 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 caracteres 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.

#### 16.4.1 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 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`), atributos (`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 raiz, também chamados os elementos filhos.

### Exercício 16.6

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 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, use o método **psutil.virtual\_memory()**, que indica a memória disponível no segundo elemento do tuplo devolvido (ver <https://github.com/giampaolo/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 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 raiz 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 raiz para utilizar num documento XML, cria depois um sub-elemento, adiciona-o à raiz 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 raiz 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.

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 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 caracteres inválidos ou em falta (ex, falta de um carácter */*), 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 **playlist.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* relativo ao formato XSPF.

---

```

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, o *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 com o nome da cidade e imprima os dados observados.

### **Exercício 16.13**

Considere o ficheiro disponível em <http://services.web.ua.pt/parques/parques>, contendo os dados relativos à capacidade dos parques da Universidade de Aveiro. Implemente 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 qual a hora da manhã a partir da qual os parques enchem?

## **Glossário**

<b>CPU</b>	Central Processing Unit
<b>CSV</b>	Comma Separated Values
<b>HTML</b>	HyperText Markup Language
<b>JSON</b>	JavaScript Object Notation
<b>RAM</b>	Random Access Memory
<b>TSV</b>	Tabulation Separated Value
<b>XML</b>	Extensible Markup Language
<b>XSPF</b>	XML Shareable Playlist Format

## **Referências**

- [1] W3C. (1999). HTML 4.01 Specification, 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.



# Aplicações e Serviços Web

## Objetivos:

- Servidores Web
- Serviços Web

### 17.1 Introdução

A World Wide Web (WWW) ou *Web* como é hoje popularmente conhecida, teve a sua génesis em 1990 no Conseil Européen pour la Recherche Nucléaire (CERN) pelas mãos de Tim Berners-Lee. Inicialmente, a *Web* pretendia ser um sistema de hiper-texto que permitisse aos cientistas seguir rapidamente as referências num documento, evitando o processo tedioso de apontar e pesquisar referências. A *Web* pretendia na altura ser um repositório de informação estruturado em torno de um grafo (daí a *Web*), em que o utilizador pudesse seguir qualquer percurso entre os diversos documentos interligados pelas suas referências.

A *Web* assenta em 3 tecnologias, já tratadas nesta disciplina:

- Um sistema global de identificadores únicos/uniformes (URL, URI).
- Uma linguagem de representação de informação (HTML).
- Um protocolo de comunicação cliente/servidor (HTTP).

Com base nestas tecnologias, podemos não só transferir ficheiros estáticos entre um servidor e um cliente equipado com um *Web browser*, como também podemos construir documentos de forma dinâmica a partir de dados recebidos ou disponíveis no servidor num determinado momento. Um bom exemplo deste último caso são os serviços meteorológicos que processam dados de observação e produzem documentos JavaScript Object Notation

(JSON)[1] e Extensible Markup Language (XML) com as observações e previsões, para consumo por humanos ou outras máquinas. Este guião irá guiar o desenvolvimento de uma aplicação Web dinâmica com base na linguagem de programação *Python* e no formato de documentos JSON.

## 17.2 Servidores Web

### 17.2.1 Servidores Aplicacionais

Um servidor Web é uma aplicação de software que permite a comunicação entre dois equipamentos através do protocolo HTTP. A função inicial de um servidor Web era a de fornecer documentos armazenados em disco em formato HyperText Markup Language (HTML)[2] a um cliente remoto equipado com um Web browser.

Esta simples tarefa desde cedo demonstrou-se demasiado restritiva, uma vez que frequentemente era necessário condicionar os dados nos documentos HTML a vários fatores, tais como: a identidade do utilizador, a sua localização, a sua língua nativa, etc.

Servidores aplicacionais como o *Glassfish*, *JBoss*, *.NET* permitem ao programador ultrapassar muitas destas dificuldades ao incorporarem em si próprios código desenvolvido por programadores externos. Não estamos mais na situação de o servidor fornecer um ficheiro estático, mas na de o próprio programa incluir o servidor Web e poder fornecer conteúdos gerados de forma programática.

Neste capítulo, vamos abordar um servidor aplicacional específico para *Python*.

O *CherryPy* é um servidor aplicacional simples mas poderoso, usado tanto para pequenas aplicações como para grandes serviços (ex.: *Hulu*, *Netflix*). O *CherryPy* pode ser usado sozinho (*stand-alone*) ou através de um servidor Web tradicional via interfaces Web Server Gateway Interface (WSGI). Nesta disciplina, vamos usar o *CherryPy* apenas como servidor *stand-alone*.

Para instalar o *CherryPy* pode recorrer ao gestor de pacotes da sua distribuição Linux ou ao **pip**.

No ubuntu pode executar:

---

```
sudo apt-get install python-cherrypy3
```

---

Em alternativa pode executar:

---

```
sudo pip3 install CherryPy
```

---

ou:

---

```
sudo pip3 install --upgrade CherryPy
```

---

**É altamente aconselhado que se instale o *CherryPy* via o comando *pip* pois a versão é mais recente.**

O *CherryPy* é composto por 8 módulos:

**CherryPy.engine** Controla início e o fim dos processos assim como o processamento de eventos.

**CherryPy.server** Configura e controla a WSGI ou servidor HTTP.

**CherryPy.tools** Conjunto de ferramentas ortogonais para processamento de um pedido HTTP.

**CherryPy.dispatch** Conjunto de *dispatchers* que permitem controlar o encaminhamento de pedidos para os *handlers*.

**CherryPy.config** Determina o comportamento da aplicação.

**CherryPy.tree** A árvore de objetos percorrida pela maioria dos *dispatchers*.

**CherryPy.request** O objeto que representa o pedido HTTP.

**CherryPy.response** O objeto que representa a resposta HTTP.

Comecemos por criar uma aplicação semelhante ao script Common Gateway Interface

(CGI) anterior.

### Exercício 17.1

Crie no seu próprio computador o seguinte ficheiro.

```
import cherrypy

class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        return "Hello World!"

cherrypy.tree.mount(HelloWorld(), "/")
cherrypy.server.start()
```

Se possuir a última versão do *CherryPy* as duas linhas finais do ficheiro anterior podem ser substituídas por:

```
cherrypy.quickstart(HelloWorld())
```

Sendo que neste caso a aplicação é lançada automaticamente quando existirem alterações ao ficheiro e permite que seja terminada usando **CTRL-C**.

No seu Web browser aceda à aplicação usando o endereço <http://localhost:8080/>.

Revendo cada linha do exercício anterior, começamos por identificar a importação do módulo *CherryPy*. De seguida temos a declaração de uma classe *HelloWorld*. Esta classe é composta por um método chamado *index* que devolve uma *String*. O *decorador* **@cherrypy.expose** determina que o método *index* deverá ser exposto ao cliente Web. Por fim, o módulo *CherryPy* cria um objeto da classe *HelloWorld* e inicia um servidor com ele.

Quando um cliente Web acede ao servidor aplicacional *CherryPy*, este procura por um objeto e método que possa atender ao pedido do cliente. Neste exemplo básico, existe apenas um objeto e método que irá servir ao cliente a *String* "Hello World".

O *CherryPy* disponibiliza através do **CherryPy.request.headers** as variáveis enviadas

pelo cliente ao servidor.

### Exercício 17.2

Altere o programa anterior para mostrar o nome do servidor ao qual o cliente fez um pedido HTTP

```
...  
host = cherrypy.request.headers["Host"]  
return "You have successfully reached " + host
```

Qualquer objeto associado ao objeto raiz é acessível através do sistema interno de mapeamento *URL*-para-objeto. Ou seja, definindo funções que implementem uma lógica, é possível expor essas funções, de forma quase automática, através de um Uniform Resource Locator (*URL*)[3], acessível por um browser.

No entanto, tal não significa que um objeto esteja exposto na *Web*. É necessário que o

objeto seja exposto explicitamente como visto anteriormente.

### Exercício 17.3

Crie um novo programa com o seguinte conteúdo

```
import cherrypy

class Node(object):
    @cherrypy.expose
    def index(self):
        return "Eu sou o índice do Node (Node.index)"

    @cherrypy.expose
    def page(self):
        return "You sou um método do Node (Node.page)"

class Root(object):
    def __init__(self):
        self.node = Node()

    @cherrypy.expose
    def index(self):
        return "Eu sou o índice do Root (Root.index)"

    @cherrypy.expose
    def page(self):
        return "Eu sou um método do Root (Root.page)"

if __name__ == "__main__":
    cherrypy.tree.mount(Root(), "/")
    cherrypy.server.start()
```

Aceda a cada um dos recursos a partir do seu navegador Web (`/page`, `/node/`). Pode alterar as mensagens devolvidas de forma a verificar que o conteúdo é dinâmico. Em alternativa, pode usar o módulo `psutil` para devolver estatísticas do sistema.

Mais uma vez, é importante reter alguns aspetos do exercício anterior. O método `index`

serve os conteúdos na raiz do URL (/) e cada método tem que ser exposto individualmente.

#### Exercício 17.4

Acrescente agora uma nova classe `HTMLDocument` que devolva o conteúdo de um ficheiro HTML lido do disco.

Ou seja, o programa irá abrir um ficheiro existente (`open`), ler o seu conteúdo (`read`) e devolver os dados lidos (`return`).

### 17.2.2 Formulário HTML

O protocolo HTTP define dois métodos principais para a troca de informação entre cliente e servidor: os métodos `GET` e `POST`. O método `GET` já foi extensivamente usado nos capítulos e secções anteriores, e permite ao cliente Web solicitar um documento que resida no servidor Web. Por sua vez o método `POST` permite enviar informação do cliente Web para o servidor Web. É geralmente usado para enviar ao servidor um ficheiro ou um formulário HTML preenchido.

#### Exercício 17.5

Crie uma página HTML com o código para formulário seguinte:

```
<form action="actions/doLogin" method="post">
    <p>Username</p>
    <input type="text" name="username" value="" size="15" maxlength="40"/>
    <p>Password</p>
    <input type="password" name="password" value="" size="10" maxlength="40"/>
    <p><input type="submit" value="Login"/></p>
    <p><input type="reset" value="Clear"/></p>
</form>
```

Não esquecer de completar a página com o código HTML apropriado.

Crie um novo método na sua aplicação:

```
@cherrypy.expose
def form(self):
    cherrypy.response.headers["Content-Type"] = "text/html"
    return open("formulario.html", "r").read()
```

No entanto, a submissão do formulário de *login* necessita ainda da implementação de mais um método.

### Exercício 17.6

Crie um novo objeto (*actions*) e método na sua aplicação. Não se esqueça de associar o novo objeto à Raiz.

```
class Actions(object):
    @cherrypy.expose
    def doLogin(self, username=None, password=None):
        return "TODO: verificar as credenciais do utilizador " + username
```

Abra o formulário através do endereço `http://localhost:8080/form/`, preencha-o e submeta.

Importa referir que os argumentos *username* e *password* chegam até à aplicação Web através de um mapeamento direto do nome das variáveis do formulário HTML para os argumentos do método **doLogin** (também mapeado diretamente).

## 17.3 Serviços Web

Na secção anterior viu-se como um cliente *Web* pode interagir com uma aplicação *Web* alojada no servidor. Nesta secção irá abordar-se como duas aplicações podem interagir entre si através do protocolo HTTP.

O primeiro desafio que se coloca é como escrever uma aplicação *Python* capaz de aceder a uma página *Web* via o protocolo HTTP. Para tal vamos fazer uso da biblioteca **requests**, cuja documentação completa encontra-se disponível em <http://docs.python-requests.org/en/master/>.

A biblioteca **requests** permite-nos aceder a uma página *Web* de forma muito semelhante à que utilizamos em *Python* para aceder a um ficheiro.

```
import requests

f = requests.get("http://www.python.org")
print(f.status_code)
```

## Exercício 17.7

Faça um pedido **GET** ao endereço <http://www.ua.pt>.

A sua aplicação deverá ler por completo o conteúdo da página da Universidade de Aveiro.

Imprima para a consola dados relevantes como os cabeçalhos da resposta ou o tamanho da página.

Utilizando o módulo **time** pode determinar qual é o tempo necessário para obter a página. Pode igualmente testar com outros ficheiros maiores, como por exemplo os disponíveis na página do *kernel Linux*.

O uso directo do método **get** permite-nos obter o conteúdo de um recurso HTTP através do método **GET**. No entanto, se pretendermos enviar algum conteúdo para uma aplicação Web, é necessário usar o método **POST** como vimos anteriormente.

O método **POST** possibilita o envio de informação codificada no corpo do pedido **POST**. A codificação dos dados segue um de dois *standards* definidos pelo World Wide Web Consortium (W3C), o **application/x-www-form-urlencoded** e o **multipart/form-data**.

O primeiro formato é o usado por omissão e permite o envio de informação trivial como variáveis não muito extensas. O segundo é apropriado para o envio de variáveis mais extensas assim como de ficheiros.

O módulo utilizado realiza esta formatação por defeito, enviando um dicionário qualquer que seja fornecido.

```
import requests

url = ...
values = {"nome": "Ana", "idade": 20}
r = requests.post(url, data=values)
print(r.status_code)
```

## Exercício 17.8

Fazendo uso da aplicação Web desenvolvida anteriormente, que continha um formulário, implemente uma aplicação capaz de fazer login.

Os exercícios anteriores demonstraram como criar uma aplicação Web capaz de interagir com um cliente (*Web browser*), mas a sua utilidade pode ser transposta para a comunicação entre duas aplicações.

### Exercício 17.9

O *OpenStreetMaps* dispõe de uma Application Programming Interface (API) que permite converter um endereço em coordenadas (latitude e longitude). Neste exercício deverá usar a API do OpenStreetMaps com base no seguinte código para encontrar as coordenadas de qualquer cidade passada ao seu programa através de um argumento de linha de comando.

```
serviceurl = "https://nominatim.openstreetmap.org/search.php?format=json&q=%s" % address
r = requests.get(serviceurl)
```

Verifique o método `json()` do resultado do pedido. Como o pedido indica que o formato deverá ser JSON, a resposta está disponível nesse método.

Imprima as coordenadas e a restante informação obtida

## 17.4 Conteúdos Estáticos

Nos exercícios anteriores permitimos ao nosso servidor aplicacional servir uma página HTML com o conteúdo de um ficheiro usando um método manual. Repare que o método não é escalável, pois é necessário abrir, ler e devolver todos os ficheiros necessário.

Uma alternativa é definir regras para servir ficheiros individuais, o que é apresentado no exemplo que se segue:

```
import os
import cherrypy

PATH = os.path.abspath(os.path.dirname(__file__))

...
conf = {
    "/omeuficheiro": {
        "tools.staticfile.on": True,
        "tools.staticfile.filename": os.path.join(PATH, "omeuficheiro.html")
    }
}
```

```
        }
cherrypy.tree.mount(Root(), '/', config=conf)
cherrypy.server.start()
```

---

### Exercício 17.10

Crie um programa que devolva um ficheiro, mas que o faça através de uma configuração do próprio servidor.

---

Podemos também automatizar este processo indicando ao **Cherrypy** que todos os conteúdos presentes num determinado diretório são estáticos. O exemplo que se segue considera que existe um diretório chamado **static**, localizado no mesmo diretório do programa **Python**, sendo que todo o seu conteúdo é estático, sendo servido automaticamente.

```
import os
import cherrypy

PATH = os.path.abspath(os.path.dirname(__file__))

...
conf = {
    "/static": {
        "tools.staticdir.on": True,
        "tools.staticdir.dir": os.path.join(PATH, "static")
    },
}

cherrypy.tree.mount(Root(), '/', config=conf)
cherrypy.server.start()
```

---

### Exercício 17.11

Implemente o exemplo anterior de forma a servir o mesmo ficheiro que usou anteriormente, mas de forma automática. Crie entradas adicionais na configuração de forma a ter ficheiros Cascading Style Sheets (CSS)[4], JavaScript (JS)[5] e ou imagens, também eles estáticos, cada um no seu diretório específico.

---

## 17.5 Para Aprofundar

### Exercício 17.12

Recupere o exercício de aprofundamento do guião anterior.

Construa uma aplicação Web que aceda ao ficheiro disponível em [http://www.ipma.pt/resources.www/internal.user/pw\\_hh\\_pt.xml](http://www.ipma.pt/resources.www/internal.user/pw_hh_pt.xml), contendo os dados meteorológicos observados nas principais cidades portuguesas.

A sua aplicação deverá receber o nome da cidade por método **POST**, pelo que deve construir um formulário com a lista de cidades possíveis usando uma *dropbox*.

À submissão do formulário deverá seguir-se a impressão dos dados da cidade indicada no formulário.

Para acesso ao serviço será necessário adicionar um cabeçalho (header) ao pedido:

```
...
url = "http://www.ipma.pt/resources.www/internal.user/pw_hh_pt.xml"
requests.get(url, headers={"referer": "http://www.ipma.pt/pt/index.html"})
...
```

## Glossário

<b>API</b>	Application Programming Interface
<b>CERN</b>	Conseil Européen pour la Recherche Nucléaire
<b>CGI</b>	Common Gateway Interface
<b>CSS</b>	Cascading Style Sheets
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	HyperText Transfer Protocol
<b>JS</b>	JavaScript
<b>JSON</b>	JavaScript Object Notation
<b>URL</b>	Uniform Resource Locator
<b>URI</b>	Uniform Resource Identifier
<b>W3C</b>	World Wide Web Consortium
<b>WWW</b>	World Wide Web

<b>WSGI</b>	Web Server Gateway Interface
<b>XML</b>	Extensible Markup Language

## Referências

- [1] E. T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, Internet Engineering Task Force, mar. de 2014.
- [2] W3C. (1999). HTML 4.01 Specification, URL: <http://www.w3.org/TR/1999/REC-html401-19991224/>.
- [3] M. Mealling e R. Denenberg, *Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations*, RFC 3305 (Informational), Internet Engineering Task Force, ago. de 2002.
- [4] W3C. (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, URL: <http://www.w3.org/TR/2011/REC-CSS2-20110607/>.
- [5] ECMA International, *Standard ECMA-262 – ECMAScript Language Specification*, Padrão, dez. de 1999. URL: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>.

# Bases de Dados

## Objetivos:

- Bases de dados relacionais
- SQL
- Acesso programático a bases de dados

### 18.1 Bases de Dados

As bases de dados são um componente fundamental em muitas aplicações, pois agem como repositórios onde é possível armazenar e consultar informação. É certo que isto pode ser feito utilizando ficheiros comuns, por exemplo é perfeitamente possível armazenar uma listagem de clientes num ficheiro de texto que depois é lido pela aplicação. No entanto,

- se a quantidade de dados é grande;
- se se pretende consultar os dados de forma não sequencial;
- se se pretende pesquisar ou ordenar dados segundo múltiplos critérios;
- se se pretende cruzar dados de várias naturezas;
- se ocasionalmente se acrescentam dados;
- se várias das anteriores são verdade;

então uma base de dados é uma solução muito mais poderosa e eficiente do que um ficheiro tradicional. As bases de dados, em particular as **relacionais**, possuem características de manipulação de informação muito interessantes, que facilitam a sua utilização face a outros métodos.

Considere uma lista de contactos de clientes em que é necessário armazenar o nome, endereço de correio electrónico e contacto telefónico. Um ficheiro simples poderia armazenar esta informação, por exemplo utilizando o formato Comma Separated Values (CSV)[1].

João, Manuel, Fonseca, jmf@gmail.com, 912345654  
 Pedro, Albuquerque, Silva, pedro23@gmail.com, 932454349  
 Maria, Carreira, Dinis, mariadi@ua.pt, 234958673  
 Catarina, Alexandra, Rodrigo, calexro@sapo.pt, 963343386

Se precisarmos de procurar um contacto de um cliente específico, temos de percorrer o ficheiro até o encontrar. Se precisarmos de alterar os dados de algum cliente, temos de reescrever o resto do ficheiro. Considerando que uma base de dados de clientes (ou outra) pode conter milhares ou milhões de entradas, usar um simples ficheiro de texto rapidamente se torna um problema.

Uma *base de dados relacional* organiza os dados na forma de *tabelas*. Cada tabela contém várias linhas, cada uma contendo um *registo* (ou *tuplo*) de dados de uma entidade de um certo tipo, e cada linha contém várias colunas, correspondendo aos diferentes *atributos* (ou *campos*) dessa entidade. Por exemplo, os dados de contactos descritos acima poderiam ser guardados numa base de dados relacional na forma de uma tabela com 4 registos com 5 atributos cada, como representado abaixo.

**Table: contacts**

firstname (TEXT)	middlename (TEXT)	lastname (TEXT)	email (TEXT)	phone (TEXT)
João	Manuel	Fonseca	jmf@gmail.com	912345654
Pedro	Albuquerque	Silva	pedro23@gmail.com	932454349
Maria	Carreira	Dinis	mariadi@ua.pt	234958673
Catarina	Alexandra	Rodrigo	calexro@sapo.pt	963343386

Na base de dados, a tabela é caracterizada por um nome e pelo identificador e tipo de dados de cada atributo.

Os *sistemas de gestão de bases de dados*(SGBD) são programas que permitem fazer pesquisas e alterações de registos numa base de dados, segundo critérios complexos, de forma versátil e eficiente. Um ponto importante é que os dados possuem relações entre si, sendo que base de dados segue um dado modelo relacional. Do exemplo anterior podemos assumir que o modelo possui pessoas e contactos. As pessoas podem ter um ou mais contactos e os contactos pode ser de dois tipos: email e telefone. Com base neste tipo de organização, durante os exercícios seguintes, irá ser explorado como se pode aceder e armazenar informação.

Para isto, existem diversos sistemas de gestão de bases de dados relacionais, mas todos suportam interação com programas clientes através de uma linguagem comum: a linguagem Structured Query Language (SQL). Esta permite efectuar operações muito detalhadas sobre a estrutura da informação ou os seus dados, o que iremos explorar de seguida.

Para criar uma base de dados vamos usar o **sqlite3**, que é um SGBD simples que permite criar bases de dados num sistema de ficheiros tradicional, usando o formato **SQLite**. Esta base de dados é muito usada e encontra-se por exemplo nas plataformas Android e iOS. Em alternativa, poderíamos utilizar um servidor de bases de dados, como se faz tipicamente em ambientes de produção, mas do ponto de vista de utilização as diferenças são mínimas.

Para criar uma base de dados é necessário executar a ferramenta **sqlite3** com a seguinte sintaxe:<sup>1</sup>

---

```
sqlite3 database.db
```

---

Deve aparecer novo *Prompt* que nos permite invocar comandos sobre a base de dados. Um exemplo de um comando é a criação de uma tabela, que se faz através da linguagem SQL:

---

```
CREATE TABLE tablename(
    field1 TYPE1,
    field2 TYPE2
);
```

---

Em que **tablename** representa o nome da tabela, **field1** representa o nome da primeira coluna e **TYPE1** o seu tipo de dados. São suportados vários tipos de dados. Os mais relevantes para este trabalho são:

**TEXT:** Texto.

**INTEGER:** Números inteiros.

**REAL:** Números reais.

**BLOB:** Um qualquer conteúdo (ex, uma imagem).

---

<sup>1</sup>Esta ferramenta deve existir nos repositórios da maioria dos sistemas.

### Exercício 18.1

Crie uma tabela chamada **contacts** contendo os campos identificados anteriormente. Para facilitar a criação da base de dados, crie os comandos num ficheiro de texto e depois copie e cole o texto para o *Prompt* da ferramenta.

Pode verificar o conteúdo da tabela se executar o comando **.tables**. Pode sair desta ferramenta executando o comando **.quit**. (Estes comandos são específicos do **sqlite**. Não são SQL.)

Depois de criada a tabela, é necessário inserir dados. Para isso, usa-se o comando **INSERT INTO**.

---

```
INSERT INTO tablename  
VALUES (  
    "value1", "value2"  
)
```

---

A directiva **VALUES** indica os valores a colocar nas colunas existentes. Valores do tipo **TEXT** devem possuir aspas. É possível inserir informação em apenas alguns campos, bastando para isso que se especifiquem quais os campos de destino da informação.

---

```
INSERT INTO tablename(field1)  
VALUES (  
    "value1"  
)
```

---

### Exercício 18.2

Construa os comandos necessários para inserir os dados de todos os clientes. Mais uma vez, crie os comandos num ficheiro de texto e depois copie-os para o **sqlite3**.

De forma consultar os dados inseridos em tabelas utiliza-se o comando **SELECT**. Com este comando é possível discriminar de forma muito detalhada que informação deve ser obtida e em que formato. Na sua forma básica o comando **SELECT** pode obter toda a informação de uma tabela:

---

```
SELECT * FROM contacts;
```

---

O que deverá produzir o seguinte resultado:

---

```
João|Manuel|Fonseca|jmf@gmail.com|912345654
Pedro|Albuquerque|Silva|pedro23@gmail.com|932454349
Maria|Carreira|Dinis|mariadi@ua.pt|234958673
Catarina|Alexandra|Rodrigo|calexro@sapo.pt|963343386
```

---

Também é possível obter apenas algumas colunas, e/ou apenas algumas linhas. Por exemplo, poderemos obter apenas o email e número de telefone dos contactos com nome Pedro:

---

```
SELECT email, phone FROM contacts WHERE firstname="Pedro";
```

---

### Exercício 18.3

Construa vários comandos que permitam obter informação específica sobre os utilizadores e teste-os.

### Exercício 18.4

Adicione a directiva **ORDER BY columnname ASC** aos seus comandos e compare o resultado. Em vez de **ASC**, também pode utilizar **DESC**

Por vezes é necessário actualizar informação, nomeadamente mudar o valor de células específicas ou mesmo apagar linhas inteiras. O comando **UPDATE** permite actualizar uma ou mais células. Por exemplo, podemos mudar o número de telefone do João Fonseca através do comando:

---

```
UPDATE contacts SET phone = 912345653 WHERE email="jmf@gmail.com";
```

---

Tem de se ter em consideração que o comando **UPDATE** altera todas as linhas, excepto se for especificada uma regra que restrinja o número de linhas a considerar. Neste caso isto é feito usando **WHERE email="jmf@gmail.com"**. Sem esta especificação o comando

**UPDATE** iria colocar todas as linhas com o mesmo número de telefone.

### Exercício 18.5

Construa um comando que altere o último nome do utilizador com telefone 963343386 para "Sousa".

Finalmente, é possível apagar linhas inteiras utilizando o comando **DELETE**. Deve-se ter o mesmo cuidado que com o comando **UPDATE** no sentido em que o comando **DELETE** pode apagar uma ou mais linhas. O exemplo seguinte apaga uma linha específica da tabela:

```
DELETE FROM contacts WHERE phone = 912345653;
```

#### 18.1.1 Modelo Relacional

Em geral, uma aplicação terá várias tabelas, cada uma com um tipo de dados. Por exemplo, à nossa base de dados podemos acrescentar uma tabela das empresas clientes, às quais pertencem os clientes anteriormente listados.

Table: companies

name	address	vatnumber
MaxiPlano	Aveiro	123123123123
Luís Manuel & filhos	Águeda	54534343435
ProDesign	Porto	54534343435

### Exercício 18.6

Crie uma tabela chamada **companies** e insira a informação anterior.

Levanta-se agora a questão de como indicar que uma dada pessoa pertence a uma empresa. Uma solução passaria por armazenar o nome da empresa junto de cada contacto. E a morada e número fiscal da empresa? Também se acrescentam à tabela de contactos? Mas isso implicaria replicar informação, visto que várias pessoas pertencem à mesma empresa. Isso é um problema porque caso uma empresa mudasse a morada, teríamos de pesquisar todos os contactos e alterar esse atributo, ou senão ficaríamos com informações incoerentes!

A solução para esta questão passa por estabelecer uma *relação* entre as tabelas da base de dados.<sup>2</sup> Para criar relações entre dados, cada registo de uma tabela deve ter uma *chave*, que pode depois ser utilizada noutra tabela para o referir. Uma chave é um qualquer atributo que permita identificar univocamente o registo. As chaves podem ser de qualquer tipo, mas são tipicamente números inteiros, que até podem ser atribuídos de forma automática quando o registo é criado.

Para estabelecer a relação no nosso exemplo, a tabela das empresas passaria a ser:

**Table: companies**

<b>id</b>	<b>name</b>	<b>address</b>	<b>vatnumber</b>
1	MaxiPlano	Aveiro	123123123123
2	Luis Manuel & filhos	Águeda	54534343435
3	ProDesign	Porto	54534343435

Enquanto que a tabela de contactos ficaria:

**Table: contacts**

<b>id</b>	<b>firstname</b>	<b>middlename</b>	<b>lastname</b>	<b>email</b>	<b>phone</b>	<b>company_id</b>
1	João	Manuel	Fonseca	jmf@gmail.com	912345654	3
2	Pedro	Albuquerque	Silva	pedro23@gmail.com	932454349	2
3	Maria	Carreira	Dinis	mariadi@ua.pt	234958673	1
4	Catarina	Alexandra	Rodrigo	calexro@sapo.pt	963343386	1

Em ambas as tabelas, a coluna **id** é uma chave: serve para identificar de forma única cada registo. Na tabela **contacts**, o campo **company\_id** contém a chave da empresa a que pertence cada pessoa. Neste caso o João pertence à empresa ProDesign, enquanto a Maria e a Catarina pertencem ambas à empresa MaxiPlano.

Assim, a informação entre contactos e empresas encontra-se relacionada. A relação é estabelecida pelo campo **company\_id** da tabela contactos. A este tipo de campo chama-se uma *chave estrangeira* por conter valores que são chaves (ditas *chaves primárias*) noutra tabela.

Os comandos **SELECT** podem ser construídos de forma a permitir pesquisar informação relacionada distribuída por diversas tabelas. Por exemplo, poderíamos listar todos os contactos da empresa MaxiPlano com o seguinte comando:

---

```
SELECT contacts.*  
FROM contacts,companies
```

<sup>2</sup>É esta característica que está na origem do termo Base de Dados Relacional.

```
WHERE contacts.company_id = companies.id  
AND companies.name = "MaxiPlano"
```

---

Analizando o comando verifica-se que ele lista todos os campos dos registo da tabela **contacts** que possuam no atributo **company\_id** dessa tabela um valor igual ao do atributo **id** da tabela **companies** sempre que **name** nessa mesma tabela seja igual a MaxiPlano.

### Exercício 18.7

Volte a criar uma tabela de empresas com uma nova coluna chamada **id**. Especifique o tipo como sendo **INTEGER PRIMARY KEY AUTOINCREMENT**. Isto irá criar uma coluna que armazena um contador inteiro, único e incrementado automaticamente quando novas linhas são inseridas. Recomenda-se que se crie uma base de dados diferente da anterior.

---

### Exercício 18.8

Aplique o mesmo processo à tabela de contactos mas neste caso adicione também uma nova coluna no final chamada **company\_id**. Esta coluna serve para indicar a que empresa pertence um dado contacto, pelo que deve atualizar a base de dados de forma a que os contactos pertençam a uma empresa.

---

### Exercício 18.9

Construa instruções que permitam listar todas os contactos das empresas de Aveiro.

---

## 18.2 Acesso Programático

Na secção anterior vimos os comandos básicos que permitem criar, aceder e modificar uma base de dados relacional. Nesta secção veremos como fazer o mesmo dentro de um programa.

Como em geral as bases de dados podem residir num sistema servidor, distinto do sistema que executa o programa cliente, o procedimento a seguir envolve três fases:

1. estabelecer a ligação à base de dados;
2. fazer as consultas e/ou alterações aos dados;
3. terminar a ligação.

Estas fases também são seguidas mesmo no caso da base de dados residir num ficheiro local.

Em Python o acesso a bases de dados do tipo *sqlite* é fornecido pelo módulo **sqlite3**, que tem de ser importado. Para instalar o módulo, pode recorrer ao gestor de pacotes do seu sistema, ou ao comando **pip3** de acordo com as permissões que possuir.

---

```
apt-get install python3-sqlite
```

---

ou:

---

```
pip3 install --user pysqlite
```

---

O exemplo seguinte mostra o esqueleto de um programa que abre um ficheiro fornecido no primeiro argumento (p.ex, **database.db**).

---

```
import sqlite3 as sql
import sys

def main(argv):
    db = sql.connect(argv[1])      # estabelecer ligação à BD
                                    ...
                                    # realizar operações sobre a BD
    db.close()                   # terminar ligação

main(sys.argv)
```

---

O método **connect** estabelece a ligação à base de dados e devolve um objeto que representa essa ligação. É através desse objeto que serão realizadas as operações na base de dados e deve-se depois terminar a ligação. Se o argumento do método **connect** for igual a "**:memory:**", a base de dados é criada em memória, sendo apagada no final do programa. Isto é útil caso de pretenda armazenar informação apenas durante a execução do programa, não sendo ela válida numa futura execução.

### Exercício 18.10

Construa um programa como indicado anteriormente e verifique que pode aceder à base de dados que criou anteriormente.

Tal como anteriormente, as operações sobre a base de dados são codificadas através de comandos SQL. As respostas podem depois ser obtidas sob o formato de linhas da tabela. O exemplo seguinte demonstra como é possível obter todos os registo da tabela de contactos.

---

```
import sqlite3 as sql
import sys

def main(argv):
    db = sql.connect(argv[1])

    result = db.execute("SELECT * FROM contacts")
    rows = result.fetchall()
    for row in rows:
        print(row)

    db.close()

main(sys.argv)
```

---

Neste exemplo o ciclo percorre os registo devolvidos pelo comando **SELECT** e, em cada iteração, a variável **row** recebe um tuplo com os atributos de um desses registo. Também é possível obter os resultados um de cada vez, o que por vezes é obrigatório caso o seu tamanho seja muito grande.<sup>3</sup> Para isso podemos utilizar o método **fetchone()**:

---

```
...
result = db.execute("SELECT * FROM contacts")
while True:
    row = result.fetchone()
    if not row:
        break
    print(row)
...
```

---

Ou pode-se simplesmente tratar o resultado como um iterador:

---

```
...
result = db.execute("SELECT * FROM contacts")
for row in result:
    print(row)
...
```

---

<sup>3</sup>As bases de dados possuem sempre um limite (por exemplo, 4MB) para o tamanho de cada resultado.

## Exercício 18.11

Considere os métodos anteriores e implemente um programa que imprima os nomes próprios de todos os contactos e indique quantos são, como no exemplo abaixo.

```
Catarina  
João  
Maria  
Pedro  
4 contactos
```

Frequentemente é necessário realizar pesquisas com argumentos variáveis. Por exemplo, podemos querer pesquisar todos os contactos cujo email pertença a um domínio indicado pelo utilizador. Assim, se o utilizador introduzir `%@gmail.com`, serão encontrados todos os contactos com email naquele servidor.

Uma solução naïf seria construir uma *String* com a estrutura do comando pretendido:<sup>4</sup>

```
...  
domain = raw_input("Domínio de email? ")  
result = db.execute("SELECT * FROM contacts WHERE email LIKE '%s'" % domain)  
# ATENÇÃO: ESTA SOLUÇÃO DEVE SER EVITADA!  
...
```

Esta abordagem, embora frequente, **deve ser absolutamente evitada!** Caso contrário, correm-se sérios riscos de segurança. Por exemplo, se o utilizador introduzir

```
%@gmail.com'; DELETE FROM contacts --
```

o comando SQL construído seria

```
SELECT * FROM contacts WHERE email LIKE '%@gmail.com'; DELETE FROM contacts --'
```

o que inclui duas instruções: **SELECT** e **DELETE**. Efectivamente o resultado é que todos os dados seriam apagados.

A solução correcta é a seguinte.

<sup>4</sup>O operador **LIKE** permite pesquisar valores com um certo padrão, que pode incluir partes desconhecidas.

```
...
domain = raw_input("Domínio de email? ")
result = db.execute("SELECT * FROM contacts WHERE email LIKE ?",
                   (domain,) )
# ESTA É A SOLUÇÃO CORRETA!
...
```

Note que se utiliza um ponto de interrogação para indicar explicitamente onde deve ser colocado o parâmetro variável e o seu valor é passado separadamente. Desta forma é o próprio módulo que constrói internamente o comando SQL e assim é impossível injetar comandos adicionais. Esta abordagem tem o nome de *Prepared Statements* e deve ser seguida independentemente da linguagem de programação.

### Exercício 18.12

Construa um programa que permita pesquisar contactos por qualquer um dos nomes. Para isto pode utilizar o operador **OR** para conjugar diferentes condições de pesquisa. Por exemplo: ...**WHERE firstname LIKE ? OR middlename LIKE ? ....**

### Exercício 18.13

Construa um programa que aceite um argumento, usando-o para localizar uma pessoa através das partes do seu nome, imprimindo a que empresa pertence.

## 18.3 Para Aprofundar

### Exercício 18.14

Obtenha a base de dados *Chinook* acedendo ao endereço <https://github.com/lerocha/chinook-database> e crie um programa que permita pesquisar por albums de música. O esquema da base de dados pode ser encontrado em <https://github.com/lerocha/chinook-database/wiki/Chinook-Schema>.

### Exercício 18.15

Obtenha a base de dados *Chinook* e crie uma aplicação que demonstre quais os 10 clientes que efectuaram o maior volume de compras.

### **Exercício 18.16**

Relembre um exercício anterior em que se capturava a taxa de ocupação de CPU.  
Replique o exercício, mas registando os valores numa base de dados relacional.

### **Exercício 18.17**

Implemente um programa que, tendo em consideração a base de dados criada no exercício anterior, imprima o valor médio de ocupação de processador entre duas datas.

## **Glossário**

**CPU**      Central Processing Unit

**CSV**      Comma Separated Values

**SQL**      Structured Query Language

## **Referências**

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



# Representação de Imagem

## Objetivos:

- Representação de cor nos diferentes espaços
- Efeitos básicos sobre imagens
- Marcas de água

### 19.1 Introdução

As imagens como fotografias, desenhos ou gráficos são deveras importantes para as aplicações computacionais actuais. Neste guião analisamos as formas de representação e processamento digital de imagens.

Para a realização deste guião será necessário instalar a biblioteca **Pillow**. Num computador com Debian/Ubuntu pode instalá-la através do comando:

---

```
sudo apt-get install python3-imaging
```

---

Em alternativa, pode instalar o pacote através do comando **pip**, mas antes terá de instalar várias dependências. Assim, em *Ubuntu* deverá correr:

---

```
sudo apt-get install libtiff4-dev libjpeg8-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.5-dev tk8.5-dev python-tk
pip3 install Pillow
```

---

Em *OS X* deverá correr:

---

```
brew install libtiff libjpeg webp little-cms2
pip3 install Pillow
```

---

Por razões pedagógicas, neste guião serão sugeridos processos que podem ser sub-ótimos. As bibliotecas **Pillow** e **OpenCV** possuem transformações otimizadas para muitos dos processos aqui tratados, devendo qualquer código de produção fazer uso desses métodos.

## 19.2 Imagens

Num computador, as imagens são representadas na forma de uma matriz composta de pequenas células organizadas em linhas e colunas. A cada célula da matriz dá-se o nome de píxel e corresponde ao menor elemento da imagem que pode ter cor distinta dos restantes. A *geometria da imagem* é definida pela largura e altura, medida em número de píxeis (ver Figura 19.1).

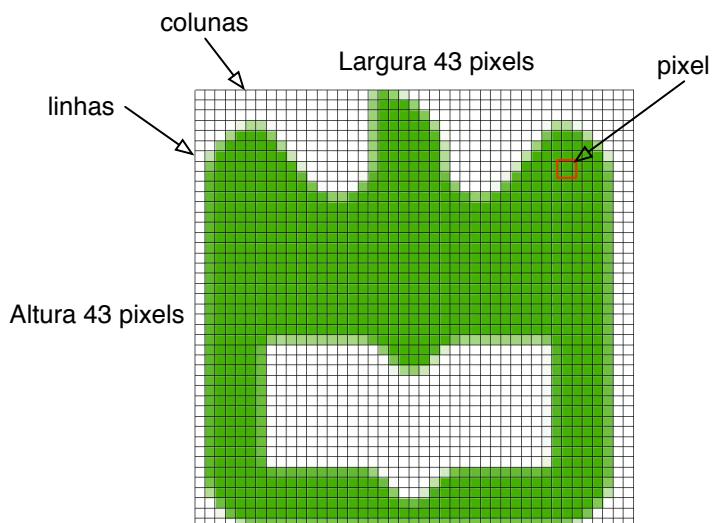


Figura 19.1: Composição de uma imagem digital.

Além da informação para representação da imagem, a maioria dos formatos de representação adiciona dados extra aos ficheiros, que são utilizados para indicar como deve o ficheiro ser processado ou que codificação é utilizada. Alguns outros, como o formato Joint Photographic Experts Group (JPEG), podem incluir dados num formato denominado Exchangeable image file format (Exif), que indicam entre outros aspetos que programa produziu a imagem ou qual a máquina fotográfica utilizada e os parâmetros da lente (no

caso de uma fotografia). Também pode ser incluída uma versão miniatura da imagem para pré-visualização no navegador de ficheiros do sistema.

Num programa em *Python* é possível inspeccionar esta informação através da biblioteca **Pillow**, como se demonstra no exemplo abaixo.

---

```
from PIL import Image
from PIL import ExifTags
import sys

def main(fname):
    im = Image.open(fname)

    width, height = im.size

    print("Largura: %dpx" % width)
    print("Altura: %dpx" % height)
    print("Formato: %s" % im.format)

    tags = im._getexif()

    for k,v in tags.items():
        print(str(ExifTags.TAGS[k])+" : "+str(v))

main(sys.argv[1])
```

---

### Exercício 19.1

Utilize o programa anterior para analisar os ficheiros de imagem fornecidos.

O tamanho em píxeis de uma imagem apresentada num monitor pode não ser igual ao tamanho da imagem armazenada. Por exemplo, quando se utiliza HyperText Markup Language (HTML)[1], o tamanho de apresentação de uma imagem pode ser especificado através de atributos que se aplicam ao elemento <img>, independentemente do tamanho original da imagem. Este redimensionamento obriga o computador a gerar uma nova imagem mais pequena ou maior que a original. Note que este processo é destrutivo e não cria informação. Ou seja, ao reduzir a dimensão de uma imagem, não se está a reduzir a dimensão de cada píxel, mas antes a substituir vários por um só, perdendo informação dos outros. Ao aumentar a dimensão de uma imagem, criam-se novos píxeis, mas os seus valores (cores) são dependentes apenas dos píxeis originais; a definição da imagem não melhora, fica apenas mais “esbatida”.

Podem-se criar imagens com diversos tamanhos usando o código seguinte.

```
...
def main(fname):
    im = Image.open(fname)
    width, height = im.size

    for s in [0.2, 8]:
        dimension = ( int(width*s), int(height*s) )
        new_im = im.resize( dimension, Image.NEAREST)
        new_im.save(fname+"-%.2f.jpg" % s)

...
```

### Exercício 19.2

Implemente o código anterior e experimente modificar a dimensão de alguns ficheiros.

### Exercício 19.3

O método utilizado para modificar as imagens é um dos mais simples (**Nearest Neighbour**). Existem vários outros, tais como **BILINEAR**, **BICUBIC** ou **ANTIALIAS**. Uns são mais adaptados à redução, outros à ampliação. Teste-os e compare visualmente o resultado.

## 19.3 Formatos de ficheiros

Existem vários formatos para a representação de imagens, sendo que cada um é optimizado para um tipo particular de informação. O formato mais utilizado para representar fotografias é sem dúvida o formato JPEG. Já os conteúdos gráficos na Web utilizam sobretudo o formato Portable Network Graphics (PNG). Outro formato popular é o Tagged Image File Format (TIFF), especialmente nos meios criativos, ou o BitMaP image file (BMP) em sistemas mais antigos. Existem razões objetivas para preferir um formato em detrimento de outro. As diferentes formas de representação e de compressão de dados dos diferentes formatos são mais ou menos adequados consoante o tipo de imagem (se é um desenho ou uma fotografia, por exemplo).

No caso concreto da compressão, os formatos BMP, PNG e TIFF não aplicam compressão ou aplicam uma compressão sem perda de informação. Por outro lado, o formato JPEG aplica algoritmos de compressão com perdas para conseguir reduzir substancialmente o tamanho dos ficheiros. Como os algoritmos estão otimizados para imagens naturais como fotografias, os artefactos introduzidos tornam-se mais evidentes quando aplicados a imagens artificiais como desenhos com áreas de cores sólidas e alto contraste.

O formato JPEG possui uma escala de compressão que varia em 0 e 100, sendo que quanto mais baixo for o valor, maior compressão e maiores perdas serão observadas. A Figura 19.2 demonstra o resultado de utilizar uma compressão de 30% para uma imagem com cores sólidas. Como se pode notar, a imagem foi bastante adulterada e torna-se evidente que o algoritmo processa as imagens por blocos.

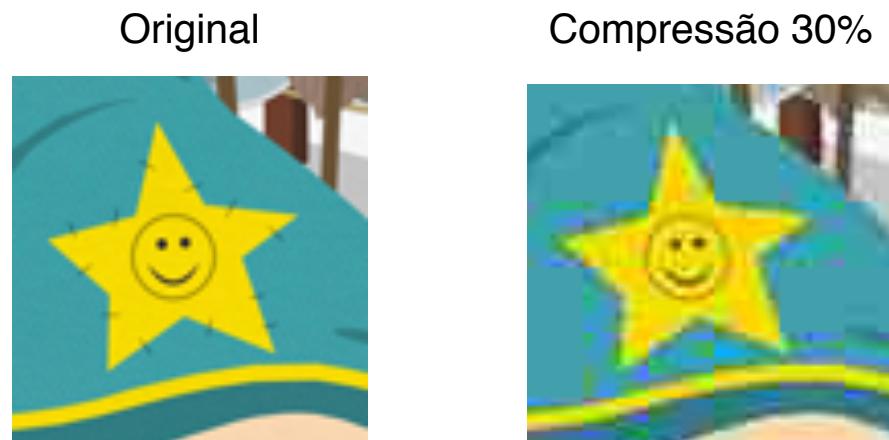


Figura 19.2: Compressão com JPG.

#### Exercício 19.4

Determine o tamanho dos blocos utilizados para comprimir a imagem anterior. Recomenda-se que simplesmente aumente o zoom do visualizador e conte os píxeis. (É conveniente desativar qualquer opção de suavização nas preferências do visualizador.) A imagem possui uma geometria de 90px por 88px.

O exemplo que se segue comprime o mesmo ficheiro com 11 valores de qualidade distintos e pode ser utilizado para verificar este aspeto.

```
from PIL import Image
import sys

def main(fname):
    im = Image.open(fname)
    for i in [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:
```

```
im.save(fname + "-test-%i.jpg" % i, quality=i)  
main(sys.argv[1])
```

---

### Exercício 19.5

Utilize o exemplo anterior para criar versões comprimidas do ficheiro **southpark.png** e do ficheiro **vasos.jpg**. Determine para ambos os casos a partir de que valor de qualidade os erros adicionados perturbam a imagem. Tenha em especial atenção as áreas de alto contraste, tais como linhas.

### Exercício 19.6

Escolha um dos ficheiros fornecidos em formato JPEG e crie uma versão em cada um dos formatos: PNG, TIFF, BMP. Compare o tamanho do ficheiro criado.

## 19.4 Representação de cor

Existem várias formas para representar cores numa imagem. A escolha do formato mais apropriado depende do tipo de imagem que se possui e do objectivo da informação. O método mais comum e já visto nas aulas anteriores (ex, HTML) é o formato **RGB**. Neste formato, cada cor é representada por 3 componentes de cores primárias: vermelho (R), verde (G), e azul (B). A Figura 19.3 apresenta alguns valores **RGB** do logótipo da Universidade de Aveiro.

As cores **RGB** são denominadas por cores primárias aditivas. Ao se somarem estas cores é possível reconstruir todas as outras. Em contrapartida, o modelo **CMYK** (Cyan, Magenta, Yellow, Black), constituído pelas cores primárias subtrativas também é capaz de formar todas as cores, mas quando elas são subtraídas (absorvidas). Num ecrã os píxeis emitem luz, portanto as cores somam a sua intensidade e é utilizado o modelo **RGB**. Numa impressora as tintas depositadas num papel absorvem luz e por isso é frequentemente utilizado o modelo **CMYK**.

Existem vários modos de representação de cor que podem ser comumente utilizados:

- **BW ou 1:** 1 canal com apenas 1 bit por píxel, que pode representar uma de duas cores. Utilizado para representar imagens a preto e branco.
- **RGB:** 3 canais, representando as intensidades de vermelho, verde e azul. Muito utilizado para a representação de cores em monitores.

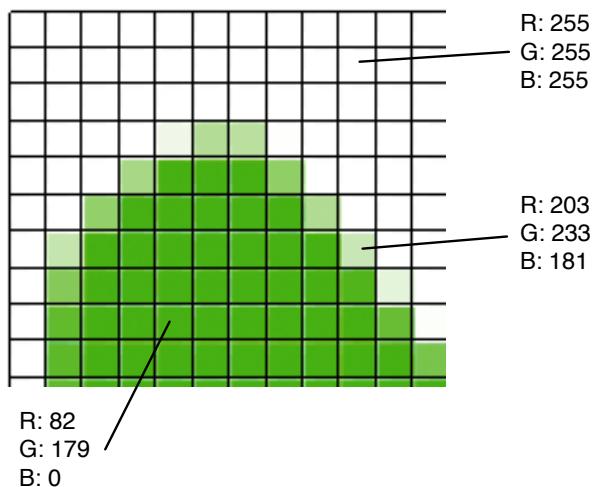


Figura 19.3: Valores RGB do logótipo da UA

- **RGBA:** O mesmo que **RGB** com um canal adicional (canal alfa) para representar a transparência. Suportado por alguns formatos como o PNG.
- **CMYK:** 4 canais, utilizando ciano/cião, magenta, amarelo e preto. Muito utilizado para a representação de cores para impressão.
- **L:** Apenas um canal representando a luminância (intensidade de luz). Utilizado para representar imagens em tons de cinza.
- **P:** Paleta de cores específica. Cada píxel da imagem tem um número que é usado como índice para uma tabela de cores (uma *paleta*), que contém a cor desejada num dos outros formatos (tipicamente RGB). Utilizada para formatos como o Graphics Interchange Format (GIF).
- **YCbCr:** 3 canais representando a luminância (Y), crominância azul (Cb), crominância vermelha (Cr). A luminância pode ser obtida por uma média ponderada dos valores RGB, enquanto os canais Cb/Cr são determinados pela diferença entre os valores de vermelho/azul e a luminância,  $Cb = B - Y$  e  $Cr = R - Y$ . O sistema é utilizado para fotografias (JPEG) e em vídeo.

### Exercício 19.7

Verifique o atributo **mode** de uma imagem para cada um dos ficheiros fornecidos.

É possível converter as imagens entre modos de representação, o que pode trazer vantagens do ponto de vista de representação e processamento. Para casos de utilização dos modos de cor, consulte a secção 19.5.

Repare que na Figura 19.3 os valores apresentados se referem a uma escala com  $2^8 = 256$  níveis para cada cor, sendo que o total de cores representáveis será igual a  $2^{3 \times 8}$ . O número pode parecer grande mas na realidade não é adequado para todos os fins. As máquinas fotográficas atuais produzem imagens *RAW* com resoluções de 14 a 16 bits por componente, que depois são convertidas para o formato de 8 bits. No processo perde-se informação, pelo que alguns formatos como o TIFF permitem guardar 8 bits, 16 bits e 32 bits por componente. Outros como o Flexible Image Transport System (FITS) permite um número indeterminado de bits por píxel.

Não sendo possível analisar em concreto e de forma fácil o resultado de se terem imagem de 16bits, pode-se fazer o exemplo contrário: restringir o número de bits de uma imagem e observar como isso altera as cores percebidas pelo olho humano. O exemplo seguinte anula (coloca a 0) os 4 bits menos significativos de cada componente, reduzindo o ficheiro para 4 bits de resolução efetiva.

```
...
def main(fname):
    im = Image.open(fname)

    width, height = im.size

    for x in range(width):
        for y in range(height):
            p = im.getpixel( (x,y) )
            r = p[0] & 0b11110000
            g = p[1] & 0b11110000
            b = p[2] & 0b11110000
            im.putpixel( (x,y), (r,g,b) )

    im.save(fname+"-4bits.jpg")
...
```

### Exercício 19.8

Repique o exemplo anterior e experimente anular quantidades diferentes de bits.

## 19.5 Efeitos sobre imagens

São várias as manipulações que podem ser aplicadas a imagens de forma a alterar o seu aspeto. Podem focar-se na manipulação das cores ou mesmo na alteração da geometria da imagem. As sub-secções seguintes demonstram como algumas manipulações podem ser conseguidas. Os programas foram escritos como forma de explicar os métodos da forma mais simples. Privilegiou-se a legibilidade e compreensibilidade, não a eficiência. A biblioteca **Pillow** possui métodos otimizados para muitas destas operações, que deverão ser utilizados em situações reais.

Recomenda-se que se implemente cada efeito como uma função que aceite como parâmetro uma imagem e devolva novamente uma imagem. Desta forma torna-se possível encadear efeitos.

### 19.5.1 Troca de Cores

A troca de cores das imagens é um efeito simples que resulta da troca dos canais de uma imagem. Tipicamente aplicado no modo **RGB** pois permite um controlo mais direto sobre a imagem. O exemplo seguinte troca o canal verde pelo vermelho, sendo o resultado o demonstrado na Figura 19.4

```
...
new_im = Image.new(im.mode, im.size)

for x in range(width):
    for y in range(height):
        p = im.getpixel( (x,y) )
        r = p[1]
        g = p[0]
        b = p[2]
        new_im.putpixel((x,y), (r, g, b) )
...

```

#### Exercício 19.9

Construa uma função que troque os canais de cores de uma imagem.

#### Exercício 19.10

Construa uma função que crie uma imagem negativa. Esta imagem consiste na substituição de cada valor  $v$  por  $255 - v$ .



Figura 19.4: Figura original em RGB e com os canais R e G trocados.

### 19.5.2 Tons de Cinza

Converter para tons de cinza implica remover toda a informação cromática, deixando apenas a intensidade. No modo **YCbCr** isto pode ser feito de forma imediata mantendo apenas o primeiro canal, mas não é o método mais poderoso. A biblioteca **Pillow** converte igualmente de forma automática qualquer imagem para tons de cinza, especificando o modo **L**.

O exemplo seguinte converte uma imagem para o modo **L** e guarda-a.

---

```
...
def main(fname):
    im = Image.open(fname)
    new_im = im.convert("L")
    new_im.save(fname+"-L.jpg")
...

```

---

O resultado é o demonstrado na Figura 19.5.

Converter uma imagem para escala de cinzas implica processar as suas cores e aplicar uma fórmula que converta **RGB** (ou outro modo) em **L**. No caso anterior a fórmula utilizada é  $L = \frac{299}{1000}R + \frac{587}{1000}G + \frac{114}{1000}B$ , mas podem ser utilizados outras expressões, tal como utilizar apenas uma cor, ou combinar as cores de forma diferente. Frequentemente existem vantagens em aplicar um processamento diferente. O exemplo seguinte aplica a fórmula descrita, mas de uma forma manual.

---

```
...
def effect_gray(im):
    width, height = im.size
    new_im = Image.new("L", im.size)
```

---



Figura 19.5: Figura original em RGB e em tons de cinza (L).

```

for x in range(width):
    for y in range(height):
        p = im.getpixel( (x,y) )
        l = int(p[0]*0.299 + p[1]*0.587 + p[2]*0.144)
        new_im.putpixel( (x,y), (l) )

return new_im
...

```

### Exercício 19.11

Replique o exemplo anterior mas combinando as cores de forma diferente. Pode, por exemplo, considerar apenas uma ou duas cores, combinadas ou utilizadas sem qualquer processamento. Verifique o resultado final.

#### 19.5.3 Controlo de Intensidade

O controlo de intensidade resulta na alteração dos valores de intensidade da imagem. Valores mais altos resultam numa imagem mais clara, valores mais baixos resultam numa imagem mais escura. Esta operação é bastante simplificada quando aplicada no modo **YCbCr**, pois o primeiro canal (Y) é o único que possui informação de intensidade.

A Figura 19.6 demonstra o resultado de manipular a intensidade de uma imagem.

A aplicação deste efeito requer assim uma conversão de modo de representação e a multiplicação do canal Y por um factor. Se o factor for superior a 1 a imagem ficará com uma intensidade superior, se o valor for inferior a 1 ela aparecerá mais escura. É necessário ter em atenção que os valores resultantes têm de ser inteiros e nunca podem ultrapassar o mínimo ou o máximo da escala.



Figura 19.6: Figura com intensidade aumentada ( $f=1.5$ ) ou diminuída ( $f=0.5$ ).

---

```
def effect_intensity(im, f):
    new_im = im.convert("YCbCr")
    width, height = im.size

    for x in range(width):
        for y in range(height):
            pixel = new_im.getpixel( (x,y) )
            py = min(255, int(pixel[0] * f))      # [0] is the Y channel
            new_img.putpixel( (x,y), (py, pixel[1], pixel[2]) )
    ...
```

---

### Exercício 19.12

Construa uma função que multiplique a intensidade de um ficheiro por um factor.

#### 19.5.4 Controlo de Gama

A gama de uma imagem diz respeito a quanto linear é a representação da intensidade dos seus valores. Tipicamente as imagens necessitam de ser corrigidas de forma a que a intensidade seja adaptada ao meio de reprodução. Nos Cathode Ray Tubes (CRTs), já raramente utilizados, é necessário aplicar curvas de compensação, pois a sua intensidade de reprodução não é linear. A Figura 19.7 apresenta este problema. Os CRTs possuem uma gamma típica de 2.2, o que significa que reproduzem as cores segundo uma linha exponencial. Uma correção típica irá distorcer as cores com o valor  $\frac{1}{2.2}$  de forma que a imagem realmente percebida pelos utilizadores seja apresentada de forma linear.

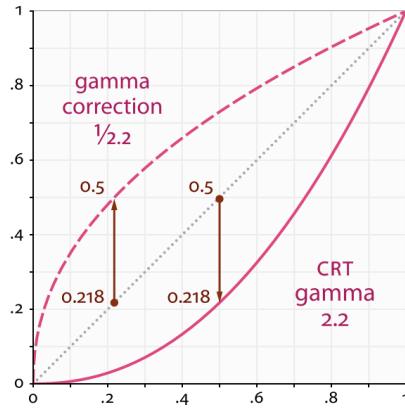


Figura 19.7: Correção de gama para um CRT (Fonte Wikimedia Foundation)



Figura 19.8: Figura com gama de 2.2 ou de 0.5.

Para além da compensação de não-linearidades dos dispositivos, este tipo de operações também se usa para melhorar imagens obtidas em condições de iluminação deficiente, por exemplo. A Figura 19.8 apresenta duas imagens com compensações de gama distintas.

Este ajuste pode ser obtido aplicando-se uma fórmula  $py = y^g * f$  no modo **YCbCr**, onde  $y$  é o valor do canal **Y**,  $g$  é o factor de correção gamma e  $f$  é um factor de normalização da intensidade de imagem. O fator de normalização é dado pela fórmula  $f = \frac{m}{m^g}$ , onde  $m$  representa o valor máximo de intensidade (frequentemente 255).

### Exercício 19.13

Construa uma função que aplique uma correção de gama a uma imagem.

### 19.5.5 Controlo de Saturação

A saturação de uma imagem refere-se à intensidade das cores apresentadas. Uma imagem muito saturada terá cores vivas enquanto uma imagem pouco saturada terá tons muito suaves. Uma imagem em tons de cinza não possui qualquer saturação. A Figura 19.9 demonstra duas imagens com saturações bastante distintas.



Figura 19.9: Figura com saturação aumentada por 1.5 ou reduzida para 0.5.

O controlo de saturação também faz uso do modo **YCbCr**, alterando neste caso os canais **Cb** e **Cr**. Estes canais codificam a saturação como a diferença em relação ao seu valor central, 128. Portanto, quando mais distante de 128 for o valor maior será a saturação, enquanto que quanto mais próximo de 128 for o valor, menor será a saturação. Um método comum de controlo de saturação é o apresentado no exemplo que se segue, onde se considera que **p** é um dado píxel da imagem.

```
...
py = p[0]
pb = min(255,int((p[1] - 128) * f) + 128)
pr = min(255,int((p[2] - 128) * f) + 128)
...

```

#### Exercício 19.14

Construa uma função que aplique uma transformação de saturação na imagem.

### 19.5.6 Sépia

Um efeito artístico que também se baseia na manipulação de cores é o efeito Sépia, frequentemente utilizado para emular fotografias antigas. Este efeito é simplesmente uma

manipulação dos valores **RGB** de cada píxel de acordo com a seguinte fórmula:

$$R' = 0.189R + 0.769G + 0.393B$$

$$G' = 0.168R + 0.686G + 0.349B$$

$$B' = 0.131R + 0.534G + 0.272B$$

onde  $R$ ,  $G$  e  $B$  são os valores da imagem original e  $R'$ ,  $G'$  e  $B'$  são os novos valores.

Este efeito pode servir de base para muitos outros, bastando apenas a manipulação dos coeficientes aplicados em cada multiplicação. O resultado será tal como representado na imagem da esquerda da Figura 19.10. A imagem da direita é semelhante a um efeito chamado de *Lomography* e é obtido pela troca de  $R'$  por  $B'$  na fórmula anterior.



Figura 19.10: Figura convertida para tons SÉPIA ou Lomo.

### Exercício 19.15

Construa duas funções que implementem os efeitos descritos.

#### 19.5.7 Detecção de Bordas

A detecção de bordas é bastante importante para tarefas como o reconhecimento de texto, mas pode ter outras utilizações no domínio do reconhecimento de padrões. O funcionamento básico deste processo é o de detectar alterações significativas entre dois píxeis e adicionar um traço. O resultado pode ser uma imagem a duas cores (preto e branco), onde o preto indica as zonas de alto contraste, ou uma imagem semelhante à original mas onde é sobreposta informação. Existem diversos algoritmos de deteção de bordas, que funcionam em diferentes modos de representação de cor e aplicam pesquisas mais ou menos exaustivas.

Um algoritmo simples para este problema consiste em detectar variações através do cálculo da diferença entre píxeis adjacentes. Caso algum vizinho apresente uma diferença

superior a um limiar pré-definido, estamos perante uma borda. O resultado será o demonstrado na Figura 19.11.

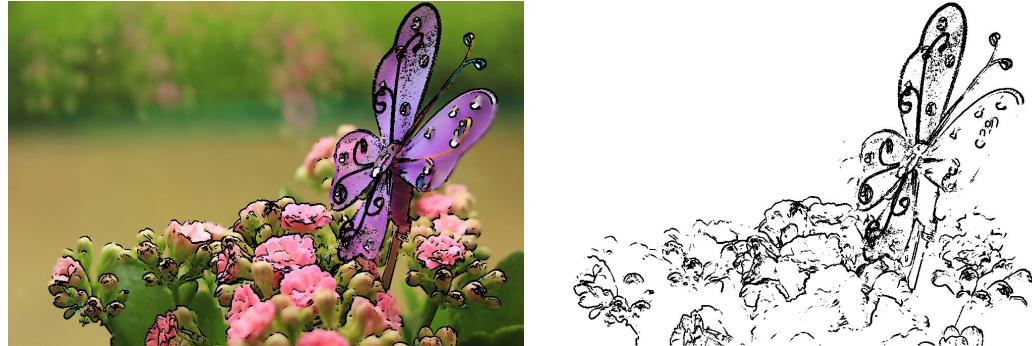


Figura 19.11: Bordas detectadas numa imagem.

A função seguinte implementa este algoritmo. Para cada píxel **p** que esteja na imagem e não seja um limite da imagem, consultam-se todos vizinhos (acima, abaixo, à esquerda e à direita). Caso a diferença para o píxel atual seja superior a **diff**, é devolvido um píxel preto. Caso contrário é devolvido um píxel original ou branco, dependendo da opção especificada em **bw**.

---

```
def is_edge(im, x,y, diff, bw):
    #Obter o pixel
    p = im.getpixel( (x , y) )
    width, height = im.size

    if x < width-1 and y < height-1 and x > 0 and y > 0:

        #Vizinhos acima e abaixo
        for vx in range(-1,1):
            for vy in [-1, 1]:
                px = im.getpixel( (x + vx, y + vy) )
                if abs(p[0]- px[0]) > diff:
                    return (0,128,128)

        #Vizinhos da esquerda e direita
        for vx in [-1, 1]:
            px = im.getpixel( (x + vx, y) )
            if abs(p[0]- px[0]) > diff:
                return (0,128,128)

    if bw :
        return (255,128,128)
    else:
        return p
```

---

### Exercício 19.16

Implemente uma função que calcule as bordas de uma imagem e teste-a para várias imagens fornecidas.

#### 19.5.8 Vignette

O efeito de Vignette é na realidade um defeito das lentes fotográficas em que as bordas das imagens ficam mais escuras que o seu centro. Diferentes lentes apresentarão diferentes níveis de *Vignette*, sendo típico de equipamento de baixa qualidade, ou mais antigo. Nas lentes modernas este efeito normalmente existe mas é pouco pronunciado. No entanto, o efeito pode ser aplicado a imagens já obtidas, sendo muito comum em algumas comunidades artísticas. A Figura 19.12 apresenta uma imagem sofrendo de *Vignette* e outra com um *Vignette* deslocado de forma a realçar o elemento decorativo.

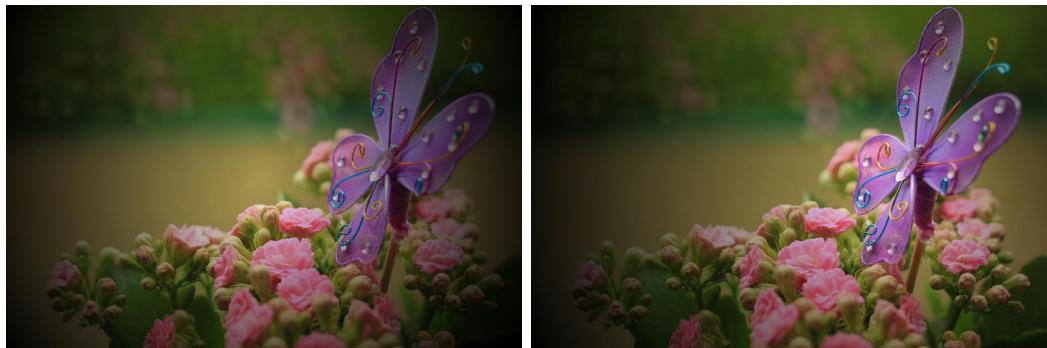


Figura 19.12: Vignette comum (esquerda) e deslocado para a direita (direita)

A implementação deste algoritmo implica que se determine um ponto de referência (normalmente o centro da imagem) e se calcule um factor de atenuação de intensidade (escurecimento) que é tanto maior quanto mais distante um píxel estiver da referência. A intensidade de todos os píxeis é multiplicada por este factor.

A distância entre dois pontos pode ser calculada através da fórmula da distância Euclidiana,

$$distance = \sqrt{(x - x_0)^2 + (y - y_0)^2}. \quad (1)$$

O factor de atenuação irá assim variar entre 0 (nenhum escurecimento) para o ponto de referência  $(x_0, y_0)$  e 1 (100%) para os limites da imagem.

```
def get_factor(x, y, xref, yref):
    distance = math.sqrt( pow(x-xref,2) + pow(y-yref,2) )
    distance_to_edge = math.sqrt( pow(xref,2) + pow(yref,2) )
```

```
return 1-(distance/distance_to_edge) #Percentagem
```

---

### Exercício 19.17

Implemente um efeito que aplique Vignette a uma imagem.

---

## 19.6 Marcação de imagens

### 19.6.1 Marca de Água

Uma marca de água é uma imagem que é sobreposta a outra imagem, normalmente utilizada para questões de direitos de autor. Para sobrepor uma imagem é necessário somar cada um dos píxeis das 2 imagens, depois de multiplicadas por um factor **f**. Este factor irá indicar o grau de transparência da marca de água. No exemplo que se segue quanto maior for **f**, menos transparente será a marca de água. Os valores **start\_x** e **start\_y** indicam a posição onde se deve iniciar a colocação da imagem.

---

```
...
#p1 é um pixel da imagem original
#p2 é um pixel da marca de água
p1 = im1.getpixel( (x+start_x, y+start_y) )
p2 = im2.getpixel( (x,y) )
if(p2[3] == 0):
    continue

r = int(p1[0]*(1-f)+p2[0]*f)
g = int(p1[1]*(1-f)+p2[1]*f)
b = int(p1[2]*(1-f)+p2[2]*f)
...
...
```

---

A Figura 19.13 mostra o resultado de uma operação destas, neste caso com **f=0.8**.



Figura 19.13: Fotografia da UA com o símbolo em marca de água.

### Exercício 19.18

Construa um programa que, aceitando duas imagens como argumento e um factor de transparência, adicione a segunda à primeira.

Uma alternativa para adicionar uma marca de água quase imperceptível é manipular os bits individuais da imagem de forma a codificar o bit mais significativo da marca de água no bit menos significativo da imagem a marcar. Ou seja, manipular o bit que introduz menos erro na imagem a marcar, codificando o bit da marca de água que possui maior valor. Isto é uma forma de esteganografia: a marca de água é escondida na imagem original de forma imperceptível. Em *Python* o processo é conseguido alterando o exercício anterior para que a transformação aplicada a cada canal seja a seguinte.

```
...
r = (p1[0] & 0b11111110) | (p2[0] >> 7)
b = ...
...
```

A recuperação da marca de água efectua-se processando toda a imagem e promovendo o bit menos significativo a mais significativo, o que se consegue com uma operação de *shift* à esquerda de 7 bits. Tipicamente este bit irá conter ruído, mas nos sítios onde foi codificada a marca de água, será possível identificá-la.

```
...
r = (p1[0] << 7) & 255
...
```

A Figura 19.14 mostra o resultado da imagem com a marca de água e a versão recuperada.

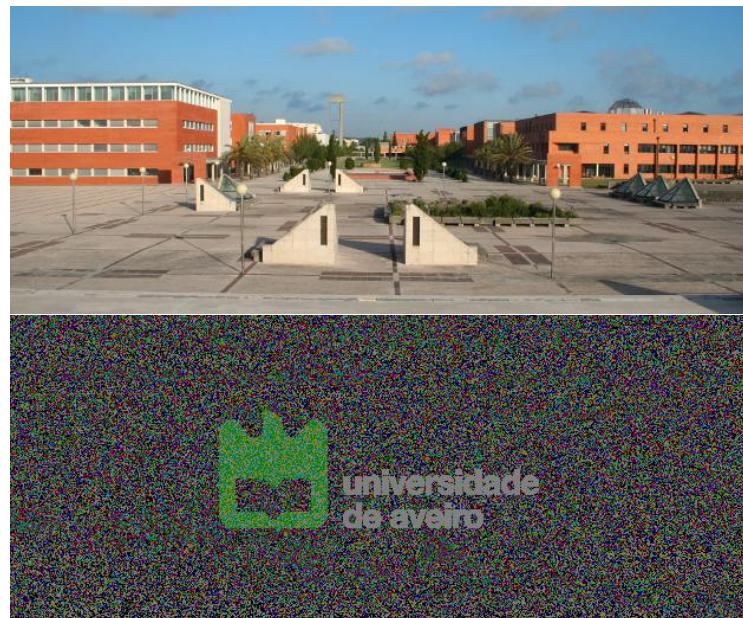


Figura 19.14: Imagem com marca de água usando técnicas de estanografia e marca de água recuperada.

### Exercício 19.19

Implemente um programa semelhante ao anterior mas que aplique a marca de água usando técnicas de esteganografia. Experimente processar todos os canais ou apenas alguns. Consegue notar diferenças na imagem original?

#### 19.6.2 Adição de texto

Uma outra forma de marcação de imagens é a sobreposição de textos sobre a imagem. Neste caso, o texto é construído directamente para a imagem através de métodos fornecidos pela biblioteca `Pillow`. O processo implica a selecção de um ficheiro de tipo de letra, um texto, um tamanho de letra, uma cor e a definição de uma posição para colocar o texto.

O exemplo seguinte acrescenta uma mensagem de texto a uma imagem `im`, neste caso a palavra `LabI` com tamanho 40, escrita a branco, na posição  $x = 20$ ,  $y = 20$ .

```
from PIL import ImageDraw  
...  
19.20
```

```
draw = ImageDraw.Draw(im)
font = ImageFont.truetype("caminho-para-um-ficheiro.ttf", 40)

draw.text( (20, 20) , "LabI" , (255,255,255) , font=font)
```

---

De notar que é necessário especificar onde se encontram os tipos de letra. Esta localização irá depender de cada sistema. Nos sistemas *Linux*, podem ser encontrados no directório **/usr/share/fonts/truetype**.

### Exercício 19.20

Implemente um programa que permita adicionar mensagens de texto a imagens.

---

## Glossário

<b>BMP</b>	BitMaP image file
<b>CRT</b>	Cathode Ray Tube
<b>Exif</b>	Exchangeable image file format
<b>FITS</b>	Flexible Image Transport System
<b>GIF</b>	Graphics Interchange Format
<b>HTML</b>	HyperText Markup Language
<b>JPEG</b>	Joint Photographic Experts Group
<b>PNG</b>	Portable Network Graphics
<b>TIFF</b>	Tagged Image File Format

## Referências

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



# Aplicações Móveis Web

## Objetivos:

- Conceitos sobre aplicações móveis
- Conceitos sobre HTML para ambientes móveis
- Interacção com câmara e GPS
- Comunicação com serviços externos

## 20.1 Introdução

Os ambientes móveis, nomeadamente os focados em *smartphones* e *tablets*, possuem métodos próprios para o desenvolvimento de aplicações, sobre a forma de um Software Development Kits (SDKs). Esses ambientes possibilitam aceder às capacidades do dispositivo, nomeadamente às Application Programming Interfaces (APIs) para acesso à câmara fotográfica, som, localização, acelerómetros ou outros sensores. As aplicações desenvolvidas podem igualmente trocar informação com servidores, permitindo a implementação de aplicações ricas de troca de mensagens, troca de imagens, ou jogos, entre muitas outras.

Uma desvantagem desta abordagem é que as três principais plataformas actualmente em utilização (*Windows Phone*, *Android*, *iOS*) apresentam APIs completamente diferentes. Além disso, todas estas plataformas exigem que as aplicações sejam desenvolvidas em linguagens diferentes. A plataforma *Android* utiliza *Java*, a plataforma *Windows Phone* utiliza *C#* e a plataforma *iOS* utiliza *ObjectiveC* ou *SWIFT*. Desenvolver uma aplicação para as três plataformas obriga a triplicar parte do trabalho, o que levanta vários problemas para o planeamento e manutenção do código produzido, assim como para o planeamento de um modelo de interacção consistente.

Felizmente, os navegadores *Web* presentes nas várias plataformas possuem capacidades bastante avançadas de processamento. Assim, em vez de três aplicações nativas, podemos

muitas vezes criar uma aplicação *Web*, que corre igualmente em qualquer plataforma. A grande vantagem é que a aplicação pode ser desenvolvida uma única vez, usando tecnologias standard como *JavaScript* e *HTML5*. Ainda para mais, bibliotecas como o *Twitter Bootstrap* foram desenhadas para permitirem uma visualização correta das páginas *Web* tanto em dispositivos móveis como em computadores com ecrãs maiores. A página do serviço *Facebook*, que se ajusta automaticamente ao dispositivo cliente, é um bom exemplo da capacidade de escalabilidade das páginas atuais.

É claro que as aplicações web também têm algumas desvantagens: a velocidade de execução pode ser inferior, o aspeto da aplicação poderá não ser exatamente igual ao das aplicações nativas, a execução depende da existência de uma ligação de dados e pode não ser possível aceder a todos os recursos disponíveis nativamente.

Para o desenvolvimento deste tipo aplicações é comum utilizarem-se sistemas como o *PhoneJS* ou *PhoneGap*. No entanto, a utilização destes sistemas requer conhecimentos mais avançados. Por isso, este guião foca-se na utilização das funcionalidades através de uma variante da biblioteca *Twitter Bootstrap* denominada de *Ratchet* que pode ser obtida em <http://goratchet.com/>. A documentação desta biblioteca pode ser obtida em <http://goratchet.com/components/>. Deve consultar esta documentação antes da execução deste guião.

## 20.2 Uma aplicação simples

As aplicações não são mais do que páginas *Web* em que a lógica é implementada em *JavaScript* e um conjunto de páginas de estilos criam a ilusão de se tratar de uma aplicação real. Antes de iniciar este ponto, aceda à documentação da biblioteca *Ratchet* e verifique que componentes estão disponíveis.

O desenvolvimento inicia-se obtendo a biblioteca *Ratchet* e colocando os ficheiros necessários nos directórios corretos (directório **dist**). Tem também de existir um ficheiro **index.html** que representará a aplicação.

O exemplo seguinte resulta numa aplicação apenas com uma barra de título com o texto **LABI**.

---

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>LABI</title>
<meta name="viewport"
      content="initial-scale=1, maximum-scale=1, user-scalable=no, minimal-ui">
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<link rel="stylesheet" href="css/ratchet.css">
```

```
<link rel="stylesheet" href="css/ratchet-theme-ios.css">
<link rel="stylesheet" href="css/app.css">
<script src="js/jquery-2.2.3.min.js"></script>
<script src="js/ratchet.js"></script>
</head>
<body>
<header class="bar bar-nav">
<h1 class="title">LABI</h1>
</header>
<div class="bar bar-footer">
</div>
<div class="content">
<ul class="table-view">
<li class="table-view-cell">Hello World</li>
</ul>
</div>
</body>
</html>
```

---

### Exercício 20.1

Crie um directório `hello/` e coloque nele um ficheiro `index.html` com o conteúdo acima. Obtenha a biblioteca *Ratchet* e copie o conteúdo do directório `dist/` para dentro do directório `hello/`. Verifique o resultado com o seu browser.

Para verificar a aplicação num telemóvel, pode enviar a pasta completa para o servidor `xcoa.av.it.pt` e depois aceder à página Web a partir do telemóvel. Ou então verifique a versão dos professores em `http://xcoa.av.it.pt/labi/lab2016/ratchet/hello/`.

Em alternativa ao `xcoa`, se estiver numa rede local sem firewalls, poderá activar um servidor HyperText Transfer Protocol (HTTP)[1] no seu computador para servir o conteúdo do diretório atual e poder aceder noutro dispositivo. Pode fazer isto executando `python -m SimpleHTTPServer`. (Na rede da UA, devido às firewalls instaladas, isto não resultará.)

---

### Exercício 20.2

Modifique a referência do tema para `ratchet-theme-android.css` e verifique que a aplicação muda para um aspeto semelhante ao sistema android.

---

### Exercício 20.3

Adicione um componente adicional à aplicação desenvolvida.

---

## 20.3 Localização

Frequentemente os dispositivos móveis possuem um equipamento de Global Positioning System (GPS) que permite determinar a localização com bastante exactidão. No entanto, é possível determinar a localização de outras formas, por exemplo através das redes sem fios disponíveis. Os diversos meios de determinação de localização foram assim englobados numa mesma API que permite obter a localização independente do método utilizado. Para salvaguardar a privacidade, o acesso da aplicação à localização requer uma autorização explícita pelo utilizador.

O código JavaScript abaixo demonstra como obter a localização.

---

```
function showPosition(position){  
    $("#location").html(position.coords.latitude+" "+position.coords.longitude);  
}  
  
$(document).ready(function() {  
    navigator.geolocation.getCurrentPosition(showPosition);  
});
```

---

O método `navigator.geolocation.getCurrentPosition` pede acesso à localização e regista a função `showPosition` como *callback* para ser chamada quando o resultado estiver disponível.

Na aplicação cliente será necessário incluir este código *JavaScript* e um elemento HyperText Markup Language (HTML)[2] com o identificador certo para apresentar o resultado.

---

```
<div class="content">  
    <div id="location"></div>  
</div>
```

---

Recomenda-se a utilização da biblioteca *jQuery* que tem de ser incluída no directório `js` e importada para a página (no `<head>`).

### Exercício 20.4

Altere a sua aplicação de forma a mostrar a localização do utilizador. Pode testar localmente ou enviar a página para o servidor `xcoa.av.it.pt`.

### Exercício 20.5

Utilizando LeafletJS adicione um mapa centrado na posição atual do dispositivo.

## 20.4 Comunicação com serviços externos

A comunicação com serviços externos é importante pois permite que uma aplicação possa trocar informação com serviços, ou mesmo com outros utilizadores. Aplicações de *chat*, de visualização do estado do tempo, ou de partilha de outra informação (ex, imagens), podem ser implementadas rapidamente. Acima de tudo, este tipo de aplicações é útil para demonstrar que as aplicações Web, mesmo em ambientes móveis, podem ser dinâmicas e interagir com outros serviços externos.

Um exemplo simples é o de obter a data e hora de um servidor remoto. Para isto serão necessários os seguintes componentes:

- Um botão para iniciar o processo (pode ser substituído por um *timer* caso se pretenda actualizar a informação de forma periódica.)
- Código *JavaScript* que obtenha a informação do servidor remoto.
- Uma aplicação que implemente o serviço pedido.
- Um elemento HTML para armazenar o resultado.

Os dois elementos HTML são adicionados no elemento de classe **content**.

```
...
<div class="content">
    <button id="refresh" class="btn btn-block btn-primary">Refresh</button>
    <div id="clock" style="width:100%; text-align:center;"></div>
</div>
```

Enquanto que o código *JavaScript* é adicionado num ficheiro que tipicamente se denomina **app.js**. Neste caso, o código espera que seja enviado um elemento JavaScript Object Notation (JSON)[3] com dois atributos: **date** e **time**. O pedido é activado quando o elemento com identificador **refresh** receber um evento de **click**.

```
function refresh(){
    $.get("/time",function(response){
        var text=<h2>"+response.date+"</h2><br /><h2>"+response.time+"</h2>";
        $("#clock").html(text);
```

```
        });
    }

$( document ).ready(function() {
    $("#refresh").on("click",refresh);
});
```

---

Do lado do serviço, é necessário implementar um método que devolva a data e hora, como o seguinte.

```
import cherrypy
import time

class HelloWorld(object):
    @cherrypy.expose
    def index(self):
        # Método serve_file tb poderia ser utilizado
        f = open("index.html")
        data = f.read()
        f.close()
        return data

    @cherrypy.expose
    def time(self):
        cherrypy.response.headers["Content-Type"] = "application/json"
        return time.strftime('{"date":"%d-%m-%Y", "time":"%H:%M:%S"}').encode('utf-8')

cherrypy.server.socket_port = 8080
cherrypy.server.socket_host = "0.0.0.0"
cherrypy.tree.mount(HelloWorld(),"/", "app.config")
cherrypy.engine.start()
cherrypy.engine.block()
```

---

Note que alguns ficheiros são estáticos, enquanto outros são gerados dinamicamente. Os ficheiros estáticos encontram-se nos directórios **css** e **js**. Desta forma, é necessário criar um ficheiro chamado **app.config** com o seguinte conteúdo:

```
[/]
tools.staticdir.root = "/home/utilizador/directorio-do-servico"

[/css]
tools.staticdir.on: True
tools.staticdir.dir: "css"

[/js]
tools.staticdir.on: True
tools.staticdir.dir: "js"
```

---

### Exercício 20.6

---

Construa um exemplo que demonstre a comunicação entre uma aplicação Web e um serviço, através de JSON.

---

### Exercício 20.7

---

O exemplo anterior pode ser modificado de forma a devolver algo mais útil, tal como a distância para o estádio do SL Benfica (38.752667, -9.184711).

Usando a função seguinte componha um exemplo que envie a localização actual do cliente para o servidor, que responderá devolvendo a distância para o estádio.

---

```
from math import radians, cos, sin, asin, sqrt
...
def distance(lat, lon):
    lat1 = 38.752667
    lon1 = -9.184711

    lon, lat, lon1, lat1 = map(radians, [lon, lat, lon1, lat1]) # Graus -> rads

    dlon = lon - lon1
    dlat = lat - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat) * sin(dlon/2)**2 # Haversine
    c = 2 * asin(sqrt(a))

    km = 6367 * c # 6367=raio da terra

    cherrypy.response.headers["Content-Type"] = "application/json"
    return json.dumps({"distance": km})
...
```

---

## 20.5 Acesso a imagens

Através de APIs específicas é igualmente possível aceder às imagens armazenadas num dispositivo móvel ou mesmo activar a câmara e obter uma imagem. Estas podem depois ser processadas localmente ou enviadas para servidores. Este método recorre a um elemento `<input>`, especificando que se pretende aceder a fotografias.

---

```
...
<div class="content">
    <input type="file" accept="image/*"></input>
</div>
...
```

---

O resultado será que o dispositivo irá pedir ao utilizador que selecione o método de entrada, podendo este ser a câmara ou os documentos já existentes. Se se especificar o atributo `capture="camera"`, a câmara será activada imediatamente para que possa capturar uma imagem.

### Exercício 20.8

Integre o exemplo anterior numa aplicação e verifique o que acontece quando activa o input criado.

### Exercício 20.9

Adicione as classes `btn`, `btn-primary`, `btn-block` e `button-input`. Considere depois o seguinte excerto de Cascading Style Sheets (CSS)[4].

```
.button{
    text-align: center;
    color: #007aff;
    width: 137px;
}
.button:before{
    color: white;
    content: 'Usar Imagem';
    padding-right: 40px;
    margin-top: -20px;
    padding-left: 10px;
}
```

---

Componha a aplicação e teste.

Depois de estar selecionada a imagem, seria interessante poder visualizá-la. Isto pode ser feito se for activada uma função depois da fotografia ter sido seleccionada e existir um elemento onde a apresentar. O elemento neste caso será um `<canvas>`. Este elemento HTML é semelhante a um `<img>`, com a diferença que é possível desenhar para ele em tempo real, enquanto que um elemento `<img>` apresenta uma imagem estática. O novo HTML seria:

---

```
...
<input type="file" accept="image/*" onchange="updatePhoto(event);"></input>
<canvas id="photo" width="530" height="400">
...

```

---

O código *JavaScript* necessário corresponde à função `updatePhoto()` e irá aplicar a imagem capturada ao elemento `<canvas>`:

---

```
function updatePhoto(event){
    var reader = new FileReader();
    reader.onload = function(event){
        //Criar uma imagem
        var img = new Image();
        img.onload = function(){
            //Colocar a imagem no ecrã
            canvas = document.getElementById("photo");
            ctx = canvas.getContext("2d");
            ctx.drawImage(img,0,0,img.width,img.height,0,0,530, 400);
        }
        img.src = event.target.result;
    }

    //Obter o ficheiro
    reader.readAsDataURL(event.target.files[0]);
    sendFile(event.target.files[0]);
}
```

---

### Exercício 20.10

Componha uma aplicação que replique o exemplo anterior.

Eventualmente a imagem pode ser enviada para um servidor remoto. Será necessário criar código *JavaScript* para construir um pedido de envio (`POST`):

---

```
function sendFile(file) {
    var data = new FormData();
    data.append("myFile", file);
    var xhr = new XMLHttpRequest();
    xhr.open("POST", "upload");
    xhr.upload.addEventListener("progress", updateProgress, false);
    xhr.send(data);
}
```

---

```

function updateProgress(evt){
    if(evt.loaded == evt.total)
        alert("OK!");
}

function updatePhoto(evt){
    ...

    sendFile(image[0]);

    //Libertar recursos da imagem seleccionada
    windowURL.revokeObjectURL(picURL);
}

```

---

Do lado do servidor será depois necessário receber o ficheiro. Os ficheiros enviados são fornecidos ao serviço e é necessário copiar a informação para um local mais permanente. Isto porque os dados serão apagados assim que o método **upload** terminar. O código de exemplo será o seguinte:

```

@cherrypy.expose
def upload(self, myFile):
    fo = open(os.getcwd() + '/uploads/' + myFile.filename, 'wb')
    while True:
        data = myFile.file.read(8192)
        if not data:
            break
        fo.write(data)
    fo.close()

```

---

### **Exercício 20.11**

Crie um conjunto de aplicações que permita o envio de imagens para o servidor.

---

### **Exercício 20.12**

Altere o exemplo de forma a enviar outros ficheiros além de imagens.

---

## Glossário

**API** Application Programming Interface

**CSS** Cascading Style Sheets

<b>GPS</b>	Global Positioning System
<b>HTML</b>	HyperText Markup Language
<b>HTTP</b>	HyperText Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>SDK</b>	Software Development Kit

## Referências

- [1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee, *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616 (Draft Standard), Updated by RFCs 2817, 5785, 6266, Internet Engineering Task Force, jun. de 1999.
- [2] W3C. (1999). HTML 4.01 Specification, URL: <http://www.w3.org/TR/1999/REC-html401-19991224/>.
- [3] E. T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 7159, Internet Engineering Task Force, mar. de 2014.
- [4] W3C. (2001). Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification, URL: <http://www.w3.org/TR/2011/REC-CSS2-20110607/>.

# Representação de Som

## Objetivos:

- Representação de informação sonora
- Operações sobre som

## 21.1 Princípios de acústica

No mundo físico o som é transmitido através de ondas sonoras que são flutuações contínuas da pressão do ar ao longo do tempo e do espaço. Se a frequência de flutuação for alta, resulta um som agudo. Se a frequência for baixa, resulta um som grave. A amplitude da flutuação está relacionada com a *força* do som. Assim, um som muito fraco como um sussurro tem uma amplitude muito baixa, enquanto um som forte como um motor tem uma amplitude mais alta. A figura 21.1 mostra as flutuações de pressão de um som ao longo de um intervalo de tempo num certo ponto do espaço. O som representado é mais fraco e agudo no início, mais forte e grave no fim.

Um tom puro corresponde a uma variação de pressão que é uma função sinusoidal do tempo:  $\Delta p(t) = A \sin(2\pi f t)$ . Aqui,  $\Delta p(t)$  representa a variação de pressão do ar no instante  $t$ ,  $A$  é a amplitude da variação e  $f$  é a frequência da variação, indicada em ciclos por unidade de tempo.<sup>1</sup> A maioria dos sons, mesmo quando emitidos por instrumentos musicais, têm formas de onda mais complexas, mas que se podem sempre considerar como combinações de tons puros (sinusóides) de diferentes frequências e amplitudes (e diferentes desfasamentos). A combinação de diferentes frequências, com diferentes amplitudes para cada frequência, produz toda a multitudine de sons que somos capazes de reconhecer.

A Figura 21.2 demonstra as frequências emitidas por sete notas diferentes de um piano. O eixo horizontal representa o tempo, enquanto o vertical representa a frequência das

---

<sup>1</sup>A unidade de frequência é o Hertz (Hz) e corresponde a um ciclo por segundo:  $1\text{Hz} = 1\text{s}^{-1}$ .

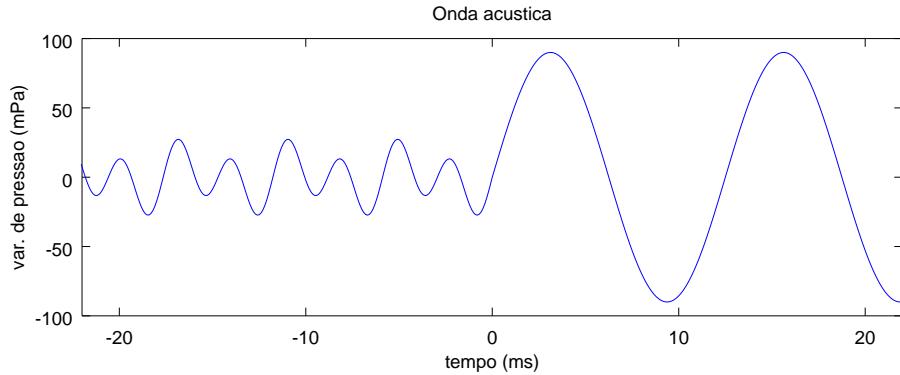


Figura 21.1: Forma de onda de um som medida ao longo do tempo num ponto do espaço.

componentes sinusoidais do som. Uma cor quente (vermelho/branco) num ponto da figura indica uma componente forte (amplitude alta) nessa frequência e instante de tempo. Cores frias (azul, cinza) indicam componentes fracas ou mesmo ausentes. Este tipo de representação chama-se um espetrograma. Como se pode ver, cada nota contém múltiplas componentes de frequências bem definidas (linhas horizontais), com amplitude suavemente decrescente ao longo da duração da nota. Isto é mais evidente na última nota, mais aguda, em que as componentes aparecem mais afastadas entre si, a primeira com  $f \approx 800\text{Hz}$  e as seguintes em frequências múltiplas dessa. Também se percebe, no início de cada nota, uma maior intensidade, mas mais espalhada por diversas frequências (linhas verticais), o que é característico de sons mais curtos e explosivos, que neste caso correspondem a componentes transitórias do som causadas pela ação de percussão dos martelos nas cordas.

### Exercício 21.1

Utilize o programa *Audacity* e analise o ficheiro **piano-c5-c6.wav** que foi fornecido pelos docentes. Em particular, experimente as diferentes formas de visualização do sinal, comutando entre forma de onda e espetrograma, por exemplo. Pode aceder a esta função se pressionar a seta que se encontra à direita do nome do ficheiro, do lado esquerdo da aplicação. Também pode selecionar um trecho curto de uma das notas e usar a função de *Analyze->Plot Spectrum* para ver o espetro desse segmento. Os picos no espetro indicam as frequências mais fortes presentes nesse trecho.

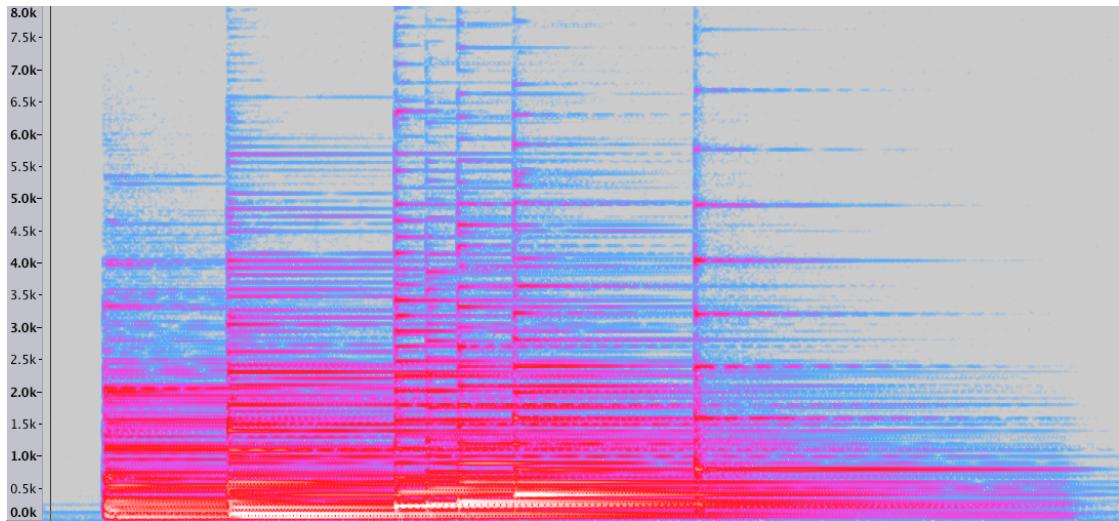


Figura 21.2: Várias notas de um piano capturadas num espetrograma.

## 21.2 Representação de informação sonora

Um microfone converte as variações de pressão em variações de tensão (ou de intensidade) de uma corrente elétrica. A tensão do sinal elétrico varia continuamente em função do tempo, de forma análoga à variação de pressão do sinal acústico. Por isso diz-se que é um *sinal analógico*.<sup>2</sup> Sistemas eletrónicos analógicos (formados por resistências, condensadores, transístores e outros componentes) permitem amplificar, processar e até armazenar sinais analógicos diretamente. Os rádios AM ou FM, as televisões antigas (não TDT), os gravadores de fita magnética de áudio (cassetes) ou vídeo (VHS), os giradiscos de vinil são exemplos de sistemas de transmissão, processamento e armazenamento puramente analógicos. O problema destes sistemas é que em cada passo de processamento, os sinais vão-se degradando com o acumular de pequenas distorções, interferências e ruído eletrónico.

Os sistemas digitais resolvem esse problema, mas para os usar, é preciso transformar os sinais analógicos em sinais digitais. Para isso, recorre-se a um conversor analógico-digital, ou Analog to Digital Converter (ADC), que é um dispositivo que faz duas operações:

1. tira amostras instantâneas do sinal a intervalos regulares (amostragem) e
2. mede a tensão de cada amostra, e converte-a num número binário com um número fixo de dígitos (quantização).

---

<sup>2</sup>Deveria talvez designar-se *sinal análogo*, mas *sinal analógico* está legitimado pelo uso.

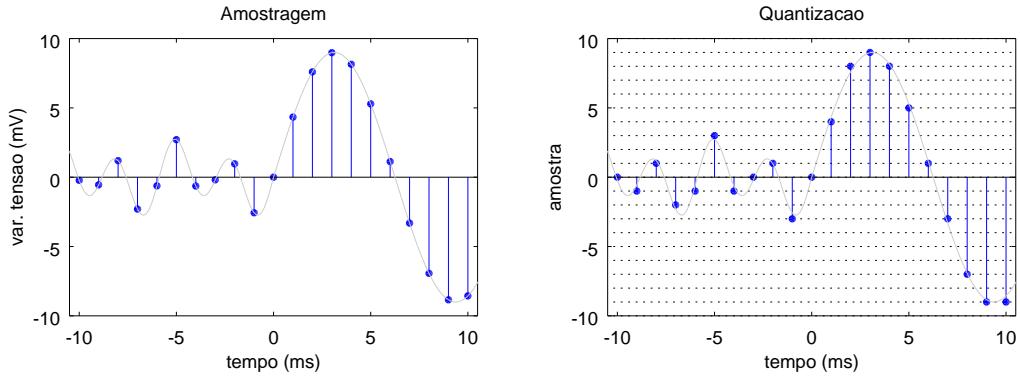


Figura 21.3: Conversão analógico-digital: amostragem e quantização. O sinal analógico original é representado em cinzento. As linhas verticais azuis representam os instantes de amostragem. A quantização aproxima as amplitudes por valores de um conjunto finito.

A figura 21.3 mostra estas duas operações aplicadas a um trecho do sinal analógico correspondente ao som da figura 21.1.

O sinal analógico que é uma função real, contínua no tempo, é assim convertido numa sequência de números inteiros. Este *sinal digital* resultante é portanto descontínuo no tempo e descontínuo em amplitude. Não consegue por isso representar a infinidade de detalhe que existe numa onda que varia continuamente ao longo do tempo, mas tem a vantagem de se poder armazenar, processar e transmitir virtualmente sem acumulação de degradação.

O número de amostras que uma ADC tira por segundo é chamada a sua *frequência de amostragem* e o número de bits usado em cada amostra é a sua *resolução*. As ADCs usadas atualmente para sinais áudio utilizam resoluções típicas de 8, 16 ou 24 bits e frequências de amostragem típicas de 8000, 11025, 22050, 44100, 48000 ou 96000Hz. Quanto maiores forem estes valores, mais precisa será a representação digital feita do som original. No entanto, também será necessário possuir um sistema mais veloz e mais espaço de armazenamento para a informação. Os valores ideais dependem das limitações da sensibilidade auditiva e da aplicação.

De acordo com teorema da amostragem de Nyquist, a frequência de amostragem deve ser, no mínimo, o dobro da máxima frequência do sinal registado para permitir a sua reconstrução exata. Como o ouvido humano detecta frequências até perto dos 20Khz, será necessária uma frequência de amostragem superior a 40000Hz para registar devidamente todas as frequências audíveis. Não é portanto surpresa que a frequência de amostragem típica para registo musical (CD, MP3) seja de 44100Hz (ou 48000Hz). Quando se pretende registar voz, como as componentes mais importantes da voz humana estão compreendidas entre 100 e 3000Hz, basta uma frequência de amostragem de 8000Hz, que

é um valor popular nas comunicações móveis.

Para reproduzir um som a partir de um sinal digital, é preciso fazer o processo inverso: usar um Digital to Analog Converter (DAC) para converter a sequência de números num sinal elétrico analógico; amplificar esse sinal e convertê-lo em ondas de pressão acústica através de um altifalante.

### 21.2.1 Armazenamento de som

Num computador o som é armazenado através da sequência de valores medidos nos instantes de amostragem. Um ficheiro sonoro pode registar uma sequência obtida de um único microfone ou pode ter várias sequências obtidas em simultâneo de vários microfones. Chama-se canal a cada sequência registada em simultâneo no mesmo ficheiro. São vulgares os ficheiros *monofónicos* (ou *Mono*), com um canal apenas, e os ficheiros estereofónicos (*Stereo*), com dois canais (esquerdo/direito), mas é possível ter ficheiros com 7 ou mais canais como é o caso dos ficheiros para os sistemas *Surround*.

A informação em si pode estar armazenada sob a forma de texto, mas tal só é comum em aplicações científicas. De resto espera-se que esteja armazenada numa forma binária, eventualmente comprimida. A compressão destina-se a reduzir os requisitos para o armazenamento de informação. Há métodos de compressão chamados *Lossless* (sem perdas), que permitem a recuperação exata da sequência de valores originais, e métodos de compressão *Lossy* (com perdas), que introduzem distorção nos sinais, mas fazem-no descartando ou alterando componentes menos perceptíveis pelo sistema auditivo humano. Esta é a abordagem seguida quando se cria um ficheiro *MP3*, por exemplo. Portanto, os sons guardados num formato *Lossy* não são iguais ao original, mas usando taxas de compressão razoáveis, soam-nos igual ao original. Durante este guião o foco serão os ficheiros não comprimidos como é o caso do formato WAVEform audio file format (WAVE) pois facilitam a manipulação da informação contida.

Os ficheiros WAVE são constituídos por um pequeno cabeçalho onde é possível indicar alguma meta-information, sendo este cabeçalho seguido de blocos com a informação sonora, geralmente no formato Linear Pulse Code Modulation (LPCM). A cada bloco dá-se o nome de *Frame*, e contém os valores registados para cada canal num certo instante de amostragem. Cada um dos valores numa *Frame* é uma amostra (*Sample*) de um dos canais e é codificada num número de bytes suficiente para a resolução usada. Por exemplo, considerando um ficheiro com dois canais, com resolução de 16 bits e frequência de amostragem de 44100Hz, cada segundo de som teria 44100 frames com 2 samples de 2 bytes cada, ou seja, um ritmo de 176400 octetos por segundo. Nestas condições, uma música de 5 minutos gera um ficheiro com um pouco mais de 50MB. Este formato está apresentado na Figura 21.4.

Em *Python* é possível inspecionar os metadados dos ficheiros WAVE e mesmo obter a informação sonora. Também é possível criar novos ficheiros, o que será efectuado na

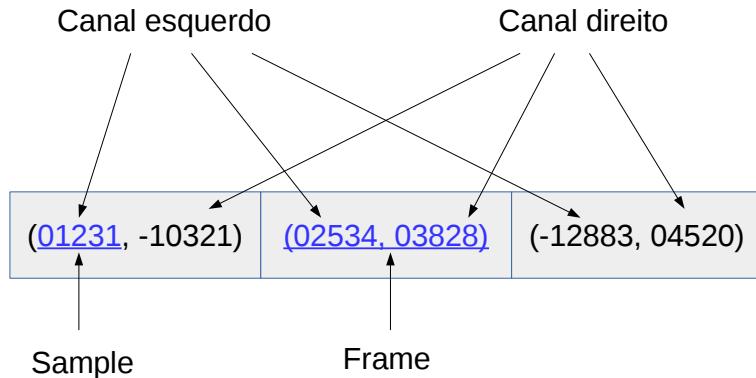


Figura 21.4: Estrutura dum ficheiro WAVE.

Subseção 21.2.2. Para isto é necessário utilizar o módulo `wave`<sup>3</sup> que pode ser instalado das formas usuais em *Python*. O exemplo seguinte demonstra como pode ser aberto um ficheiro WAVE e obtida informação do seu cabeçalho.

---

```

import wave
import sys

def main(argv):
    wf = wave.open(argv[1], "rb")
    print(wf.getnchannels())
    ...
    wf.close()

main(sys.argv)

```

---

### Exercício 21.2

Implemente um programa que obtenha a frequência de amostragem de um ficheiro, o tamanho de cada *Sample*, o número de canais e o número de *Frames* de som contidas no ficheiro.

Também é possível obter os dados sonoros e reproduzir o som directamente de forma programática. O módulo que permite isto possui o nome de **PyAudio** podendo ser

---

<sup>3</sup>ver <https://docs.python.org/3/library/wave.html>

instalado através de `pip` ou pelo gestor de pacotes da distribuição. O exemplo seguinte cria um `player` que pode ser utilizado para reproduzir um ficheiro WAVE.

---

```
import pyaudio
player = pyaudio.PyAudio()

...
stream = player.open(format = player.get_format_from_width(sample_width),
                      channels = nchannels,
                      rate = frame_rate,
                      output = True)

while True:
    data = wf.readframes(1024)
    if not data:
        break

    stream.write(data)

stream.close()
player.terminate()
```

---

### Exercício 21.3

Melhore o programa anterior de forma a que apresente informação de um dado ficheiro e o reproduza. Experimente modificar a variável `frame_rate` para um qualquer outro valor e volte a reproduzir o ficheiro.

#### 21.2.2 Geração de tons

Além da leitura de ficheiros WAVE, ou a sua construção através da leitura do microfone, também é possível a criação de sons através da sintetização das diversas frequências de uma forma matemática. Isto porque sendo um som composto por uma onda a oscilar numa frequência específica e com uma determinada amplitude, esta onda pode ser recriada através da função `sin` (seno) multiplicada por um factor de amplitude.

Para gerar um tom com frequência  $f$  é necessário criar uma sequência de valores através da expressão:

$$v(i) = amplitude * \sin\left(\frac{2 * \pi * freq * i}{rate}\right) \quad (1)$$

em que  $v(i)$  representa a amostra no instante  $i$ ,  $freq$  a frequência desejada e  $rate$  a frequência de amostragem do som (p.ex 44100Hz).

Aplicando a fórmula à linguagem *Python*, podem ser gerados ficheiros WAVE com tons puros da seguinte forma:

---

```
from struct import pack
from math import sin, pi
import wave
import sys

def main(argv):
    rate=44100
    wv = wave.open(argv[1], "w")
    wv.setparams((1, 2, rate, 0, "NONE", "not compressed"))

    amplitude = 10000
    data = []
    freq = 440
    duration = 1 # Em segundos
    for i in range(0, rate * duration):
        data.append(amplitude*sin(2*pi*freq*i/rate))

    # Gerar (pack) a informação no formato correto (16bits)
    wvData = []
    for v in data:
        wvData += pack("h", int(v))

    wv.writeframes(bytarray(wvData))
    wv.close()

main(sys.argv)
```

---

#### Exercício 21.4

Implemente o exemplo anterior e gere ficheiros com vários tons. Analise os ficheiros criados através da aplicação *Audacity*.

Podemos criar sons compostos somando tons de múltiplas frequências gerados em simultâneo. Por exemplo, para criar um som com duas componentes, uma a 440Hz e outra a 880Hz, podemos fazer:

---

```
...
freq_a = 440
freq_b = 880
for i in range(0, rate):
    data.append(
```

```

amplitude*sin(2*math.pi*freq_a*i/rate) +
amplitude*sin(2*math.pi*freq_b*i/rate)
)

```

---

### Exercício 21.5

Crie um novo programa baseado no anterior que gere um som composto por dois tons. Analise o resultado na aplicação *Audacity*.

A simplicidade deste método foi explorada em muitos sistemas, sendo que um dos mais famosos é o sistema Dual-Tone Multi-Frequency signaling (DTMF). Este sistema é utilizado para enviar algarismos e outros 6 símbolos através de ligações analógicas. Cada símbolo é codificado como um par de tons com certas frequências e enviado para o receptor. O receptor separa e deteta o par de frequências para descodificar o símbolo. A tabela de codificação é a seguinte:

	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

A figura 21.5 mostra o spectrograma de um número codificado no sistema DTMF.

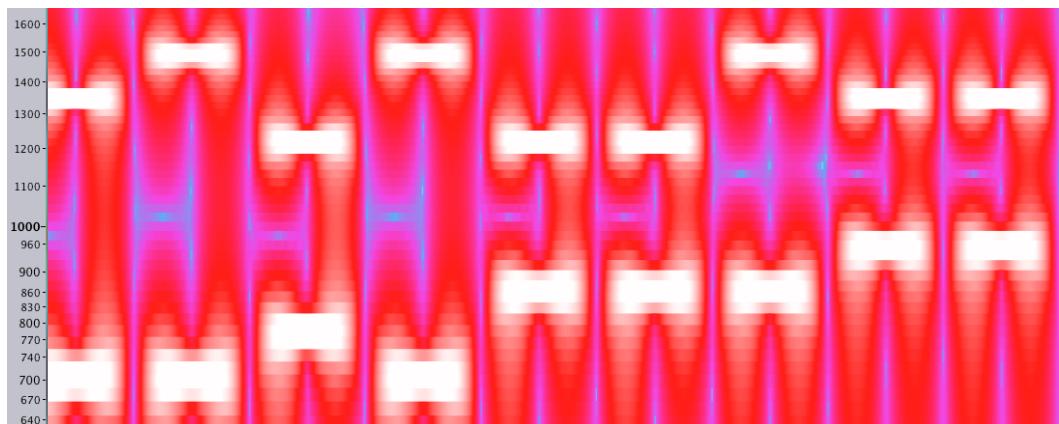


Figura 21.5: Número de telefone codificado em DTMF

### Exercício 21.6

Identifique qual o número de telefone representado na Figura 21.5.

Em Python uma maneira simples de implementar esta tabela seria através de um dicionário, em que cada símbolo possui uma lista com as frequências a utilizar.

```
...
tones = {\\"
    "1": (697, 1029), \
    "2": (697, 1336), \
...}
```

Isto pode depois ser utilizado para gerar sons DTMF, com duração de 40ms, seguidos de uma pausa de outros 40ms, tal como um telefone ou telemóvel actual fazem. O exemplo seguinte demonstra a estrutura básica de um programa para codificar qualquer número em DTMF.

```
...
tones = {...}
number = "" # número a codificar
for n in number:
    # Códigos DTMF
    for i in range(0, int(rate*0.040)):
        data.append(
            # Valores dos tons
        )
    # Pausa (silêncio)
    for i in range(0, int(rate*0.040)):
        data.append(
            # Silêncio
        )
...
...
```

### Exercício 21.7

Crie um programa que leia um número do teclado e gere um ficheiro com os códigos DTMF respectivos. Analise o resultado na aplicação Audacity.

### 21.3 Operações sobre som

São várias as operações que podem ser efectuadas sobre os ficheiros de som, além claro de os reproduzir. Nomeadamente é possível aplicar transformações, normalmente denominados de efeitos, que alterem as características sonoras da informação. Muitas das transformações são complexas, necessitando de conceitos mais complexos sobre o processamento de sinal. Alguns são bastante triviais ou relativamente simples de implementar, em particular os que operam sobre a informação numa perspectiva puramente temporal.

O seguinte trecho de código *Python* mostra uma forma bastante geral de aplicar uma qualquer transformação a um ficheiro WAVE, devolvendo o resultado noutro ficheiro do mesmo tipo. As secções seguintes deverão fazer uso deste programa, ou de um com estrutura semelhante para testar os efeitos desenvolvidos.

---

```
import wave
import struct
import sys
from struct import pack
import math

def copy(data):
    output = []
    for index,value in enumerate(data):
        output.append(value)
    return output

def main(argv):
    stream = wave.open(argv[1], "rb")

    sample_rate = stream.getframerate()
    num_frames = stream.getnframes()

    raw_data = stream.readframes( num_frames )
    stream.close()

    data = struct.unpack("%dh" % num_frames, raw_data) # "B" para ficheiros 8bits

    # Aplica efeito sobre data, para output_data
    i = 2
    output_data = []
    while i < len(argv):
        if argv[i] == "copy":
            output_data = copy(data)
        elif argv[i] == "foo":
            param = int(argv[i+1])
            output_data = foo(data, param)
            i += 1
        elif... #Outros filtros
```

```

    i += 1

    wvData = b""
    for v in output_data:
        wvData += pack("h", int(v))

    stream = wave.open("out-"+argv[1], "wb")
    stream.setnchannels(1)
    stream.setsampwidth(2)
    stream.setframerate(sample_rate)
    stream.setnframes(len(wvData))
    stream.writeframes(bytarray(wvData))
    stream.close()

if len(sys.argv) < 3:
    print("Usage: %s wave-file filter1 <params> filter2 <params> ..." % sys.argv[0])
else:
    main(sys.argv)

```

---

### Exercício 21.8

Crie um programa com o código anterior e verifique que consegue processar um ficheiro WAVE, criando uma cópia semelhante ao original. Use para isso um filtro chamado `copy`. Invoque o programa criado com a sintaxe: `python process.py file.wav copy`.

### Exercício 21.9

Adicione um filtro ao código anterior que devolva `reversed(data)` e verifique que consegue processar um ficheiro WAVE, criando uma cópia invertida do original. Use para isso um filtro chamado `reverse`. Invoque o programa criado com a sintaxe: `python process.py file.wav reverse`.

#### 21.3.1 Controlo de Volume

O volume a que o som é reproduzido depende essencialmente da amplitude do sinal, o que no caso de um ficheiro WAVE é representado pelo valor em absoluto de cada impulso. Para controlar o volume basta multiplicar todos os valores de amplitude por um factor multiplicativo. Se este factor for 0.5 o volume deverá ser diminuído em metade. Se for

2.0 o volume deverá ser multiplicado por 2.

#### Exercício 21.10

Implemente um filtro chamado **volume** que aceite um factor multiplicativo como parâmetro. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav volume 0.5`. Verifique o resultado obtido através da aplicação *Audacity*.

#### 21.3.2 Normalização

A normalização de um ficheiro diz respeito a controlar o volume de forma a que o valor máximo encontrado corresponda ao valor máximo possível. No caso de um ficheiro de 16bits, um ficheiro normalizado para o valor máximo deverá conter pelo menos um valor igual a -32768 ou a igual a 32767.

Este filtro necessita de dois passos de processamento. O primeiro determina qual o valor absoluto máximo dos valores. Daqui pode-se calcular um factor multiplicativo. O segundo passo aplica o factor multiplicativo tal como no caso do filtro de volume.

#### Exercício 21.11

Implemente um filtro chamado **normalize**. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav normalize`. Verifique o resultado obtido através da aplicação *Audacity*.

#### 21.3.3 Fade In-Out

Um filtro de *Fade* aplica um envelope progressivo à amplitude do som de forma a que o seu volume aumente ou diminua de forma contínua. A Figura 21.6 demonstra um som a que foi aplicado um *Fade In* e *Fade Out*.

A aplicação deste filtro é em tudo semelhante ao controlo de amplitude, com a diferença que o valor de amplitude é variável e só aplicado num intervalo temporal. Para ambos os casos é importante determinar onde iniciar e terminar a aplicação do efeito, que deve ter em consideração a frequência de amostragem do som. Também se deve ter em consideração qual o declive do factor multiplicativo a aplicar. Desta forma, o valor final do sinal será  $vf_i = vo_i * index * step$ . Em que  $vf_i$  representa o valor final  $i$ ,  $vo_i$  o valor original  $i$ ,  $index$  o número do *Sample* e  $step$  o tamanho de cada incremento ao longo do processo de *Fade*.

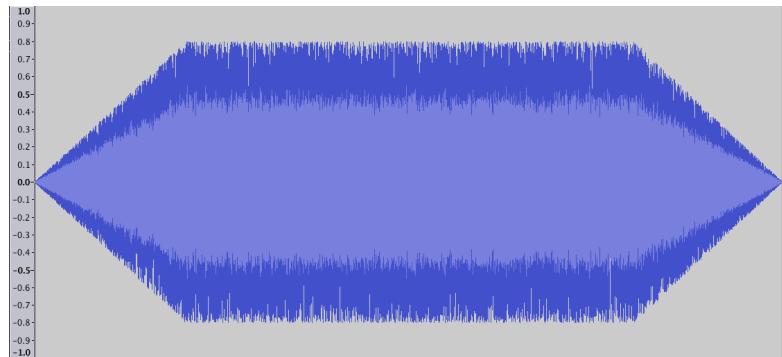


Figura 21.6: Exemplo de *Fade In* e *Fade Out*

O código Python seguinte demonstra o início de uma função aplicando este efeito:

---

```
def fadein(data, sample_rate, duration):
    time_start = 0
    time_stop = duration * sample_rate
    step = 1.0 / (sample_rate * duration)
    for index, value in enumerate(data):
        ...

```

---

### Exercício 21.12

Implemente um filtro chamado **fade-in** que aceite uma duração em segundos como parâmetro. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav fade-in 2`. Verifique o resultado obtido através da aplicação Audacity.

### Exercício 21.13

Implemente um filtro chamado **fade-out** que aceite uma duração em segundos como parâmetro. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav fade-out 2`. Verifique o resultado obtido através da aplicação Audacity.

Pode inclusive aplicar os dois efeitos usando: `python process.py file.wav fade-in 2 fade-out 2`

### 21.3.4 Máscaras

A aplicação de máscaras serve para omitir parte do conteúdo do som. É frequentemente utilizado para remover partes sensíveis, tal como palavras menos cuidadas. A operação deste filtro resume-se à substituição dos valores sonoros por outros no intervalo pretendido. Estes novos valores ( $vo_i$ ) podem ser o resultado de:

- uma sinusóide com uma frequência específica (um tom):  $vo_i = amplitude * \sin(\frac{2 * \mathit{math.pi} * freq * i}{rate})$
- silêncio:  $vo_i = 0$
- valores aleatórios:  $vo_i = random.randint(-32768, 32767)$

Este filtro pode ter como parâmetro o tipo de máscara a aplicar, o instante de tempo inicial e o instante de tempo final para aplicação do efeito. Tal como no caso do filtro anterior é necessário calcular em que *Sample* iniciar a máscara e em que *Sample* terminar a máscara.

#### Exercício 21.14

Implemente um filtro chamado `mask` que aceite como parâmetro um tipo de máscara com os valores `silence`, `noise`, `tone`, um instante de início e uma duração em segundos. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav mask silence 2 2`. Verifique o resultado obtido através da aplicação Audacity.

### 21.3.5 Modulação

A modulação tem como princípio multiplicar um dado som por um outro tom. Considerando  $vf(i)$  o valor final,  $vo(i)$  o valor original,  $freq$  uma frequência de modulação e  $rate$  uma frequência de amostragem, o filtro pode ser concretizado com:

$$vf(i) = vo(i) * \sin\left(\frac{2 * \mathit{math.pi} * freq * i}{rate}\right) \quad (2)$$

Uma variante deste efeito, mas considerando um tom variável, é o *Wah Wah* aplicado a guitarras e voz. O resultado de usar um tom fixo é um som com aspecto metálico, como

se tivesse sido emitido por um robot num filme de televisão. Se a frequência for muito baixa o resultado é apenas uma variação cíclica no volume do som original.

### Exercício 21.15

Implemente um filtro chamado **modulate** que aceite como parâmetro uma frequência em *Hertz*. Deve conseguir invocar o programa desenvolvido através da sintaxe: **python process.py file.wav modulate 3000**. Verifique o resultado obtido através da aplicação *Audacity*.

#### 21.3.6 Atraso

O atraso é normalmente chamado de *Reverb* ou *Echo* e consiste na repetição de um sinal emitido num instante  $t_i$ , para um instante  $t_i + x$  mas com uma amplitude ligeiramente inferior. Devidamente aplicado este efeito confere profundidade ao som, simulando as ondas refletidas naturalmente quando um som é reproduzido numa sala.

Considerando uma lista **output** que irá conter o resultado final, e uma lista **data** que contém o som original, pode-se construir este filtro através da seguinte estrutura:

```
output = [0] * len(data)+tdelay
...
for index,value in enumerate(data):
    output[index] += value
    output[index+tdelay] += value * amount
```

Neste caso o valor **tdelay** consiste no número de *Samples* que se pretende atrasar o som (consideram-se valores na ordem de 0.5-1.5 segundos), e **amount** consiste na força do efeito. Considera-se um valor inferior a 1. Um aspeto interessante deste efeito é que ele pode ser aplicado de forma recursiva, sendo que em cada nova iteração o efeito deve ser aplicado mais tarde e ter menos força.

Considerando que o efeito se encontra implementado numa função chamada **delay**, pode-se implementar este princípio da seguinte forma:

```
def delay(data, sample_rate, amount, delay):
    if amount < 0.05:
        return data
    ...
    ...
```

```
#Repetir com 80% da força e com 20% mais de atraso.  
return delay(output, sample_rate, amount * 0.8, delay * 1.2)
```

---

### Exercício 21.16

Implemente um filtro chamado `delay` que aceite como parâmetro um atraso segundos e uma força. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav delay 0.8 0.6`. Verifique o resultado obtido através da aplicação *Audacity*.

---

### Exercício 21.17

Implemente a versão recursiva deste filtro e avalie a complexidade do efeito resultante, em comparação com a versão mais simples.

---

#### 21.3.7 Esteganografia

Os ficheiros de som, em particular no formato WAVE também permitem a aplicação de técnicas de esteganografia. Em particular é simples codificar mensagens através da manipulação do último bit de cada *Sample*. Pode-se considerar que para codificar um valor 0, o valor do último bit deverá ser 0, sendo 1 para codificar um valor 1. Quando aplicado com a linguagem *Python*, é possível codificar um texto em *Samples* sonoros através da seguinte estrutura:

---

```
def steg_add(data, message):  
    bitstream = "" # Irá conter uma string. ex: "011101010"  
    for c in message:  
        bitstream += format(ord(c),2)  
  
    output = []  
    encoded_bit = 0  
    for index, value in enumerate(data):  
        ....
```

---

A descodificação é muito semelhante sendo que um dado carácter `c` (neste caso o primeiro) pode ser obtido de uma *String* com os bits fazendo: `c = chr(int(decoded_bits[0:8],2))`.

Normalmente é útil adicionar um marcador para sinalizar a existência e/ou final de uma

mensagem, separadores ou mesmo códigos de detecção de erros.

### Exercício 21.18

Implemente um filtro chamado `steg_add` que aceite como parâmetro uma mensagem. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav steg_add 'mensagem de teste'`. Verifique o resultado obtido através da aplicação Audacity.

### Exercício 21.19

Implemente um filtro chamado `steg_get`. Deve conseguir invocar o programa desenvolvido através da sintaxe: `python process.py file.wav steg_get` e este deverá imprimir a mensagem previamente escondida.

## 21.4 Para aprofundar

### Exercício 21.20

Considere as notas musicais podem ser reconstruídas a partir de uma nota inicial segundo a seguinte fórmula:  $freq_n = 2^{n/12} * 440$ . Relembre que as notas são: La, La#, Si, Do, Do#, Re, Re#, Mi, Fa, Fa#, Sol, Sol#, La, La#, Si, Do..., correspondendo estas notas a valores de  $n$  entre 1 e 16.

Implemente um programa que leia uma sequência de notas e as reproduza (p.ex: 1 1 8 8 10 10 8)

### Exercício 21.21

Melhore o programa anterior de forma a considerar duração das notas e pausas entre as notas. Pode fazê-lo através de um carácter como o '-' para indicar a duração e 0 para indicar uma pausa. Uma nota Do# com um terço da duração ficaria 2-3 (Do-um\_terço) e uma pausa com um quarto da duração normal de uma nota ficaria 0-4 (Pausa-um\_quarto).

### **Exercício 21.22**

Desenvolva outros efeitos a aplicar a informação sonora, nomeadamente:

- Expansor: Sons menores com uma amplitude menor que  $x$  são aumentados para o máximo de amplitude. Os restantes mantêm-se inalterados.
- Compressor: Possui dois intervalos  $x_{max}$  e  $x_{min}$ . Amplitudes superiores a  $x_{max}$  são iguais a  $x_{max}$ , enquanto amplitudes inferiores a  $x_{min}$  são iguais a  $x_{min}$ .
- Limitador: Limita a amplitude a um valor. Ou seja, se o valor for superior a  $x$ , este passa a  $x$ .

## Glossário

<b>ADC</b>	Analog to Digital Converter
<b>DAC</b>	Digital to Analog Converter
<b>DTMF</b>	Dual-Tone Multi-Frequency signaling
<b>LPCM</b>	Linear Pulse Code Modulation
<b>WAVE</b>	WAVEform audio file format