



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 aspetos 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 procedimental.

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.¹

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')
```

¹A 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 aspeto 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, suportando *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 seleccionar 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

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*. Nomedamente, 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}, {:.4.2f}'.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 acedidos 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:²

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

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

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

```
l2.append('MIEET')

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

print(len(l1))
```

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 caracteres a expressar algumas condições não parece compensar o que se perde em termos de legibilidade e compreensibilidade 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 caracteres 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 calcule 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:

- `'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 $[0; 100]$,^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

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.