universidade de aveiro

deti departamento de electrónica, telecomunicações e informática

Fundamentos de Programação | Python

Gonçalo Matos | MEC 92972
Licenciatura em Engenharia Informática
1.º Semestre | Ano letivo 2018/2019

Índice

1. Introdução à programação

O programador e o programa

capacidade fundamental de um programador

ser capaz de resolver problemas

algoritmo

uma lista de instruções passo por passo que se seguida exatamente como descriminada resolve o problema em consideração

Linguagem de alto nível

alto nível e baixo nível

Linguagens de baixo nível têm as suas instruções escritas em código binário, permitindo assim uma execução imediata por parte dos computadores. Por outro lado, as de alto nível necessitam de compiladores ou interpretadores para serem executadas.

No entanto, as últimas são mais fáceis de serem escritas, compreendidas pelo programador e ainda portáteis, não necessitando de grandes modificações para serem lidas em equipamentos com características diferentes, o que não acontece nas linguagens de baixo nível.

Compilador

compilação

um compilador lê um programa em linguagem de alto nível (**source code**) e traduz-lo para baixo nível, passando este a chamar-se **object code** ou **executabe**.

uma vez traduzido, este pode ser executado repetidamente sem mais traduções

Interpretador

Modos de execução

Interativo

Execução imediata dos comandos
prompt indicador de que o interpretador está pronto par executar código

Script

Armazenamento dos comandos num ficheiro (.py), que posteriormente poderá ser executado por um interpretador

Prompt

>>> Indicador de que o interpretador está pronto para receber instruções

Compilador vs interpretador

Diferenças o **compilador** traduz todo o **source code** e produz o **object code** ou o **executable** de uma só vez, enquanto que o **interpretador** o faz linha a linha

Ficheiros Python

.py Extensão indicativa de que o ficheiro é um programa Python

Programa

O que é um programa? Um programa é uma sequência de instruções que o computador consegue compreender e executar

Erros

bug erro de programação

debugging deteção de correção de erros

sintax error código que não respeita a sintaxe da linguagem de programação

são tipicamente identificados pelo **compilador/interpretador**

exemplo esquecer uma vírgula no final de uma expressão onde é necessária

runtime error ou exceptions só ocorrem quando o programa é executado

são tipicamente identificados pelo **compilador/interpretador**

exemplo dividir um número por 0 (impossível em Python)

semantic error o programa pode correr sem mensagens de erro, mas existem erros de programação que fazem com que o programa não cumpra com os objetivos para os quais foi criado, ou seja, este não faz exatamente o que foi criado para fazer

são tipicamente identificados pelo **programador**, quem sabe qual o objetivo para o qual o programa foi criado e consegue identificar facilmente se este foi comprido

exemplo esquecer de multiplicar por 100 quando queremos converter em percentagem

Programação vs debugging

Qual a diferença? programar é escrever um programa e gradualmente **debugging** um programa, até que este cumpra com o seu objetivo

Linguagens naturais vs formais

Linguagem natural São linguagens como as que falamos, que não foram criadas por pessoas, tendo evoluindo naturalmente

Linguagem formal É uma linguagem criada por pessoas para um propósito específico, por exemplo a notação matemática é uma linguagem formal.

Regras de sintaxe **Tokens (símbolos)** elementos básicos da linguagem, como letras, palavras, números ou elementos químicos

Estrutura a maneira como os símbolos são organizados , por exemplo 3+=6 não é uma estrutura correta em matemática formal

Parsing Quando lemos uma frase na nossa língua ou uma instrução numa determinada linguagem de programação, temos de decifrar o que a estrutura da frase é, a este processo chamamos **parsing.**

Mas qual a diferença entre as duas? **Ambiguidade** enquanto que as linguagens naturais estão cheias de ambiguidades, uma linguagem formal é desenhada para ser isenta destas (ou quase totalmente isenta), tentando que cada instrução tenha um único significado, independentemente do contexto

Redundância para compensar a ambiguidade e reduzir mal-entendidos, as linguagens naturais utilizam a redundância, abundando em palavras. Por outro lado, as formais são menos redundantes e mais **concisas.**

Sentido literal ao contrário das linguagens naturais, cheias de metáforas, as formais significam exatamente o que está escrito

Programa O significado de um programa é literal e não ambíguo, podendo ser entendido na integra a partir de uma análise aos seus símbolos e estrutura (as suas regras de sintaxe).

Tipos de instruções

input	recebe dados do teclado, ficheiro ou outro dispositivo
output	mostra dados no ecrã ou envia dados para ficheiro
assigment	armazena valores em variáveis, para usar posteriormente
math	realiza operações matemáticas básicas
conditional execution	verifica o valor de verdade de certas condições e executa as instruções apropriadas
repetition	realiza alguma ação repetidamente, normalmente com alguma variação

2. Estrutura

Estrutura

keywords — palavras reservadas que o interpretador usa para reconhecer a estrutura do programa

operadores — símbolos especiais que representam cálculos (+, -, *, /, %, <=, or)

operandos — valores combinados por operadores

Expressão *expressions* — combinação de valores, variáveis e operadores. Para serem executadas, têm de ser avaliadas, sendo depois exibido o resultado

Instrução *statement* — unidade de código que um interpretador de Python consegue executar (existem vários tipos de instrução, de **atribuição, while, for, if and import statements,** etc.)

nota uma expressão tem um valor (mesmo que *None*) enquanto que uma instrução não

Indentação

Indentação — Blocos de código são denotados por indentação, sendo o número de espaços entre diferentes elementos variável, mas elementos do mesmo bloco devem ter o mesmo.

Linhas e parágrafos

/ — Indica a mudança de linha (desnecessário, uma vez que o Python faz esta mudança por definição)

; — Permite instruções múltiplas na mesma linha

Precedência

precedência — em Python, aplica-se a precedência em expressões com operadores por ordem decrescente de prioridade:

1. Parêntesis
2. Exponenciação (da direita para a esquerda)
   2**2**3 = $(2^2)^3$ = 2^{2^3} = 2^8 = 256
3. Multiplicação e divisão (da esquerda para a direita)

4. Soma e subtração (da esquerda para a direita)

Tipos (ou classes) de dados (para determinados valores)

O valor (por exemplo um número ou uma palavra) é uma das coisas fundamentais que um programa manipula.

Para serem determinados pode ser utilizada a função **type(var)**

tipos numéricos

int	número inteiro
float	número decimal
complex	número complexo (x+yj, x,y ∈ IR e j=i^2)

sequências

str	texto
list	lista
tuple	tuplo

boleano

bool	Boleano (valor de True ou False)

nota para determinar a classe de um determinado valor, basta usar a instrução **type(var)** no interpretador

nota2 não é possível concatenar valores de classes diferentes, sem antes as converter por exemplo print("O número é o" + str(23))

2.1 Statements | Instruções

Definição no início do capítulo “estrutura”

Import Statement I Instrução de importação

import math
math.sin(pi/2)

um módulo contém um conjunto de funções que podem ser invocadas depois da sua importação

Assigment Statement I Atribuição de Variáveis

Atribuição de variáveis

Instruções de atribuição de variáveis não só criam novas variáveis como também lhes atribuem valores

Reatribuição de variáveis

Se uma variável for definida mais que uma vez ao longo do programa, a que é considerada é a última a ter sido definida

Símbolo (token) de atribuição

=

Nomes das variáveis

Var ≠ var (distinção entre maiúsculas e minúsculas)

NUNCA pode ter espaços (em alternativa, podem ser utilizados *underscores* “_”)

Podem conter números e letras, mas NUNCA podem começar por números

Não podem conter símbolos matemáticos (+, -, etc.)

Execução condicional
Sempre em minúscula

```
if 'condição':
    print(…)
else: / elif 'condição':
    print(…)/pass (para ignorar else)
```

Expressão boleana

```
n=2
maioridade=n<5
maioridade 'enter'
>>>True
```

3. Debugging

Erros mais comuns

Geralmente os erros são indicados na linha seguinte à onde este se encontra.

Nunca podem ocorrer depois da linha indicada, uma vez que a execução do programa termina nesta.

Dica Utilizar print statements para verificar valores das variáveis.

ParseError

São como erros gramaticais, normalmente associados à falta de sinais de pontuação, como parêntesis, vírgulas

TypeError

Ocorre quando se tentam combinar dois elementos não combináveis (através de operações), como por exemplo uma variável do tipo *int* com um do tipo *string*.

NameError

Acontece quando uma variável é utilizada antes de ser declarada. Geralmente este tipo de erros está associado ao erro de escrita de uma variável quando a utilizamos depois de a termos definido, ou mesmo de uma função.

Por exemplo_ escrever imt() em vez de int(); chamar a variável telefone escrevendo com erros (telfone)

ValueError

Ocorre quando queremos introduzir um parâmetro numa função e este não é compatível com ela. Por exemplo convertermos um input vazio utilizando a função int(), vai dar erro do tipo ValueError porque a função int() funciona apenas para parâmetros numéricos.

4. Turtle Graphics

Breve introdução

Turtle é um módulo do Py. O seu nome tem origem nas tartarugas (animais), pois quando estes se movem a sua cauda deixa uma marca na areia (seu habitat natural), desenhando assim figuras.

A utilização deste módulo consiste em "dizer" à tartaruga para onde se mover e quando levantar ou não a cauda.

Por definição quando é criada a turtle está virada para este (direita) e encontra-se nas coordenadas (0,0).

```
//EXEMPLO BÁSICO
import turtle                # allows us to use the turtle library by importing it.
                               This has all the built in functions for drawing on the screen
                               with the Turtle object
wn = turtle.Screen()         # creates a graphics window
alex = turtle.Turtle()       # create a turtle named alex
                               The first "turtle" (before the period) tells Python that we are
                               referring to the turtle module, which is where the object
                               "Turtle" is found.
alex.forward(150)            # tell alex to move forward by 150 units
alex.left(90)                # turn by 90 degrees
```

Métodos

Um objeto pode ter vários métodos (coisas que podemos fazer com ele), por exemplo avançar, virar à esquerda…

```
//ALGUNS MÉTODOS
wn.exitonclick()                # wait for a user click on the canvas
wn.window_width()               # gives the value of the window width
wn.window_height()               # gives the value of the window width
object.left()                   # makes the object turn left
object.forward()                # makes the object go forward
//NOTA Se o argumento do left ou forward for negativo, este será executado como o método inverso, espetivamente right e backward
object.up()
object.forward(x)
object.down()
#Estas três instruções "levantam a cauda" do objeto enquanto este se desloca, fazendo então mover a caneta sem esta escrever
object.shape()                   # each object can have it's own shape
//Shapes available are: arrow, blank, circle, classic, square, triangle, turtle
object.speed()                   # defines the speed o drawing between1(slowest) and 10(fastest)
object.stamp()                  # after this instruction all the movements of the pen instead on
                                   drawing lines will stamp the object shape
```

Atributos

O objeto pode ter ainda atributos, como por exemplo a posição da *turtle* dentro da janela, para que lado está virada, a cor de fundo da janela, etc. O atributos referem-se então ao **state** dos elementos, seja do objeto, seja da janela.

```
//ALGUNS ATRIBUTOS
wn.bgcolor("lightgreen")          # set the window background color
object.color("blue")              # make object blue
object.pensize(3)                 # set the width of the pen
object.pos()                      # returns a tuple [x,y]
object.xcor()                     # returns the value of the x coordenate
object.ycor()                     # returns the value of the y coordenate
```

Um rebanho de tartarugas

Podemos na mesma função trabalhar com mais do que um objeto, tendo apenas de os definir com nomes diferentes, de acordo com as regras de nomenclatura do Py (capítulo 2).

```
//REBANHOS DE TARTARUGAS
alex = turtle.Turtle()      # create a turtle named alex
john = turtle.Turtle()      # create a turtle named john
maria = turtle.Turtle()      # create a turtle named maria
```

O ciclo for e a função range()

A função range é bastante útil para criar um ciclo for com vista a gerar figuras geométricas cuja construção exige a repetição de instruções n vezes.

5. Modules

Nota// Este capítulo do livro ainda não foi analisado, resulta apenas de um apanhado da matéria presente noutros capítulos.

Matemáticos | math/random module

math.	pow(x,y)	exponenciação x^y
	max(x, y, …, z)	devolve valor máximo da lista (x, y, …, z)
random.	randrange(x,y)	Devolve um número aleatório do intervalo [x,y[
	randrange(x,y,z)	Devolve um número aleatório do intervalo a começar em x, com incrementação de z até atingir y (ver range(x,y,z))

Desenho | turtle module

Ver capítulo 4

6. Functions | Funções

Introdução

Uma função é uma sequência de instruções com um nome.

Não são necessárias a um programa, mas ajudam na organização do mesmo, permitindo a utilização dessas instruções mais do que uma vez e evitando a necessidade da repetição de código.

Definição de funções

```
def name( parameters ):
    statements
```

O nome da funçãonão pode ser uma *keyword* do Py.

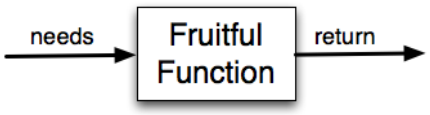
Os parâmetros da função podem ser nulos: **def name():**

Não devendo neste caso ser esquecidos os parêntesis, porque mesmo que "vazios" são indispensáveis para chamar a função!

Para uma função ser executada esta tem de ser "chamada", chamamos a isto **invocação da função**.

Funções que devolvem valores

Nem todas as funções devolvem valores. Por exemplo uma função para desenhar um quadrado com base na função *turtle* não devolve nenhum valor, no entanto funções como *range* ou *int* têm como objetivo devolver um valor.



Funções que devolvem valores são chamadas de **fruitful functions**, enquanto que às outras chamamos **procedures**.

```
//EXEMPLO DE UMA FRUITFUL FUNCTION
def square(x):
    y = x * x
    return y

toSquare = 10
result = square(toSquare)
print("The result of", toSquare, "squared is", result)
```

Se não utilizarmos o *return*, a função vai devoler por definição o valor **None**.

Instruções escritas **depois do return não são executadas**, uma vez que a função termina onde este se encontra.

Unit testing

Utilizar a função test.testEqual(x,y) é um bom princípio para verificar o funcionamento da função (**debugging**).

Variáveis das funções são locais

Variáveis utilizadas dentro das funções (**variáveis locais**), mas cujos valores não são devolvidos (*return*), não podem ser invocadas fora da função (**variáveis globais**), pois não vão estar definidas (NameError).

É também um mau princípio utilizar variáveis globais dentro das funções, uma vez que as variáveis que queremos "introduzir" na função devem ser escritas nos parâmetros da mesma.

Para executar uma função o Py procura primeiro o nome das variáveis no **local scope** (variáveis locais) e só se não as encontrar é que analisa o **global scope** (variáveis globais). Podemos por isso ter variáveis locais com o mesmo nome das globais (na função serão consideradas as locais), mas é também considerado um mau princípio.

Scope é o intervalo de instruções onde variáveis podem ser acedidas.

Funções que chamam funções I Composition

Funções podem chamar funções (**composition**), não devendo ser esquecidos as variáveis locais que devem ser inseridas nos parâmetros da função a ser chamada!

Fluxo de execução

Quando um programa com funções é executado o Py lê apenas a primeira linha da função (def function()) e o conteúdo da função à frente, sendo este apenas lido quando a função é chamada mais tarde no corpo do programa.

Main functions

Em alternativa a ter a estrutura: importar módulos, funções e "corpo do programa", podemos colocar o corpo do programa dentro de uma **main function**, que depois chamamos para que o programa seja executado. É necessário noutras linguagens de programação e possível em Py, mas não necessário.

Nota Quando o Py executa um programa, uma das variáveis que é definida automaticamente é `__name__`, que terá o valor *string* `__main__` se o programa estiver a ser executado por si só, o que não acontece se este for importado por outro programa.

Funções do Python

Input | Entrada de dados

input("Texto de instrução") — A função input permite a entrada de dados no programa por parte do utilizador

Tem um argumento principal chamado de *prompt* e devolve um valor do tipo string

Como por defeito os valores obtidos são do tipo string, para obter outros tipos de dados, como numéricos, estes devem ser convertidos int(input("Texto de instrução"))

Print | Saída de dados

print ("Texto") — para se escrever um determinado valor no ecrã, utiliza-se a função print

para escrever múltiplas linhas, adiciona-se \n
linha1\nlinha2 ou aspas triplas (""Texto c/ paragrafo""")

para escrever diferentes valores, separados por espaços, utilizamos vírgula "texto",var1,"texto"

Conversão de valores

print("Idade: " + str(varidade)) — A expressão print não consegue concatenar valores de diferentes classes, pelo que estes devem ser convertidos para que a mesma ocorra

Nota ao converter um float para int, o python não arrendonda, truncando simplesmente as casas decimais (por ex. 53,7 fica 53)

Espaços na instrução 'Print'

print('texto',var,'texto') — texto var texto

print('texto',var,'texto',sep='---') — texto---var---texto

Preenchimento de espaços

print("Texto { }".format(var1)) — Os espaços { } vão ser preenchidos, respetivamente pelas variáveis var1 e var2.

message="Texto {} "
print(message.format(var1)) — **Nota** {:2d} e {:.2f} garantem que números do tipo int e float, respetivamente, tenham dois algarismos

Parágrafos

print('texto', end="") — Desta forma, não será feito parágrafo automaticamente após a escrita.

String format

Bases

“{}, {}”.format(“texto1”,texto2”)	texto1 texto2
“{1}, {0}”.format(“one”,”two”)	two one

Alinhar texto

“{:>10}”.format(“texto”)	alinhar à direita
{:10}	alinhar à esquerda
{:^10}	alinhar ao centro

Números

{:d}	integers
{:f}	floats
{:.2f}	duas casas decimais

Listas

List=[name1,name2,name3] “{d[0]},{d[2]}”.format(d=list)	name1, name3

Type | Avaliação de dados

type(var)	Identifica qual a classe (ou tipo de dados) do valor da variável

Len | Número de caracteres

len(var)	Indica o número de caracteres da variável em consideração

Matemáticas | Outras

abs(x)	devolve o valor absoluto de x
rango(stop)	

range(start, stop, step)	devolve lista de caracteres desde zero (inclusive) até stop (exclusive) devolve lista começando em start (inclusive), avançando de step em step até stop (exclusive)

Testes | Test module

test.	testEqual(x,y)	Devolve **Pass** ou **Test Failed**, ao comparar se as expressões x e y têm o mesmo valor

7. Selection | Seleção

Valores e expressões boleanas

Valores boleanos são um tipo de dado (**bool**) que representa o valor de verdade, em Python, **True** ou **False**.

Como Py é uma linguagem **case sensitive**, é fundamental que estes sejam escritos com inicial maiúscula.

Um **valor boleano** é então um **valor de verdade**.

Uma expressão boleana é uma expressão que compara dois valores entre si.

```
//OPERADORES DE COMPARAÇÃO MAIS COMUNS

x == y                  # x is equal to y
x != y                  # x is not equal to y
x > y                   # x is greater than y
x < y                   # x is less than y
x >= y                  # x is greater than or equal to
y
x <= y                  # x is less than or equal to y
```

Operadores lógicos

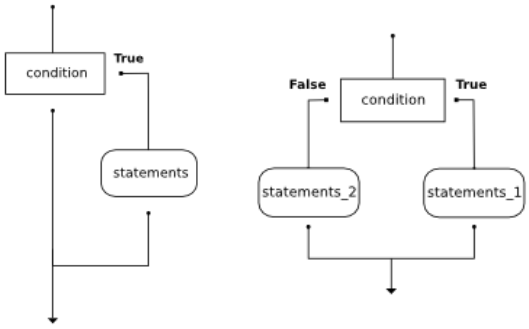
```
and                     # conjunção
or                      # disjunção
not                     # negação
```

Precedência dos operadores

LEVEL	CATEGORY	OPERATORS
7(HIGH)	exponent	**
6	multiplication	*,/,//,%
5	addition	+,-
4	relational	==,!=,<=,>=,>,<
3	logical	not
2	logical	and
1(LOW)	logical	or

Execuções condicionais

Seleção unitária e binária I if/else statement




```
if (condition):
        instructions
elif (condition):
        instructions
else (condition):
        instructions
```

Seleção unitária (if) vs seleção binária (if/else)

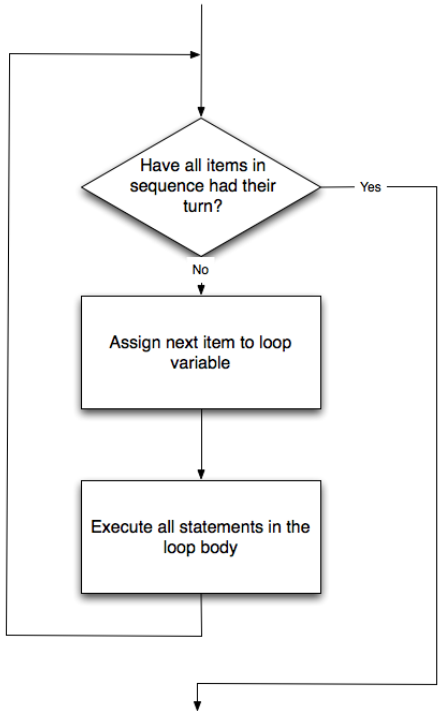
Podemos ainda fazer estruturas encadeadas (if's dentro de if's), mas é mais correto a utilização de estruturas em cadeia (if/elif/else).

8. Mais sobre iteração

Iteração é a execução repetida de uma sequência de instruções.

Ciclos

Repetição n vezes | Ciclo **for**

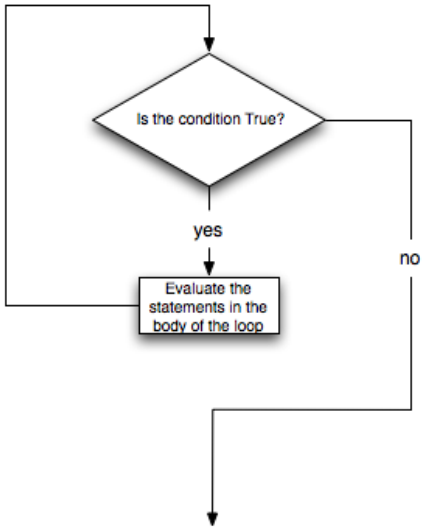



```
//ESTRUTURA

for var in list:

      instructions
```

```
//EXEMPLO 1

for name in ["Joe", "Amy", "Brad", "Angelina",
"Zuki", "Thandi", "Paris"]:

    print("Hi", name, "Please come to my party on
Saturday!")
```

```
//EXEMPLO 2

for x in range(0, 3):
    print "We're on time %d" % (x)
```

Repetição até | Ciclo **while**




```
//ESTRUTURA

while (comparison):

      instructions
```

```
//EXEMPLO 1

n = 10
answer = 1
while n > 0:
    answer = answer + n
    n = n + 1
print(answer)
```

Nota

Enquanto que o termo *while* em inglês aplicado à frase "while X, do Y" dá a entender que assim que X deixar de se verificar, deixamos de fazer Y, em Python, a verificação é realizada apenas uma vez por cada repetição do bloco. Portanto, se a condição for verdadeira no início da execução do bloco e deixar de o ser na primeira linha do bloco, as restantes linhas serão executadas até ao fim, sendo só depois feita uma nova verificação do valor de verdade da condição X.

Sentinel value

Quando criamos um ciclo *while* e o queremos repetir até que o utilizador nos diga quando parar, criamos um **sentinel value**, um valor que quando introduzido pelo utilizador vai quebrar o ciclo de repetição do ciclo.

Por exemplo num programa de *checkout* de uma loja online adiciona os valores do produto introduzidos pelo utilizador até que este introduza o valor 0, que quebrará o ciclo. Ver programa Aula 05\thinkpy_c8.8_carrinho de compras.py

for vs while

Uma vez que o ciclo **for** é executado um número limitado e determinado de vezes pelo *range*, este é chamado de **definite iteration**.

Por outro lado, o ciclo **while** depende de uma condição cujo valor tem de ser falso para que a sua execução termine. Como não sabemos necessariamente quando este vai terminar, chamamos-lhe **indefinite iteration**.

Nota

Qualquer ciclo **for** pode ser substituído por um ciclo **while**.

Embora o **while** utilize uma sintaxe diferente, é tão poderoso quanto o **for** e mais flexível.

Criando tabelas simples

//CARACTERES ESPECIAIS

```
\t                  #tab chacacter

\n                  #newline
```

//DIFERENCE BETWEEN TABS AND SPACES

A tab will line up items in a second column, regardless of how many characters were in the first column, while spaces will not.

//NOTA
Ao usarmos o print, o newline (\n) não é necessário, uma vez que a função o faz automaticamente!

//EXEMPLO 1

```
print("n", '\t', "2**n")     #table column
headings
print("---", '\t', "-----")

for x in range(13):         # generate values for
columns
    print(x, '\t', 2 ** x)
```

//RESULTADO

```
n           2**n
---         -----
0           1
1           2
2           4
3           8
4           16
5           32
6           64
7           128
8           256
9           512
10          1024
11          2048
12          4096
```

Processamento de imagens

Matéria não lecionada nas aulas

9. String Methods

Method	Parameters	Description
upper	none	Returns a string in all uppercase
lower	none	Returns a string in all lowercase
capitalize	none	Returns a string with first character capitalized, the rest lower
strip	none	Devolve uma string sem espaços nos extremos e caracteres especiais removidos.
lstrip	none	Returns a string with the leading whitespace removed
rstrip	none	Returns a string with the trailing whitespace removed
count	item	Returns the number of occurrences of item
replace	old, new	Replaces all occurrences of old substring with new
center	width	Returns a string centered in a field of width spaces
ljust	width	Returns a string left justified in a field of width spaces
rjust	width	Returns a string right justified in a field of width spaces
find	item	Returns the leftmost index where the substring item is found, or -1 if not found
rfind	item	Returns the rightmost index where the substring item is found, or -1 if not found
index	item	Like find except causes a runtime error if item is not found
rindex	item	Like rfind except causes a runtime error if item is not found
format	substitutions	Involved! See String Format Method, below

11. Files | Ficheiros

Neste capítulo vamos abordar ficheiros de texto, existindo, no entanto, outros tipos de ficheiros.

Introdução

Em Python, devemos de **abrir** um ficheiro para que o possamos usar e depois da sua utilização devemos **fechá-lo**.

METHOD NAME	USE	EXPLANATION
OPEN	`open(filename,'r')`	Open a file called filename and use it for reading. This will return a reference to a file object.
OPEN	`open(filename,'w')`	Open a file called filename and use it for writing. This will also return a reference to a file object.
CLOSE	`filevariable.close()`	File use is complete.

Trabalhar com ficheiros do disco

Para trabalhar com um ficheiro que não se encontre na pasta do programa Py (neste caso basta indicar o seu nome), é necessário indicar ao programa qual o "caminho" para descobrir o documento com o qual queremos trabalhar, o seu **path**.

```
//EXEMPLOS DE PATHS
hello.txt                                   #file in the same directory as the program
/Users/yourname/hello.txt                   #file path in Unix Systems
C:\Users\yourname\My Documents\hello.txt    #file path in Windows
```

Abrir o ficheiro

```
//COMO ABRIR UM FICHEIRO
fileref = open("qbdata.txt", "r")
statements
fileref.close()
```

Iteração na leitura de ficheiros

```
//CICLO FOR
fileref = open("qbdata.txt", "r")
for line in fireref:
        statements
fileref.close()

#Melhor método para ler ficheiro
linha a linha
```

```
//CICLO WHILE
fileref = open("qbdata.txt", "r")
while True:
      line = fileref.readline()     # returns line to the end
      if line == "": break          # empty means end-of-file
      print(line)
fileref.close()
```

```
//CICLO WHILE
fileref = open("qbdata.txt", "r")
line = fileref.readline()
while line:
    values = line.split()
    print('QB ', values[0], values[1], 'had a rating of ', values[10] )
    line = infile.readline()
infile.close()
```

Ler um ficheiro sem o abrir nem fechar

```
//MÉTODO WITH
with open(file_name, mode) as fileobj:
          statements
```

Caracteres especiais

Para indicar a mudança de linha, um tab ou outro tipo de formatação de texto, em Py utilizam-se caracteres especiais.

```
//PRINCIPAIS CARACTERES ESPECIAIS
\n                  #new line
\t                  #tab
```

Métodos de leitura e escrita

METHOD NAME	USE	EXPLANATION
WRITE	`filevar.write(astring)`	Add astring to the end of the file. filevar must refer to a file that has been opened for writing.
READ(N)	`filevar.read()`	Reads and returns **a string of n characters**, or the entire file as a single string if n is not provided.
READLINE(N)	`filevar.readline()`	**Returns the next line** of the file with all text up to and including the newline character. If n is provided as a parameter than only n characters will be returned if the line is longer than n. **Devolve uma string vazia quando o ficheiro termina.** **Nota //** Se utilizarmos este método uma vez, quando o voltarmos a utilizar ele vai devolver a próxima linha e não a primeira do ficheiro!
READLINES(N)	`filevar.readlines()`	**Returns a list of strings**, each representing a single line of the file. If n is not provided then all lines of the file are returned. If n is provided then n characters are read but n is rounded up so that an entire line is returned.

Escrever em ficheiros

Quando utilizamos o método open("file name", "w"), se o file name não existir, este será criado. Caso já exista, será substituído por um ficheiro vazio que será preenchido pelo programa, dependendo das instruções que este tiver.

É importante adicionarmos caracteres de quebra de linha no final de cada escrita, caso contrário vamos estar sempre a adicionar informação na mesma linha!

Ao utilizarmos o método **write** temos de converter valores que não são do tipo string para este formato!

```
//ALTERNATIVAS AO MÉTODO WRITE
/FORMAT
fname.write("{:s} costs {:.2f}.".format("coffee",x))
/PRINT
print("X:", x, file=fname)
print("{:s} costs {:.2f}.".format("coffee",x), file=fname))
```

Navegar num ficheiro

Existem alguns métodos que nos permitem avançar na leitura de um ficheiro, de modo a não termos de começar a leitura pelo início. Não nos é prático, uma vez que estes métodos trabalham com a localização em *bytes* e é rara a vez em que sabemos em que *byte* começa uma linha.

```
//MÉTODOS DE NAVEGAÇÃO
file.seek(offset)            #this method changes the current file position to
offset bytes from the start
file.tell()                  #returns the current position in bytes
```

13.Exceptions I Exceções

```
//EXEMPLO

try:
  # Your normal code goes here.
  # Your code should include function calls which might raise exceptions.
except Exception_one:
  # If Exception_one was raised, then execute this block.
except Exception_two:
  # If Exception_two was raised, then execute this block.
else:
  # If there was no exception then execute this block.
finally:
  # This block of code will always execute, even if there are exceptions
raised
```

Código | Notas

Comentários

\# iniciam-se com um cardinal, tornando-se a continuação dessa linha num comentário, ou seja, é ignorada aquando da execução do programa

Configuração de texto

Aspas ou pelicas?

“” | ‘’ | ‘’’ | “”” pode estar entre aspas, pelicas, ou até pelicas triplas (‘’’)

quando colocado entre “””aspas triplas””” pode ter linhas múltiplas

Operadores em Python (distintos)

// Divisão inteira 18//4=4

% Resto da divisão inteira 18%4=2

** Exponenciação 3**2=9