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

TEMA 12

Programação em Python

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

12.1 Linguagem Python

A linguagem *Python***python** é uma das linguagens de uso geral mais populares e com uma larga documentação**python.doc** Considerada como sendo 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 aspetos de baixo nível. Programas escritos em *Python* são tipicamente compactos (considerando o número de linhas), especialmente em comparação com outras linguagens como a *Java* ou a *C*. Estas funcionalidades tornam a linguagem *Python* ideal para a prototipagem rápida de sistemas, o desenvolvimento de sistemas complexos e, através dos módulos que possui, a implementação de algoritmos dedicados a cálculo numérico intensivo.

Um aspeto importante da linguagem é que é considerada como sendo multiparadigma. Enquanto *Java* assume obrigatoriamente a utilização de classes, uma caraterística da

Programação Orientada a Objetos (POO), a linguagem *Python*, suporta o paradigma POO mas não obriga a este, permitindo muitos outros, tal como a programação funcional ou imperativa.

Visto ser uma linguagem interpretada, não são necessários os passos de compilação e ligação a bibliotecas, tal como acontece noutras linguagens. Isto por outro lado possibilita que o mesmo código seja facilmente executado em múltiplos sistemas operativos, ou que partes de um sistema sejam alterados dinamicamente a quando da execução de uma aplicação[1].

No contexto de Laboratórios de Informática, *Python* será a linguagem principal para a exploração de conceitos relacionados com a informática. Os pontos seguintes são o primeiro passo nessa direção, introduzindo a utilização de *Python* para a resolução de problemas comuns.

12.2 Características Básicas

Python é uma linguagem com algumas 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.
- **Tipos Dinâmicos** - Uma variável não é declarada com um tipo específico, sendo que o seu tipo depende do valor que tem armazenado. O tipo de uma variável pode igualmente modificar-se dinamicamente, bastando para isso a atribuição de um objeto de um tipo diferente.
- **Alto Nível** - Não são expostos detalhes da plataforma de computação, tal como registos, ponteiros, ou endereços a memória. O programador foca-se na implementação de um algoritmo e não nos detalhes do *hardware*.
- **Com gestão de memória** - A alocação e libertação de memória é gerida de forma automática, não sendo este detalhe exposto para o 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

[1] A alteração de um ficheiro não implica qualquer recompilação.

o conteúdo de um programa *Python* que imprime para o ecrã a frase "Laboratórios de Informática":

```
print "Laboratórios de Informática"
```

Uma forma alternativa de o fazer, sera através do módulo **sys**, no formato:

```
import sys

sys.stdout.write("Laboratorios de Informatica\n")
```

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

Em qualquer um dos casos a sua execução é conseguida através do comando:

```
python ./hello.py
```

O interpretador de *Python* não necessita de ser invocado exclusivamente para interpretar programas em ficheiros. É perfeitamente comum invocar o interpretador para depois escrever nele código *Python*, especialmente quando se pretendem testar pequenas instruções. O exemplo seguinte demonstra como se poderia imprimir a mesma frase, mas sem existir qualquer programa de *Python*. Repare que o interpretador apresenta um *Prompt* tal como a **bash**, neste caso com o formato **>>>**. Para terminar o interpretador pode ser utilizada a sequência **CTRL+D**.

```
user@host $~> python
Python 2.7.5 (default, Aug 25 2013, 00:04:04)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Laboratórios de Informática"
Laboratórios de Informática
>>> ^D
```

A linguagem *Python* tem uma carta de princípios que definem tanto o desenvolvimento da linguagem, como os programas que a utilizam. A esta carta dá-se o nome de *The Zen of Python* e pode ser consultado através do próprio interpretador da linguagem

executando **`import this`**.

Exercício 12.1

Execute o interpretador de comandos *Python* e insira **`import this`**. Deverá ser-lhe apresentada a carta de princípios da linguagem. Leia cada linha. Ao desenvolver aplicações em *Python* deverá ter em consideração estes conceitos.

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, funções e classes, com a implementação do algoritmo necessário. Um aspeto importante é que não existem delimitadores de blocos explícitos tal como na linguagem *Java*. Em vez disso, assume-se que cada instrução ocupa uma linha e a indentação dessa linha define a que bloco pertence. O pseudocódigo a seguir apresentado demonstra esta caraterística da linguagem.

```
# encoding=codificação

# Zona de importação de módulos
import modulo1
import modulo2

# Funções
def funcao(a):
      print a
      if a:
            print "SIM"
      return False

# Outras instruções
funcao(True)
print "d"
```

A linha 6 do código define uma função (isto será abordado na Secção 12.2.6), e a linha 7 pertence a essa função apenas porque possui uma indentação superior. A linha 9 possui uma indentação ainda maior, indicando que pertence a um sub-bloco, neste caso uma condição. Por sua vez a instrução na linha 10 já não pertence à condição. De notar que neste caso a função é declarada mas só executada quando invocada, na linha 13. Também não existe qualquer indicador de fim de instrução, sendo que esta corresponde a toda uma linha.

No pseudocódigo apresentado também se pode verificar que existem comentários, sendo estas linhas iniciadas pelo caráter #. Em *Python* não existe a noção de comentários sobre múltiplas linhas, pelo que cada linha de comentário tem de iniciar pelo caráter #.

12.2.2 Variáveis

As variáveis são declaradas no momento em que são necessárias e atribuídas a um dado valor. De notar que em *Python* todas as variáveis são sempre referências, não existindo a distinção entre variáveis primitivas e não primitivas que existe na linguagem *Java*. O exemplo seguinte declara duas variáveis **a** e **b** e imprime a soma de ambas.

```
a = 3
b = 5.0

print a + b
```

De notar que **a** será uma variável inteira e **b** uma variável real. Na adição o impacto é nulo, mas não numa divisão. O exemplo seguinte possui código que pode ser utilizado para exemplificar este aspeto, demonstrando como o tipo de uma variável muda dinamicamente.

```
a = 3
b = 5.0

print a / b
b = 5
print a / b
```

O resultado deverá ser:

```
0.6
0
```

Exercício 12.2

Crie um ficheiro *Python* com o número deste exercício e verifique o resultado de adicionar, subtrair, multiplicar a dividir diferentes valores. Pode começar pelo exemplo anterior de forma a verificar como o operador de divisão se comporta.

Strings

Um dos tipos mais utilizados para além dos tipos matemáticos é talvez o tipo *String*. Em *Python* ele define um conjunto com um ou mais caracteres. Isto aplica um dos princípios da linguagem *Python* que diz *Special cases aren't special enough to break the rules*. Neste caso, assume-se que o caso especial de uma *String* com um caráter não merece um tipo novo, assumindo-se que é também uma *String*.

A declaração de uma variável do tipo *String* faz-se atribuindo um nome a uma sequência de carateres, iniciados e terminados por aspas ou plicas. O exemplo seguinte demonstra a declaração de três variáveis do tipo *String*, a sua concatenação e impressão.

```
# encoding=utf-8

a = "Laboratórios"
b = " de "
c = "Informática"

print a+b+c
```

Um aspeto muito importante é que se existirem caracteres acentuados, devem-se utilizar *Strings* numa codificação *Unicode*, através da inclusão de um caráter `u` antes da *String*. Isto é particularmente importante para a lingua Portuguesa. A não especificação da codificação como sendo *Unicode* irá criar problemas em funções que manipulem *Strings*. Portanto, seria mais correto fazer:

```
# encoding=utf-8

a = u"Laboratórios"
b = u" de "
c = u"Informática"

print a+b+c
```

O exemplo seguinte elabora um pouco mais a utilização destas variáveis, demonstrando como aceder a pedaços da informação. De notar que a sintaxe é semelhante ao modo como são acedidos os *arrays* em *Java*, mas permitindo também selecionar sequências.

```
# encoding=utf-8

a = u"Laboratórios"
b = u" de "
c = u"Informática"
```

```
# Imprime Lab-I 2014
print a[0:3]+"-"+c[0]+" "+str(2014)
```

Exercício 12.3

Repita 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*. Nomeadamente de forma a determinar o seu comprimento, converter para maiúsculas ou minúsculas, retirar espaços, etc... Para uma lista completa, consultar `http://docs.python.org/2/library/stdtypes.html#string-methods`.

```
# encoding=utf-8

a = "   Laboratórios de Informática 2014 "

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 ainda encontradas na referência da linguagem *Python*.

Não existindo a função **printf**, é ainda assim formatar a apresentação. Neste caso o funcionamento baseia-se na formação de variáveis do tipo *String* compostas por um formato e uma série de valores. Usa-se o caráter % para separar ambos as partes da formatação. O funcionamento da especificação de formato é em tudo semelhante à encontrada noutras linguagens. O exemplo seguinte produz a linha "O Benfica ganhou 4-0 ao Porto" a partir de uma formatação de *Strings*.

```
equipa_casa = "Benfica"
equipa_fora = "Porto"
resultado = "ganhou"
golos_equipa_casa = 4
golos_equipa_fora = 0

print "%s %s %d-%d ao %s" % (equipa_casa, resultado, \
      golos_equipa_casa, golos_equipa_fora, equipa_fora)
```

Exercício 12.5

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

Por vezes é necessário converter *Strings* em valores numéricos, tais como inteiros ou reais 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()`** e **`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, %4.2f") % (a,sa,b,c)
```

Listas

As listas são tipos de dados com algumas semelhanças aos *arrays* usados noutras linguagens, sendo que a linguagem *Python*, nativamente, não possui a noção de *array*. Estas estruturas são compostas por um conjunto de valores, armazenados de forma ordenada, podendo ser acedidos através de um índice. Este índice tem origem em 0 e final em **`len(array) - 1`**. O exemplo seguinte demonstra a declaração 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', 'LTSI', '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** insere um novo elemento na lista, enquanto o método **extend** permite extender uma lista com outra. A lista criada no exemplo anterior poderia ser refeita através destes métodos da seguinte forma[2]:

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

l1.append('MIECT')
l1.append('LTSI')

l2.append('MSI')
l2.append('MIEET')

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

print len(l)
```

Em qualquer altura, a dimensão de uma lista pode ser obtida recorrendo à função **len()**,

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

Um caso importante de listas é a que é utilizada para fornecer ao programa os argumentos passados pela invocação na linha de comandos. Esta lista (**sys.argv**) obtém-se através da utilização do módulo **sys**. Esta lista contém todos os argumentos passados

[2]Embora estes métodos sejam úteis, não existe qualquer vantagem nesta implementação, sendo preferido o método anterior.

ao programa, sendo que o primeiro valor da lista é o nome do ficheiro com o código *Python*. Um programa que imprima os valores dos argumentos que lhe são passados seria o seguinte:

```
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 existe a 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 situações pois permitem ter uma estrutura em que o acesso é efetuado através de uma chave com significado para o programador. A sintaxe básica de um dicionário é a seguinte:

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

Em que o acesso ao dicionário se faz da 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 considera 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

Existem diferente modos de fornecer ou obter informação a programas implementados em *Python*. Um dos métodos de saída de dados já abordados foi a utilização da função **`print`** que imprime texto para a consola.

A leitura de dados do teclado pode ser efetuada através da função **`input("mensagem")`** para valores numéricos, ou através de **`raw_input("mensagem")`** para *Strings*. Estas funções imprimem para o ecrã o texto passado como parâmetro e esperam pela introdução de uma linha de texto. Esta linha poderá depois ser convertida para inteiro ou real através das funções **`int()`** e **`float()`**.

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: %f * %f = %f" % ( 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 Condições

As instruções de condições, através das palavras chave **if** e **else**, permitem executar um conjunto de instrução caso uma ou várias condições de verifiquem. O modo de funcionamento da condição é em tudo semelhante a outras linguagens de programação, tal como descrito no exemplo seguinte. No entanto, deve-se ter em consideração que não existe delimitador de bloco, sendo a indentação o que define 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
```

De notar que uma instrução **if** pode conter várias condições, utilizando para isso operadores matemáticos tais como: **and**, **or**, ou **not**. Estes operadores são escritos exactamente nesta forma. O exemplo seguinte imprime "Sim" se um valor for divisível por 2 e 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 aspeto um pouco diferente de outras linguagens é que a condição é avaliada por qualquer tipo de dados. Por exemplo, uma condição pode avaliar diretamente uma variável inteira, sem ser necessário converter a mesma para booleana. Neste caso, considera-se como **True** qualquer valor diferente de 0, ou que a variável tem um tamanho superior a 0 (ex, *String*), e **False** qualquer valor igual a 0, ou a variável está vazia (ex, *String*). 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()`.

12.2.5 Ciclos

Os ciclos permitem iterar sobre um conjunto de valores, ou sobre um conjunto de elementos, um de cada vez, ou de outra forma executar 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** - Itera sobre elementos de um conjunto, ou valores de um intervalo, executando um conjunto de instruções a cada iteração. Por exemplo, pode iterar sobre carateres de uma *String*, ou sobre todos os valores entre 1 e 10.
- **while** - Executa repetidamente um conjunto de instruções enquanto uma variável for verdadeira.

O exemplo que se segue descreve a implementação de um ciclo **`for`**. De notar que é utilizada a função **`range`** e não é realizada qualquer aritmética para incrementar valores (ex, **`i++`**). Isto deve-se ao facto que de a instrução **`for`**, por definição, itera sobre um conjunto de valores. A função **`range`** tem como objetivo gerar um conjunto com valores entre dois limites indicados como parâmetros. A variável **`i`** terá depois cada um dos valores do conjunto 0,1,2,3,4...9.

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

A execução da instrução **`range`** pode ser analisada em detalhe se se executar o exemplo seguinte. O resultado deverá ser uma lista de valores entre 0 e 9 (10 - 1).

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

Exercício 12.13

Utilizando um ciclo **for**, implemente um programa que imprima uma sequência de Fibonacci. Uma sequência de Fibonnaci 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 variável do tipo *String* irá iterar sobre todos 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.

O ciclo **while** difere do ciclo **for** pois não itera sobre um conjunto de valores. Em vez disso, repete instruções enquanto uma condição for verdadeira. Um ciclo que apresenta os valores entre 0 e 9. Neste caso é necessário inicializar e decrementar a variável **i** de forma a esta variar o seu valor.

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

Exercício 12.17

Repita o exercício anterior mas 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 noutros modos de acesso, seria necessário utilizar os especificadores:

- **`"a"`** - Modo de adição. Escritas são adicionas ao final do ficheiro.
- **`"r"`** - Modo de leitura. Não é possível escrever.
- **`"r+"`** - Modo de leitura e escrita. Se o ficheiro não existir ele é criado.
- **`"w"`** - Modo de escrita. Se o ficheiro não existir ele é criado.

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 na sua integra para a memória do computador deve ser sempre evitada! Favorece-se sempre a leitura parcial 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()
```

De notar que o ciclo **`for`** permite iterar por quase qualquer elemento, o que se inclui um ficheiro. 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, é realmente um ficheiro, se existem permissões, 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 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 10 notas necessárias (R1,R2,R3,R4,MT1,MT2,MT3,MT4,P1 e P2), calcular e apresentar 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 $[0; 100]$[a], 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).

[a] import random
random.randint(0,100)

Glossário

POO Programação Orientada a Objetos