
Introdução à Programação e Resolução de Problemas

com Python

Ernesto Costa

Versão 1.0
2 de Junho de 2013

Ao G.M. que há longos anos me faz acreditar que os rios também podem nascer no mar.

Conteúdo

1	Introdução	1
1.1	Computadores	1
1.2	Linguagens	4
1.3	A linguagem Python	17
1.4	Programas	19
1.5	O meu primeiro programa	20
1.6	Fazer escolhas	26
1.7	Repetir	27
1.8	Intermezzo	29
1.9	Módulos	29
1.10	Adivinhar	32
1.11	Modo não interactivo	35
	Sumário	37
	Teste os seus conhecimentos	38
	Exercícios	39
2	Visões (I)	45
2.1	Coisas que mexem	45
2.2	Tartarugas e geometria	48
2.3	Intermezzo	52
2.4	Gráficos de funções	56
	Sumário	60
	Teste os seus conhecimentos	62
	Exercícios	62
3	Objectos (I)	65
3.1	Generalidades	65
3.2	Números	70

3.3 Booleanos	82
3.4 Cadeia de Caracteres	85
3.5 Range	101
3.6 Tuplos	104
3.7 Intermezzo	114
3.8 Mutabilidade	116
Sumário	120
Teste os seus conhecimentos	121
Exercícios	122
4 Instruções Destrutivas	127
4.1 Generalidades	128
4.2 Atribuição	130
4.3 Leitura	136
4.4 Escrita	138
Sumário	141
Teste os seus conhecimentos	142
Exercícios	142
5 Instruções de Controlo	149
5.1 Introdução	149
5.2 Sequências	153
5.3 Condicionais	154
5.4 Ciclos	162
5.5 Intermezzo	167
5.6 Qual o valor de π ?	169
5.7 Outras Instruções de controlo	177
5.8 Excepções	182
Sumário	187
Teste os seus conhecimentos	187
Exercícios	187
6 Objectos Estruturados	195
6.1 Introdução	195
6.2 Listas	195
6.3 Dicionários	214
6.4 Mais exemplos	230
Sumário	232
Teste os seus conhecimentos	233
Exercícios	238

7 Ficheiros	239
7.1 Generalidades	239
7.2 Leitura	242
7.3 Escrita	247
7.4 Navegar	248
7.5 Intermezzo	251
7.6 Exemplo	253
7.7 De um ficheiro de palavras a um dicionário de frequências	262
7.8 Outros Tipos de Ficheiros	263
Sumário	271
Teste os seus conhecimentos	271
Exercícios	272
8 Metodologia da Programação	279
8.1 Introdução	279
8.2 Um Problema Simples	280
8.3 Uma questão de consenso	293
8.4 Protecções	302
8.5 Novo exemplo: quadrado mágico	305
Sumário	316
Teste os seus conhecimentos	316
Exercícios	316
9 Visões (II)	319
9.1 Introdução	319
9.2 Representação de imagens	321
9.3 O módulo cImage	325
9.4 Exemplos Básicos	327
9.5 Manipulações simples	333
9.6 Intermezzo: abstracção	338
9.7 Exemplos complementares	340
9.8 Filtros	349
9.9 O formato de imagem PPM	358
Sumário	361
Teste os seus conhecimentos	362
Exercícios	362
10 Recursividade	371
10.1 Conceitos	371
10.2 Exemplos	376
10.3 Exemplos Complementares	398

10.4 Quando usar?	403
Sumário	407
Teste os seus conhecimentos	407
Exercícios	407
A O módulo <code>turtle</code>	415
A.1 Introdução	415
A.2 Tartaruga	415
A.3 Caneta	420
A.4 Janela	422
A.5 Eventos	423

Listas de Tabelas

1.1	Modelo PCAP	17
3.1	Literais Numéricos	70
3.2	Operações básicas sobre números	73
3.3	Operadores de conversão de tipos	76
3.4	Operadores ao nível do Bit	80
3.5	Operadores Booleanos	82
3.6	Operadores de Comparação	83
3.7	Literais Para Cadeias de Caracteres	86
3.8	Caracteres de controlo em cadeias de caracteres	87
3.9	Operações básicas Para Cadeias de Caracteres	89
3.10	Operações adicionais para cadeias de caracteres	93
3.11	Métodos das cadeias de caracteres	94
3.12	Operações para tuplos	105
4.1	Palavras Reservadas	129
5.1	Representações de Falso	153
5.2	O que escolher? Preços em euros.	188
6.1	Literais para Listas	197
6.2	Operações sobre Listas	197
6.3	Métodos Pré-Definidos para Listas	208
6.4	Literais para dicionários	219
6.5	Operações sobre Dicionários	221
6.6	Métodos Pré-Definidos para Dicionários	226
7.1	Modos de abertura de ficheiros de texto	241
7.2	Operações de leitura com ficheiros	242

7.3	Operações de escrita com ficheiros	247
7.4	Operadores de navegação	249
9.1	Imagens e tamanhos	321
A.1	Movimentos	416
A.2	Orientação	416
A.3	Desenho	418
A.4	Estado da tartaruga	419
A.5	Visibilidade	419
A.6	Aspecto da tartaruga	419
A.7	Medidas	420
A.8	Estado da caneta	420
A.9	Cor	422
A.10	Preenchimento	422
A.11	Específico da Janela	423
A.12	Controlo da janela	423
A.13	Eventos	423

Listas de Figuras

1.1	Arquitectura de um computador	2
1.2	Compilação	3
1.3	Interpretação	4
1.4	Interpretação: Python (Adaptado de [1])	5
1.5	O processo de comunicação humano - computador	6
1.6	A unidade de processamento central	7
1.7	A unidade de controlo	8
1.8	Programa em Assembly	9
1.9	Modelo PCAP	16
1.10	O ciclo lê - avalia - escreve.	21
1.11	Objectos e seus atributos	22
1.12	A função peso: conhecido o valor de altura podemos calcular o peso.	24
1.13	Associação entre um nome e uma definição	25
1.14	Instruções	30
1.15	Entrada e saída de objectos	30
1.16	De cartesianas a polares	42
2.1	O passeio (curto) da tartaruga	46
2.2	Mas que lindo quadrado	49
2.3	Um quadrado defeituoso	50
2.4	Uma bela circunferência	53
2.5	Duas tartarugas independentes	55
2.6	De costas voltadas	56
2.7	A função seno	58
2.8	Seno e coseno	61
2.9	Uma estrela	62
2.10	Sem rei nem roque	63

2.11 Tanto quadrado...	64
2.12 Parece um nautilus...	64
3.1 Ambiente, nomes e objectos	67
3.2 Tipos básicos	71
3.3 Conversão automática de tipos	77
3.4 Cadeia de caracteres: indexação	90
3.5 Organização dos tuplos na memória	109
3.6 Partilha da memória	109
3.7 Embricamento	110
3.8 Partilha da memória	117
3.9 Mutabilidade (I): antes (A) e depois (B)	120
3.10 Mutabilidade (II): antes (A) e depois (B)	121
4.1 Ligação Nomes - Objectos	131
4.2 Tipagem Dinâmica	133
4.3 Mudança de identidade	134
4.4 Entrada e Saída de dados	136
5.1 Os blocos de um programa	150
5.2 Sequência	153
5.3 A condicional if-then	155
5.4 A condicional if-then-else	156
5.5 A condicional if-elif-else	158
5.6 Ciclo for	164
5.7 O ciclo while	165
5.8 O Padrão Acumulador	168
5.9 Simulação de monte Carlo: o caso de π	173
5.10 Monte Carlo animado	176
5.11 A função e^{-x^2}	177
5.12 Hierarquia de excepções (visão parcial)	184
5.13 Calcular probabilidades	190
5.14 Desenho de uma grelha	192
5.15 Passeio aleatório	193
6.1 A representação de uma lista simples	200
6.2 A lista alterada	201
6.3 Uma lista mais complexa	201
6.4 <i>Aliasing</i>	202
6.5 Alterar sem efeitos não desejados	203
6.6 Tudo se complica...	204
6.7 Sem problema...	206

6.8 Ambientes, objectos e nomes	208
6.9 As Baleias dia a dia	215
6.10 Código Genético	216
6.11 Expressão Genética	217
6.12 Códificação das Bases	225
7.1 Representação em memória	241
7.2 Um ficheiro depois de aberto	243
7.3 Situação depois de lidos os primeiros 8 bytes	246
7.4 Temperaturas das cidades de Portugal	257
7.5 Gráfico com legenda	258
7.6 Temperatura e pluviosidade	261
7.7 Coeficiente de Correlção de Pearson: exemplos	268
7.8 A Apple e a Coca-cola	271
7.9 Ficheiro de dados: entrada	275
7.10 Ficheiro transformado	275
8.1 Que linda pirâmide	280
8.2 Um desenho diz mais do que mil palavras?	281
8.3 Onde estamos	282
8.4 O primeiro produto	284
8.5 Mas que lindo ... quadrado!	285
8.6 Tanto quadrado ... fora do sítio!	286
8.7 Mudar de linha só não chega	287
8.8 Agora sim!	289
8.9 Pirâmide colorida	291
8.10 A pirâmide vista de outro modo	292
8.11 Simetrias	310
8.12 Representação de uma lista de listas	311
8.13 Representação de uma lista de listas: recurso a deepcopy	312
8.14 Mastermind	317
9.1 Mapeamento RGB cores	320
9.2 Os tipos de cImage	326
9.3 Um janela simples	328
9.4 Fundo vermelho	328
9.5 Uma imagem em branco ... é preta!	329
9.6 Lidar com a cor e a posição	330
9.7 Reproduzir uma imagem	332
9.8 Negativo de uma imagem	334
9.9 Escala de cinzentos	336

9.10 Mais cinzento	337
9.11 Sepia	338
9.12 Alterando o brilho	341
9.13 O chão da cozinha	342
9.14 O problema do posicionamento	343
9.15 Distorcer uma imagem	345
9.16 Clonar um pixel	346
9.17 Espelho vertical	347
9.18 Onde colocar os pixeis?	348
9.19 Tratamento da pixelização	349
9.20 Vizinhança	350
9.21 Aplicar um filtro a uma imagem	353
9.22 Filtro gaussiano	354
9.23 Aplicando um filtro gaussiano	354
9.24 Operadores de Sobel	356
9.25 Operadores de Sobel	358
9.26 Acrescentar uma moldura	364
9.27 Corte de imagem	364
9.28 A preto e branco	365
9.29 Redução de uma imagem	366
9.30 Espelho horizontal	367
9.31 Redução de cores	367
9.32 Suavizar sem moldura	368
9.33 GandyStein	368
9.34 Encriptar uma imagem	369
10.1 <i>Sierpinski Gasket</i>	372
10.2 Sierpinski: construção	372
10.3 Torres de Hanói	373
10.4 Resolução de um sub problema semelhante	374
10.5 Torres de Hanói: solução final	374
10.6 Factorial: fase de desenrolar	377
10.7 Factorial: fase de enrolar	378
10.8 Fibonacci: coelhos e reprodução	382
10.9 Números de Fibonacci e Binómio de Newton	384
10.10 Inverter uma sequência	387
10.11 Inversão: alternativa	388
10.12 Sobe e Desce	390
10.13 Uma figura simples	394
10.14 Mudando o ângulo	394
10.15 Variando o lado e o ângulo	395

10.16Que linda pirâmide	396
10.17Um desenho diz mais do que mil palavras?	396
10.18Pirâmide: pensar recursivo	397
10.19Uma pirâmide colorida	399
10.20Ordenamento por Fusão	401
10.21Ordenamento Rápido	402
10.22Sierpinski: 200,4	404
10.23Fibonacci: cálculos duplicados	405
10.24Uma árvore recursiva	409
10.25O processo	409
10.26Uma árvore mais realista	410
10.27Detector de Paridade Par	412

Listagens de Código

1.1	Arranque do interpretador	19
1.2	Volume de uma esfera	31
1.3	Raiz quadrada	36
2.1	Um quadrado com o rabo de fora	49
2.2	Um quadrado perfeito	51
2.3	A função seno	57
2.4	Seno e cosseno	59
3.1	Operações com números	72
3.2	Uso das plicas	86
3.3	Marcas em cadeias de caracteres	87
3.4	Caracteres: operações básicas	89
3.5	Mais operações	93
3.6	Partilha	116
5.1	Blocos	150
5.2	Operadores relacionais	151
5.3	Outros operadores	152
5.4	Sequência	153
5.5	Condisional Geral: exemplo	158
5.6	Raízes: solução trivial	159
5.7	Raízes múltiplas	159
5.8	Testa raízes reais	160
5.9	Raízes: solução geral	161
5.10	Desenhar por repetição	162
5.11	De novo o quadrado	163
5.12	Formas simples	163
5.13	Ciclo for	164
5.14	Introdução protegida de dados	166
5.15	π segundo Leibniz	171

5.16	Método de Monte Carlo animado	174
5.17	Calcular uma área	176
5.18	Break: exemplo de uso	178
5.19	Ciclos <i>potencialmente</i> infinitos	178
5.20	Break: quadrados perfeitos	179
5.21	Continue: ímpares	179
5.22	Continue: entra código	179
5.23	Else: números primos	180
5.24	Tudo junto	180
5.25	Excepções: divisão por zero	182
5.26	Try: raízes reais apenas	183
5.27	Excepções: definição pelo utilizador	184
5.28	try: uso de finally	185
5.29	Teste para abortar programa	186
5.30	Ciclo while	189
6.1	Nota	196
6.2	Baleias	196
6.3	Mutabilidade e Referências	201
6.4	Mutabilidade e Referências (II)	202
6.5	Média	213
6.6	Média 2	213
6.7	Mediana	213
6.8	Desvio Padrão	214
6.9	Visualização	214
6.10	código genético	218
6.11	Genealogia I	232
6.12	Genealogia II	232
6.13	Pares e Ímpares	234
6.14	alterna	234
6.15	contar menores	234
6.16	Índices das ocorrências	236
7.1	Abertura de um ficheiro	240
7.2	Leitura completa de um ficheiro	244
7.3	Leitura de ficheiros	244
7.4	Operações de entrada	245
7.5	Escrita num ficheiro	247
7.6	'Navegar num ficheiro'	249
7.7	Ler e mostrar temperaturas	253
7.8	Ler ficheiro: alternativas	254
7.9	Todas as temperaturas	255
7.10	Temperaturas	255

7.11 Gráfico com legendas	257
7.12 Compara cotações	269
7.13 'Desemprego'	276
7.14 'Crescimento do PIB'	276
8.1 'Devagar se vai ao longe ...'	281
8.2 'Continuemos...'	281
8.3 'Chegaram os quadrados ...'	282
8.4 'Um quadrado de lado 20 ...'	283
8.5 'Tudo junto ...'	283
8.6 'Um problema já está resolvido ...'	285
8.7 'Mudar de linha ...'	287
8.8 'Versão final'	287
8.9 'Versão colorida'	289
8.10 'Primeira aproximação'	291
8.11 'Versão final'	292
8.12 Usar cadeias de caracteres	296
8.13 'Cálculo da sequência de consenso'	299
8.14 'Filtrar símbolos'	300
8.15 'O "tamanho" não é importante'	301
8.16 'Um bug??'	302
8.17 'Uso de assert'	303
8.18 'Um pouco melhor'	303
8.19 'Protega-se...'	304
10.1 'Torres de Hanói'	375
10.2 'Exemplo de sessão'	375
10.3 Factorial	376
10.4 'Soma'	379
10.5 'Produto'	379
10.6 'Exponencial'	379
10.7 'Modelo de recursividade linear'	379
10.8 'Somatório'	380
10.9 'MDC: iterativo'	380
10.10 'Algoritmo de Euclides'	381
10.11 'Sequência de Fibonacci'	382
10.12 'Binómio de Newton'	383
10.13 'Par - Ímpar'	384
10.14 'Sequência Alternada'	385
10.15 'Procura Simples'	386
10.16 'Capicua'	387
10.17 'Inverte'	387
10.18 'Inverte: alternativa'	388

10.19'Inverte: mais uma alternativa'	389
10.20'Sobe e Desce'	389
10.21Alternar	390
10.22Intercalar	391
10.23Intercalar: variante	391
10.24'Procura Binária'	392
10.25Árvore Binária de Procura	393
10.26'Uma figura simples'	393
10.27'Mais uma figura'	394
10.28'Ainda outra figura'	395
10.29'As pirâmides'	397
10.30'As linhas'	397
10.31'Quadrados'	398
10.32'Anagrama'	399
10.33'Anagrama: variante'	399
10.34'Permutações'	400
10.35'Ordenamento por Fusão'	400
10.36'Ordenamento Rápido'	402
10.37'Sierpinski Gasket'	403
10.38De novo o factorial	405
10.39Tipo de Dados Pilha	405
10.40Factorial Iterativo	406
10.41'Detector de Paridade Par'	412

Prefácio

And now for something
completely different ...

Monty Python

Este texto é sobre o uso de computadores para resolver problemas por recurso a programas escritos numa linguagem de programação de alto nível. No nosso caso Python. No entanto estamos menos interessados na linguagem concreta e mais nos blocos construtores presentes genericamente nas linguagens. E ainda nas boas práticas de programação, entendidas como os princípios que nos guiam do problema à sua solução informática.

Mas em que tipo de problemas estamos interessados? Todos sabemos que os computadores estão presentes em todas as actividades humanas. De um telefone celular ao controlo de tráfego aéreo passando pelo escalamento de tripulações de comboios, os computadores estão em todo o lado. De um modo mais directo os computadores estão presentes na nossa vida diária pois fazemos uso deles para enviar e receber correio, trocar mensagens curtas, jogar, escrever documentos ou navegar na Internet. Como utilizadores comuns apreciamos o facto de não termos que nos preocupar como é que os programas que usamos para essas actividades foram desenvolvidos ou funcionam. Podemos dizer que alguém (um programador ou um grupo de programadores) tornou simples o que é complexo, escondendo os detalhes. A nossa interacção com esses programas faz-se através de uma **interface** que disponibiliza um conjunto de funções que usamos com mais ou menos à vontade. Neste texto procuraremos iniciar o leitor nesta actividade conhecida por programação, mistura de arte e ciência, de intuição e princípios sólidos de projecto. Tratando-se de um texto introdutório os problemas que nos vão interessar serão necessariamente pequenos, mas não forçosamente simples. Caminharemos do elementar para o mais elaborado introduzindo as características da

linguagem escolhida à medida que for necessário.

Este livro pode ser usado numa disciplina inicial sobre programação mas também como instrumento de auto-estudo. Para quem já programa pode ainda servir para tomar conhecimento com a linguagem Python que oferece um conjunto muito significativo de vantagens. Programar significa resolver problemas concretos escrevendo programas. Os exemplos que aparecem ao longo do texto são na sua maioria exemplos clássicos que o leitor pode encontrar em qualquer outro livro sobre programação.

Organização O texto foi pensado para uma leitura sequencial podendo no entanto, em função dos conhecimentos prévios do leitor, ter um percurso de leitura diferente. A nossa experiência diz-nos que pode ser usado para um curso introdutório de programação procedural, de duração semestral, usando os capítulos do 1 ao 10. Os capítulos XXX a XXX lidam com aspectos mais avançados. —TBD— . Pode ser usado num curso semestral de complexidade intermédia.

No livro, associado ao exercícios propostos no final de cada capítulo, vai encontrar sinais que dão uma indicação do grau de dificuldade dos problemas. Assim usamos:

- **MF** para problemas triviais
- **F** para fáceis
- **M** para problemas de dificuldade média
- **D** para problemas que exigem reflexão e tempo
- **MD** para problemas de elevada complexidade

O grau de dificuldade que indicamos não é absoluto, antes é relativo ao nível de conhecimentos que o leitor é suposto ter no momento em que o resolve. Igualmente, caso o problema envolva o uso de um módulo especial tal será indicado por **Módulo nome**

Agradecimentos Este texto começou e ser construído em 2006, quando pela primeira vez foi usada a linguagem **Python** na cadeira introdutória de programação da licenciatura em Engenharia Informática do Departamento de Engenharia Informática da Universidade de Coimbra. Ao longo dos anos foram vários os alunos e colegas que comigo partilharam a disciplina que tiveram a oportunidade de ler de modo crítico o documento. A todos agradeço.

Ao meu colega Paulo Marques que me entusiasmou com a linguagem Python e com quem aprendo todos os dias algo mais sobre programação e a resolução de problemas um agradecimento especial.

Introdução

Objectivos

- ✓ Identificar as componentes principais de um computador baseado na arquitectura de Von Neumann
- ✓ Perceber o modo como um computador funciona
- ✓ Introduzir ideias básicas sobre linguagens e paradigmas de programação
- ✓ Introduzir, usando exemplos simples, aspectos básicos da linguagem Python

1.1 Computadores

Um computador não é mais do que uma associação de uma máquina com um conjunto de programas que permitem a construção e execução de **outros** programas escritos numa dada linguagem de programação. O fim último é a resolução de problemas. O computador tanto pode estar instalado comodamente na nossa secretária como estar embebido noutros equipamentos, desde uma máquina de lavar roupa a um controlador de uma central nuclear passando pelo nosso telemóvel. Nas secções seguintes vamos abordar de modo muito sumário estes aspectos.

1.1.1 Arquitectura

O ser humano tem uma característica fundamental: construtor de artefactos. Ao longo dos séculos foi inventando objectos que ampliaram as suas capacidades mecânicas (e.g., um martelo) ou os seus sentidos (e.g., um telescópio).

Com o aparecimento do computador o desafio passou a ser maior e traduz-se por criar uma extensão às suas capacidades mentais¹. Será esse o papel dos computadores! O debate sobre a possibilidade de identificar o ser humano com as máquinas tem contornos filosóficos e pré-existe mesmo a construção do primeiro computador². Há quem defende (e.g., Herbert Simon) que o ser humano e os computadores são apenas duas instâncias de algo mais abstrato a que chamaram sistema simbólico de símbolos. Outros (e.g., Rodney Brooks) falam de algo mais ousado que designam por *Robot Sapiens*, um novo ser simbiótico que resulta da associação Homem-Máquina. Independentemente deste debate é certo que, a num certo nível de abstracção, podemos associar a arquitectura funcional de um computador dos nossos dias com algumas das nossas capacidades. Muito superficialmente o ser humano possui órgãos de captura de informação (e.g., os olhos), órgãos de actuação (e.g., os nossos braços) e órgãos de processamento e armazenamento da informação (e.g., o cérebro).

Esta decomposição quando transposta para o computador dá lugar à arquitectura simplificada (por exemplo, não incluímos as componentes de rede, essenciais nos computadores de hoje) de um computador que podemos observar na figura 1.1.

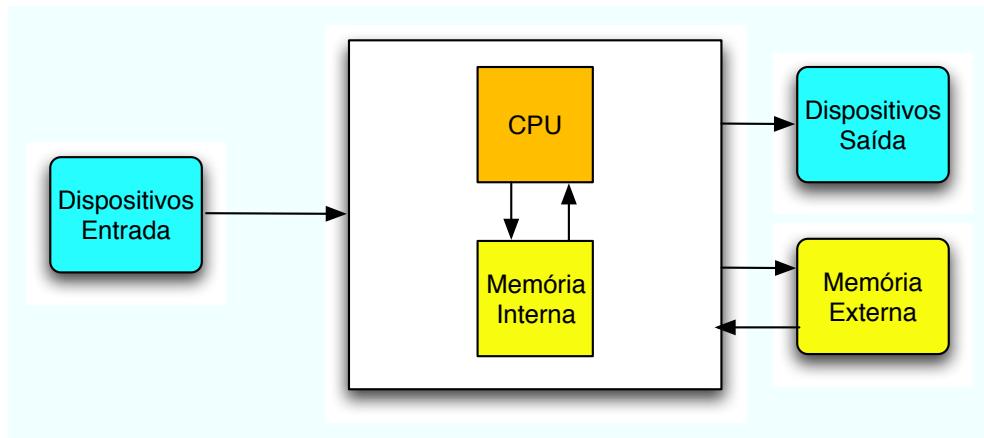


Figura 1.1: Arquitectura de um computador

Do ponto de vista da sua dinâmica um computador executa programas que foram previamente armazenados na sua memória externa (e.g., um disco duro). Esses programas são formados por instruções que são transferidas para a memória interna (e.g., RAM) e executadas pela máquina graças à

¹O sociólogo Daniel Bell chama ao computador uma *tecnologia intelectual*. Um relógio ou um mapa são outros exemplos.

²Basta pensar em Leonardo Da Vinci, Charles Babbage, ou Alan Turing.

unidade de processamento central (CPU). Eventualmente pode haver recurso à entrada de dados através de dispositivos apropriados (e.g., teclado) e visualização de resultados graças aos dispositivos de saída (e.g., o monitor). Esta é a arquitectura dita de Von Neumann que ainda hoje é predominante³.

1.1.2 Funcionamento

Quando recebemos o nosso novo computador ele vem equipado com um programa fundamental, o sistema operativo⁴. Tanto pode ser Windows, como Linux, Mac OS X, ou outro qualquer. Um sistema operativo é constituído por um núcleo mais um conjunto de programas que permitem entre outras coisas uma interacção amigável com o utilizador (e.g., um sistema de janelas, controlo por meio de um rato, ...⁵). Os nossos programas são escritos graças a um editor. Este pode existir autonomamente ou estar embebido num ambiente integrado de desenvolvimento (designado por IDE na terminologia anglo-saxónica) como o Visual Studio, o Eclipse ou o Xcode. Depois de escrito e depurado de eventuais erros o nosso programa pode ser executado por recurso a programas como os compiladores ou os interpretadores. Na figura 1.2 podemos ver a filosofia do uso de um compilador.

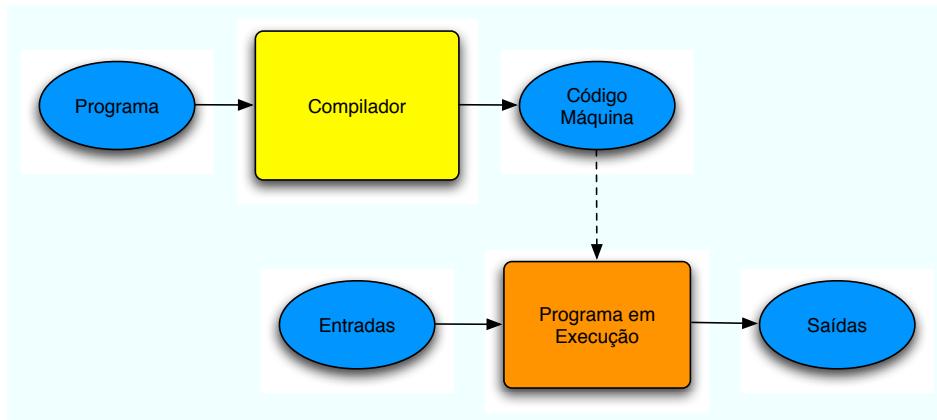


Figura 1.2: Compilação

³A entrada em cena de vários processadores cada um com vários núcleos veio alterar um pouco a visão simples da arquitectura convencional, embora as diferenças tenham mais que ver com o modo como o programa é executado do que com a filosofia subjacente à arquitectura descrita.

⁴Se fomos nós que fabricámos o computador vamos ter que resolver a questão de instalar o SO adequado ao hardware escolhido. Pode ser uma tarefa divertida!

⁵Hoje já estamos também no tempo dos ecrans sensíveis ao toque que faz das mãos de novo um instrumento central na interacção do ser humano com o computador.

O nosso programa (designado por programa fonte) começa por ser traduzido pelo compilador num novo programa escrito em linguagem máquina, i.e., numa linguagem que aquela arquitectura particular de computador entende. O programa traduzido é posteriormente executado havendo lugar eventualmente à entrada de dados e saída de resultados⁶

Já no caso do recurso a um interpretador (ilustrado na figura 1.3) o programa não é todo traduzido mas antes cada instrução que o compõe é interpretada e executada de acordo com uma dada ordem especificada pelo programa.

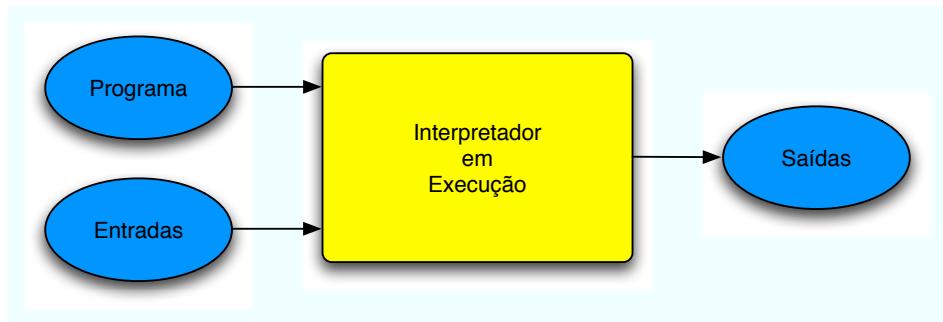


Figura 1.3: Interpretação

Existem vantagens e inconvenientes em cada uma das abordagens. À velocidade da compilação podemos contrapor a maior facilidade de desenvolvimento ou portabilidade quando optamos por um interpretador⁷

Existem modelos que podemos considerar híbridos. O programa é compilado para código intermédio sendo de seguida interpretadas as suas instruções graças a uma máquina virtual. É a opção usual em **Java** ou em **Python** (ver figura 1.4).

1.2 Linguagens

Em qualquer sociedade a comunicação entre os seus membros é mediatisada por uma linguagem. Este princípio aplica-se às formigas, aos macacos, aos seres humanos, ou seja, a qualquer conjunto de entidades onde existe o sentido de colectivo. Desde que os computadores passaram a fazer parte da

⁶O processo é um pouco mais complexo, pois pode haver lugar à ligação do nosso programa a outros programas ou bibliotecas.

⁷As linguagens de programação não são forçosamente prisioneiras de um destes modos. Mas é mais normal ver um programa escrito em C optar por um compilador e um programa escrito em **Lisp** optar por um interpretador.

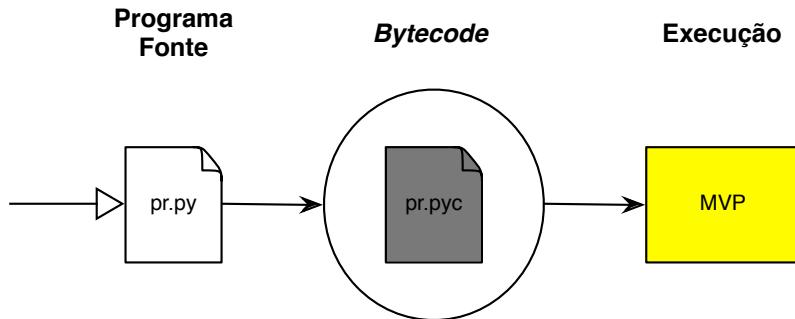


Figura 1.4: Interpretação: Python (Adaptado de [1])

nossa sociedade tornou-se incontornável a necessidade de dialogarmos com eles por meio de uma linguagem que eles entendam. Porém, ao contrário dos seres humanos, que são máquinas baseadas no carbono, os computadores são máquinas baseadas no silício. O seu interior é um conjunto bastante denso de componentes electrónicas ligadas entre si de modo como o descrito na figura 1.1. Esses componentes electrónicos funcionam segundo uma lógica binária levando a que no alfabeto da sua linguagem existam apenas dois símbolos que convencionamos serem o 0 e o 1. Entre a linguagem dos humanos e a linguagem da máquina, formada a partir de 0s e 1s, foi preciso estabelecer uma ponte que torne a comunicação simultaneamente natural para o ser humano e entendível para o computador. Aceite este princípio a consequência natural é de que é preciso ao ser humano descrever os seus problemas num programa escrito numa linguagem de programação que seja próxima da sua linguagem e é forçoso que o computador traduza (compile/interprete) esses programas na sua própria linguagem (ver figura 1.5). A segunda questão já foi aflorada sumariamente (ver secção 1.1.2). A primeira questão é um dos objectos principais do presente texto.

[Carbono versus Silício](#)

1.2.1 Da máquina ao utilizador

Como referimos, se é verdade que a comunicação que as linguagens de programação permitem se dirige em última análise à máquina, não é menos verdade que devem sobretudo possibilitar o enunciar de soluções a um nível que facilite também a comunicação entre agentes humanos. Uma linguagem de programação serve por isso fundamentalmente dois objectivos, evidentemente relacionados:

- é um veículo para exprimir acções a serem executadas pelo computador,

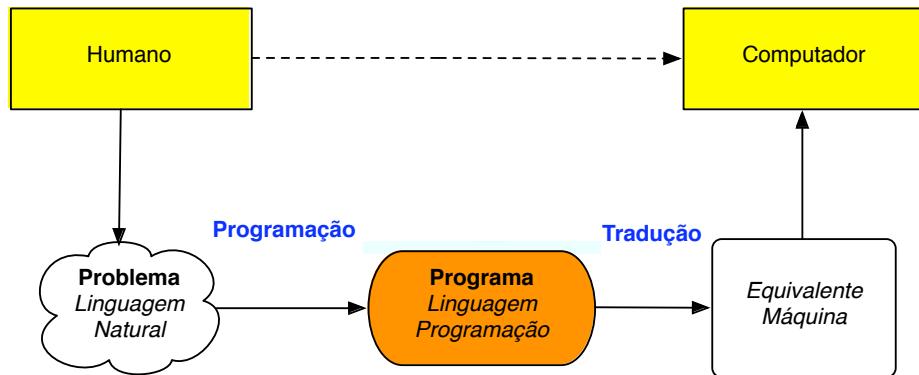


Figura 1.5: O processo de comunicação humano - computador

- é um conjunto de conceitos que permite modelar de forma abstracta os problemas do mundo real.

Acontece que historicamente a programação teve que percorrer um longo caminho do ponto de vista do utilizador para se "afastar" da máquina e se aproximar do utilizador humano. Para perceber melhor esse caminho temos que detalhar um pouco mais a arquitectura que apresentámos na secção 1.1.1. A figura 1.6 mostra o interior da unidade de processamento central (CPU).

A sua arquitectura interna corresponde às três funções que a CPU tem que realizar:

- realizar operações aritméticas (UAL)
- armazenar localmente valores (registos)
- controlar os passos a realizar de acordo com as instruções do programa (UC)

Podemos ainda concretizar melhor a unidade de controlo (ver figura 1.7).

O contador de programa (CP) controla a próxima instrução a ser executada. Depois de extraída da respectiva memória a instrução é descodificada dando origem a vários sinais de controlo que envolvem os registos e afectam eventualmente o contador de programa. Se houver lugar a saltos na ordem natural de execução das instruções o seu endereço é carregado no CP. Caso contrário o CP é incrementado de uma unidade.

Conhecida a arquitectura concreta do processador foram desenvolvidas linguagens de programação simples, chamadas linguagens *assembly*, que permitiam que em vez de escrever programas só como sequências de 0s e de 1s

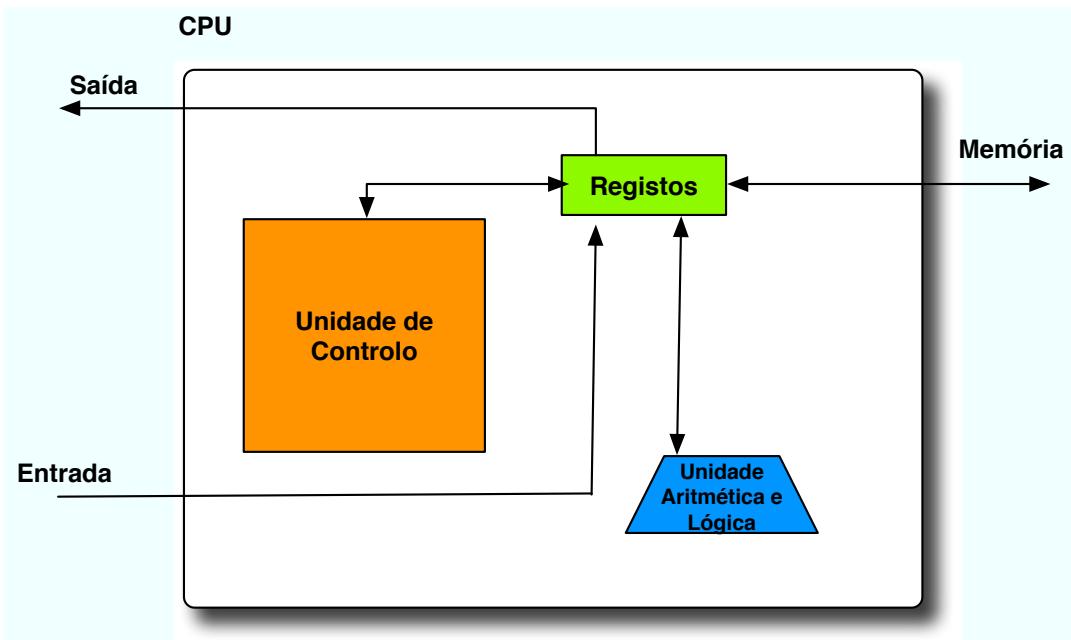


Figura 1.6: A unidade de processamento central

pudesssem ser usadas mnemónicas. No exemplo da figura 1.8 procura ilustrar-se um programa que permite determinar qual o maior de dois números e escrevê-lo.

O programa começa por ler os dados para os registos 1 e 2. De seguida (instrução 3) compara-os, deixando no registo 3 o valor booleano da comparação. Se o conteúdo do registo 3 for negativo salta para a instrução de endereço #07, alterando o contador de programa, e executa-a. A instrução seguinte (em #08) faz parar a execução. Caso contrário, o contador de programa não é alterado, é escrito o conteúdo do registo 1 e a execução termina.

Para que o utilizador humano pudesse escrever programas em *assembly* foi preciso escrever o respectivo tradutor que muito naturalmente se chamou *assembler*. Este modo de programar é evidentemente um grande avanço relativamente ao processo de programação em linguagem máquina. Permite, porque se conhece bem a arquitectura do processador, escrever programas altamente optimizados. Mas, como o exemplo atrás ilustra, para efectuar uma pequena operação é preciso escrever muito código. Foi pois com naturalidade que apareceram linguagens que nos permitiram afastarmo-nos da máquina. O preço a pagar foi o desenvolvimento de tradutores complexos e dificuldades em optimizar o código. Na próxima secção falaremos um pouco de alguns tipos de linguagens de programação.

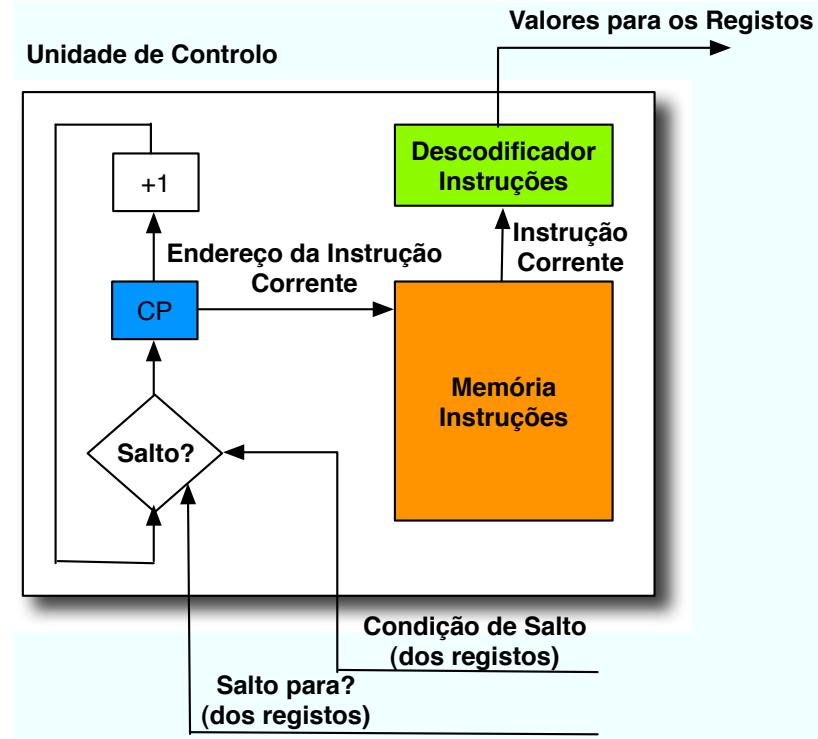


Figura 1.7: A unidade de controlo

1.2.2 Diferentes paradigmas

Existem largas dezenas de linguagens de programação. Que têm de comum os programas escritos nas diferentes linguagens? De um modo simples os programas de computador podem ser vistos como “frases” de uma linguagem de programação envolvendo fundamentalmente duas componentes:

- Dados
- Operações

Programa = facto que se pode traduzir pela equação:
Dados +
Operações

$$\text{Programa} = \text{Dados} + \text{Operações}$$

Tradicionalmente a execução de um programa é vista como a actuação das operações sobre os dados levando à modificação do estado do programa. Uma computação é então vista como a sequência de estados por que vai passando o programa.

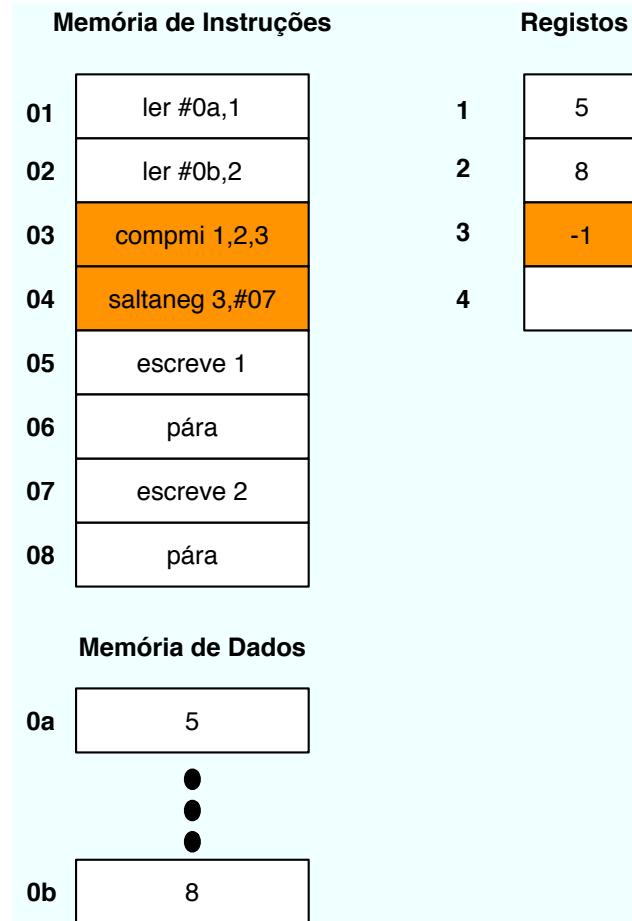


Figura 1.8: Programa em Assembly

Exemplo 1.1**Contas bancárias**

- dados: conta (nome, saldo, taxa de juros,...)
- operações: levantamentos, depósitos, cálculo de juros,...

1.1

Como dissemos, as operações actuam sobre os dados modificando-os. Por exemplo, um levantamento de uma conta vai alterar o saldo (estado) da conta.

Se a linguagem de programação favorecer a proximidade à máquina podemos comparar o programador com um construtor civil. No caso da linguagem se aproximar do problema o programador assemelha-se mais a um arquitecto.

Nível das linguagens

Convém não confundir proximidade da máquina com linguagem de “baixo nível”, ou proximidade do problema com “alto-nível”. Ser orientada para a máquina significa apenas que dispomos de modos de dialogar directamente com o hardware (registos, memória, dispositivos de entrada/saida) . No entanto uma linguagem próxima da máquina pode possuir outros elementos, como seja estruturas de controlo, estruturas de dados, módulos, etc. A linguagem *C* é um exemplo paradigmático.

O projecto de uma linguagem de programação é sempre um exercício de compromisso entre várias características desejáveis numa linguagem. Alguns dos objectivos são:

- **utilidade:** a característica é usada muitas vezes, não pode ser feito de outro modo?
- **conveniência:** a característica permite a escrita sucinta de código?
- **eficiência:** computacionalmente é fácil ou difícil traduzir a característica?
- **portabilidade:** pode ser implementada em qualquer máquina?
- **legibilidade:** torna o programa mais legível?
- **capacidade de modelização:** torna o significado do programa mais claro?
- **simplicidade:** A linguagem possui um conjunto simples, unificado e genérico de características ou é um aglomerado de possibilidades dedicadas?
- **clareza semântica:** ausência de ambiguidade semântica?

Parafraseando Orwell podemos afirmar que as linguagens são todas iguais mas há umas mais iguais do que outras no sentido de responderem melhor às características que disponibilizam tendo em atenção o problema que pretendem resolver⁸. Podemos agrupar as diferentes linguagens em famílias de

⁸Com efeito, teoricamente todas as linguagens de programação existentes são formalmente equivalentes à Máquina de Turing Universal, tendo por isso o mesmo poder computacional.

acordo com determinados princípios. É sempre um exercício subjectivo e nem sempre os subconjuntos resultantes são disjuntos. Vamos apresentar uma classificação em termos do modelo ou **paradigma** subjacente. Queremos com isto apenas dizer que uma determinada linguagem favorece um dado *estilo de programação*. Por estilo de programação entende-se ([2])

"um modo de organizar programas na base de um dado modelo conceptual de programação e uma linguagem que torna os programas escritos, com base nesse estilo, claros."

Declarativas ou relacionais

Um programa seguindo o modelo relacional é normalmente expresso por um conjunto de fórmulas lógicas que estabelecem relações entre objectos⁹ A execução de um programa não é mais do que uma "query" ao conjunto de relações que definem o programa. Existe a noção de variável lógica que recebem os seus valores graças ao mecanismo de unificação. Têm uma natureza declarativa significando isto que programar não envolve a indicação do modo como a solução se obtém mas apenas a definição do que é a solução: o controlo é implícito. Assim podemos dizer que a nossa equação se transforma em:

$$\text{Programa} = \text{Lógica} + \text{Controlo}$$

Vejamos um exemplo muito simples na linguagem Prolog.

```

1  pai-de(patricia, ernesto).
2  pai-de(ernesto, jose).
3  avo(X,Y) :- pai-de(X,Z), pai-de(Z,Y).
4
5  ?- avo(patricia,W).
6  W= jose

```

Neste programa tão simples afirmamos uma relação de paternidade entre dois objectos (ditas constantes em Prolog) e definimos o significado de ser avô, através de uma cadeia de duas relações de paternidade. Executar o programa traduz-se em questionar o sistema se existe alguém que seja avô da **patrícia**, ao que o sistema responde afirmativamente e mostrando o objecto

⁹Tecnicamente um programa na linguagem Prolog, que favorece este paradigma, é um conjunto de cláusulas de Horn que podem ser separadas em factos, e.g., (*pai – de(patricia, ernesto)*). ou regras, e.g., (*avo(X,Y) : –pai – de(X,Z), pai – de(Z,Y).*).

que satisfaz a relação.

A linguagem Prolog foi desenvolvida por Alain Colmerauer e a sua equipa da Universidade de Montpellier (França). Inicialmente tratava-se apenas de um sistema potente para tratamento da linguagem natural. O chamado Projecto Japonês do Computador de 5^a Geração veio dar num entanto forte impulso à linguagem Prolog que passou a ser vista como uma linguagem de uso geral, usada sobretudo pela comunidade de Inteligência Artificial.

Funcionais

Neste paradigma, um programa é um conjunto embricado de expressões envolvendo chamadas de funções. Sintacticamente não existe distinção entre dados e algoritmo. Podemos dizer então que:

$$\text{Programa} = \text{Dados} + \text{Funções}$$

Não existe a noção de atribuição (modificação destrutiva do valor associado a um objecto). A transmissão de valores é feita pelo mecanismo de ligação de parâmetros no momento da activação das funções.

Analisemos um exemplo simples na linguagem Lisp.

```

1 (defun ultimo (l)
2   ( cond
3     ((null (rest l)) (first l))
4     (t (ultimo (rest l)))))
5
6 ?- (último '(a b c))
7 = c

```

Neste exemplo calculamos o último elemento de uma sequência de elementos.

A função *ultimo* tem um corpo formado por duas opções o que é indicado por *cond*. A primeira opção, corresponde à situação em que a lista que contém os elementos tem apenas um elemento caso em que o resultado é o primeiro (*first*) e único elemento da lista. A segunda opção, diz-nos simplesmente que caso a lista tenha mais do que um elemento o último obtém-se procurando, **recursivamente**, no resto da lista (*rest*) ¹⁰.

A linguagem Lisp é das mais antigas sendo contemporânea de linguagens como FORTRAN. Foi desenvolvida por John McCarthy nos anos 50 e é usada

¹⁰Mais adiante, no capítulo 10, trataremos com mais cuidado do problema da recursividade.

ainda hoje activamente pela comunidade de Inteligência Artificial, devido às facilidades que dá ao programador para efectuar cálculo simbólico envolvendo objectos complexos.

Imperativas ou procedimentais

O paradigma mais comum até há alguns anos atrás era o paradigma imperativo ou procedural. Um programa procedural é caracterizado por uma sequência de instruções que actuam ou sobre os dados (variáveis), alterando de forma destrutiva o seu valor, ou sobre a ordem pela qual as intruções são executadas (instruções de controlo). A equação passa a ser:

$$\text{Programa} = \text{Estruturas de Dados} + \text{Algoritmo}$$

A linguagem *C* é o exemplo clássico. Vejamos um exemplo simples.

```

1 #include <stdio.h>
2
3 int power(int base, int exp){
4     // Calcula base ** exp
5     int i, p;
6     p = 1;
7     for (i = 1; i <= exp; ++i)
8         p = p * base;
9     return p;
10 }
11
12 int main() {
13     // teste de power
14     int i;
15     for (i = 0; i < 10; ++i)
16         printf("%2d %5d %7d\n", i, power(2,i), power(-3,i));
17     return 0;
18 }
19 }
```

Este programa imprime uma tabela com os valores de **i**, potência de base 2 de **i** e potência de base -3 de **i**, para **i** a variar entre 1 e 10.

```

1 run
2 [Switching to process 5247]
3 Running...
4 0      1      1
```

```

5   1      2      -3
6   2      4       9
7   3      8     -27
8   4     16      81
9   5     32    -243
10  6     64     729
11  7    128   -2187
12  8    256    6561
13  9    512  -19683
14
15 Debugger stopped.
16 Program exited with status value:0.

```

O programa principal, **main**, repete os cálculos para cada valor de **i**. Socorre-se da função **power**. Esta última calcula a base levantada ao expoente multiplicando a base por ela própria o número de vezes dado pelo expoente.

Orientadas aos Objectos

A programação orientada aos objectos (POO) apareceu como um modo de dominar a complexidade do processo de análise, desenvolvimento e implementação de aplicações informáticas. Uma das primeiras tentativas para dominar a complexidade aparece nas linguagens imperativas através do conceito de subrotina. No entanto a dependência de cada subrotina de um estado global (variáveis globais) tem como consequência que cada subrotina não constitua verdadeiramente um módulo separado. A programação funcional tentou responder a este problema fazendo com que o único modo de comunicação com o exterior das funções seja através dos parâmetros que lhe são passados. A POO aparece como um compromisso ao partir o estado global por pequenas estruturas fechadas denominadas objectos.

Retomemos de novo a equação:

$$\text{Programa} = \text{Objectos} + \text{Operações}$$

O que distingue a POO da programação imperativa tradicional é o facto de que um programa agora deve ser visto como um conjunto de objectos que são manipulados pelas operações às quais estão intimamente ligados.

Ao longo do tempo têm aparecido várias tentativas de definir claramente o que se entende por orientação aos objectos. A definição que é maioritariamente consensual é a de Peter Wegner ([3]):

Orientação aos Objectos = Objectos + Classes + Herança

Um programa segundo a filosofia orientado a objectos pode ser descrito pela equação:

$$\text{POO} = \text{Orientação aos Objectos} + \text{Mensagens}$$

Os objectos devem ser vistos com entidades que encapsulam não apenas dados mas também operações sobre os dados;

Uma **classe** é um modelo¹¹ de um grupo de objectos semelhantes com comportamento idêntico. Um objecto aparece assim como instância de uma classe. É possível definir uma hierarquia de classes. **Herança** é um modo de definir como novas classes herdam propriedades das classes que estão acima delas na hierarquia. As **mensagens** da equação acima são o que permite a comunicação entre objectos. **Métodos** é o modo como um objecto particular trata uma mensagem.

Vejamos um exemplo simples na linguagem Java.

```

1 class Pessoa{
2     String nome;
3     int idade;
4     void dizOla () {
5         System.out.println ("Olá eu sou o " + nome);
6     }
7 }
8
9 class Programa {
10    public static void main (String [] args) {
11        Pessoa cliente = new Pessoa();
12        cliente.nome = "Ernesto";
13        cliente.idade = 60;
14        cliente.dizOla();
15    }
16 }
```

Este exemplo simples mostra a definição de duas classes e o modo como interagem. A classe pessoa define objectos com dois atributos (*nome* e *idade*) e tem associado o método **dizOla** que se limita a imprimir uma frase. A segunda classe, Programa, cria um objecto da classe Pessoa, instanciado com o nome Ernesto e a idade 60 e de seguida executa o método *dizOla*.

¹¹No sentido do inglês *template*.



Linguagens e paradigmas

Convém mais uma vez referir que, embora existam linguagens “puras” dentro de cada filosofia de funcionamento, as linguagens de programação mais usadas possuem características híbridas. Assim , por exemplo, a linguagem Common LISP possui mecanismos destrutivos idênticos aos das linguagens procedimentais, a linguagem Pascal possui elementos funcionais, Logtalk é uma extensão à linguagem Prolog que comporta elementos das linguagens orientadas a objectos, é possível fazer programação procedural em C++, etc..

1.2.3 Modelo PCAP

Construímos programas como construímos qualquer sistema. Precisamos de um conjunto de primitivas que usamos para compor sistemas mais complexos. Podemos abstrair sistemas para posterior reutilização, transformando-nos na realidade em novas primitivas. Essas mesmas abstracções podem, por seu turno, ser elas próprias abstraídas, originando padrões (ver figura 1.9).

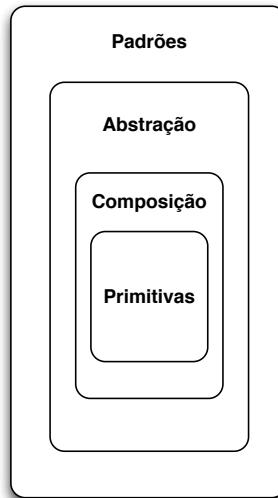


Figura 1.9: Modelo PCAP

Quando pensamos em linguagens de programação esta maneira de ver, baseada nas ideias de Primitivas, Composição, Abstracção e Padrões (**PCAP**)¹²,

¹²Esta ideia aparece de modo explícito nas notas do curso 6.01 - *Introduction to EECS I*, do MIT, podendo ser consultada um <http://mit.edu/6.01/www/index.html>.

pode ser instanciada tendo em atenção as duas componentes (dados e processos), como se mostra na tabela 1.1.

Tabela 1.1: Modelo PCAP

	Processos	Dados
Primitivas	<code>+, *, <=</code>	Números, cadeias caracteres
Composição	<code>if, for</code>	lists, dicionários, conjuntos
Abstracção	<code>def</code>	Tipos de dados abstractos, classes
Padrões	Funções de segunda ordem	Funções Genéricas

A apresentação da linguagem **Python** e o modo como podemos construir programas será feito de acordo com a identificação destes quatro aspectos, caminhando do mais simples (primitivas) para o mais complexo (padrões).

1.3 A linguagem Python

A linguagem **Python** foi criada por Guido Van Rossum no final dos anos 80, início dos anos 90. Deve o seu nome à série Monthly Python e não a nenhuma cobra, embora o símbolo convencional da linguagem tenha passado a ser o lindo e esguio animal. Trata-se de uma linguagem interpretada, de alto nível e que suporta vários paradigmas de programação (procedimental, funcional e orientada aos objectos ¹³), sem forçar nenhum deles. É uma linguagem gratuita, de fonte aberta, existindo versões para as diferentes plataformas em <http://www.python.org>. O código é altamente portável, correndo em todas plataformas, como por exemplo o Microsoft Windows, Linux e Mac Os X¹⁴. É uma linguagem simples de aprender e de usar, tornando-se por isso uma escolha crescente de diversas universidades e escolas quando se trata das disciplinas de iniciação à programação. Tem um conjunto de características que a tornam uma linguagem poderosa, como por exemplo tipagem dinâmica, gestão automática da memória, estruturas de dados dinâmicas como listas e dicionários, exceções, funcionais, iteradores, geradores de decoradores. Pode ser misturada com outras linguagens, como C ou Java. Se a componente de base do interpretador é pequena tem um conjunto de módulos nativos ou de terceiras partes que permitem expandir a linguagem e a

¹³O paradigma relacional não tem expressão em Python, embora o leitor interessado pode procurar pelo projecto **pylog**.

¹⁴É também possível fazer correr **Python** em plataformas como o iPad ou o iPhone!

tornam uma excelente opção para diferentes aplicações do mundo real, incluindo programação de sistemas, bases de dados, programação em redes, computação científica, programação para a Web, *scripting*, interfaces gráficas, ou ainda jogos. A sua utilidade pode ser medida pelas companhias que de modo crescente tem vindo a usar **Python** no desenvolvimento das suas aplicações, como a Google, a Intel, a Cisco, ou a Youtube para nomear apenas algumas.

A filosofia da linguagem pode ser melhor explicitada através de um conjunto de máximas conhecidas como o **Zen de Python** ¹⁵:

```
1 The Zen of Python, by Tim Peters
2
3 Beautiful is better than ugly.
4 Explicit is better than implicit.
5 Simple is better than complex.
6 Complex is better than complicated.
7 Flat is better than nested.
8 Sparse is better than dense.
9 Readability counts.
10 Special cases aren't special enough to break the rules.
11 Although practicality beats purity.
12 Errors should never pass silently.
13 Unless explicitly silenced.
14 In the face of ambiguity, refuse the temptation to guess.
15 There should be one-- and preferably only one --obvious way to
   do it.
16 Although that way may not be obvious at first unless you're
   Dutch.
17 Now is better than never.
18 Although never is often better than *right* now.
19 If the implementation is hard to explain, it's a bad idea.
20 If the implementation is easy to explain, it may be a good
   idea.
21 Namespaces are one honking great idea -- let's do more of
   those!
```

¹⁵Este texto pode ser obtido lançando o interpretador de Python e efectuando o comando `import this` ou em <http://www.python.org/dev/peps/pep-0020/>.

1.4 Programas

De um modo simples um programa é a descrição rigorosa numa linguagem de programação de um processo que permite ao computador resolver um problema. É o nosso veículo de comunicação com o computador. Já sabemos que diferentes linguagens organizam os seus programas de diferentes modos ou paradigmas (ver secção 1.2.2) e recorrem a uma linguagem própria. No caso da linguagem **Python** essa decomposição é feita do seguinte modo (adaptado de [1]):

1. Os programas são compostos por **módulos**;
2. Os módulos contêm sequências de **comandos**;
3. Alguns comandos são **expressões**;
4. As expressões criam e manipulam **objectos**.

Iremos explorar cada uma destas construções, da mais elementar para a mais complexa. Saliente-se no entanto, desde já, que o conceito de objecto é central em **Python**, como aliás noutras linguagens. O grau de **integração** entre objectos e operações determina em parte o paradigma de programação suportado pela linguagem. Em **Python** no entanto esta centralidade é total pois em **Python tudo são objectos**, como ficará claro ao longo do texto. Os objectos de um programa têm associado operações que permitem efectuar sobre eles diferentes coisas, tais como construir, consultar, modificar ou testar.

Exemplo muito simples

Chegou o momento de sabermos o que podemos usar **Python**. A forma mais simples consiste em trabalhar no **modo interactivo**: activa-se o interpretador e escrevem-se instruções para serem executadas uma a uma de modo ordenado. Quando chamamos o interpretador de Python aparece algo o que se ilustra na listagem 1.1.¹⁶

- ```
1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
 2336.1.00)]
3 Type "help", "copyright", "credits" or "license" for more
 information
```

---

<sup>16</sup>O que efectivamente aparece varia de máquina para máquina e de versão do interpretador.

4    >>>

### Listagem 1.1: Arranque do interpretador

O símbolo `>>>` é o **caractér de pronto**<sup>17</sup> e indica que o interpretador está pronto para receber uma instrução e executá-la. Uma primeira experiência que podemos fazer é usar Python como se de uma vulgar calculadora se tratasse.

```
1 >>> 6 + 7 ←
2 13
3 >>> 8 * 24 ←
4 192
5 >>> 16 // 2 ←
6 8
7 >>> 7 - 3 ←
8 4
9 >>>
```

Neste exemplo simples mostramos que podemos fazer somas (+), produtos (\*), divisões inteiras (//) e subtrações (-), envolvendo números inteiros. Por exemplo, na primeira linha o utilizador escreve `6 + 7` e depois de carregar na tecla de *return*, indicado na listagem por `←`, o interpretador ecoa na linha seguinte o valor (objecto) que resulta de efectuar a operação indicada. Também podemos efectuar as mesmas operações com números reais ou números complexos. Como é lógico, em Python podemos efectuar todas as operações convencionais com números. A partir de agora não incluiremos nas listagens de código a indicação da tecla de *return*. No modo interactivo, o interpretador funciona num ciclo conhecido por **lê - avalia - escreve**<sup>18</sup> (ver figura 1.10): é lida uma expressão (**lê**), é determinado o valor associado à expressão (**avalia**), e é visualizado o resultado (**escreve**). Quando compomos objectos e operações formamos **expressões**.

Expressões

## 1.5 O meu primeiro programa

Os computadores são um pouco mais do que simples calculadoras. Vejamos com um exemplo simples o que é possível ser feito. Admitamos que conhecemos a fórmula que nos dá o peso ideal de uma pessoa em função da sua altura e género. Para alguém com 1.81m de altura é fácil efectuar os cálculos (o primeiro valor é para os homens, o segundo para as mulheres).

---

<sup>17</sup>Do inglês *prompt character*.

<sup>18</sup>Do inglês *Read - Eval - Print*.

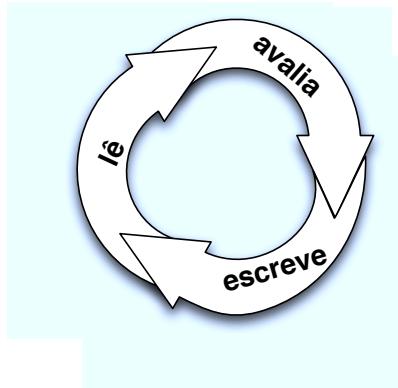


Figura 1.10: O ciclo lê - avalia - escreve.

```

1 >>> (72.7 * 1.81) - 58
2 73.587000000000018
3 >>> (62.1 * 1.81) - 44.7
4 67.701000000000008
5 >>>

```

Notar que se trata de números reais indicado pela presença de uma parte decimal<sup>19</sup>. Podemos tornar a expressão mais legível associando nomes aos valores manipulados. Para tal introduzimos o conceito de variável. Em termos simples, uma variável é um nome pelo qual um objecto passa a poder ser designado, isto é, o nome é um **atributo** do objecto. Em Python para estabelecermos a associação de um nome a um objecto usamos a **instrução de atribuição**, através do uso do sinal de igual (=)<sup>20</sup>. Vejamos para o homem como pode ser feita a associação.

```

1 >>> altura = 1.81
2 >>>

```

Ao contrário do que aconteceu no caso das operações, agora nada é ecoado. No entanto, internamente a associação foi feita, como se pode verificar na listagem seguinte.

```
1 >>> altura = 1.81
```

---

<sup>19</sup>O leitor não se assuste com o resultado com tantos zeros e uns dígitos lá no final. Com o tempo perceberá que as máquinas não são perfeitas e, por isso, não são capazes de precisão infinita.

<sup>20</sup>Para quem está habituado ao significado usual de igualdade, o recurso ao sinal de igual para estabelecer esta associação nome-objecto é motivo de equívocos e de muitos erros de programação. Por exemplo, qual o significado de  $x = x + 5$ ?

```

2 >>> altura
3 >>> 1.81

```

Quando, na segunda linha acima, digitamos **altura** seguido da tecla de *return*, o interpretador lê a expressão, vai procurar o objecto cujo nome é **altura** e devolve o **valor** do objecto. Simples! Para além do atributo **nome**, os objectos têm outros três atributos muito importantes: **identidade**, **valor**, e **tipo**. De um modo simples a identidade indica o local da memória onde está armazenado o objecto, o valor é o valor do objecto, e o tipo diz-nos qual a natureza do objecto (inteiro, real, complexo, no caso dos números). Na figura 1.11 mostramos graficamente a situação que é criada após ser executada a instrução de atribuição acima indicada.

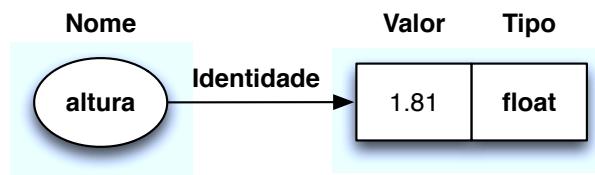


Figura 1.11: Objectos e seus atributos

Regressemos à questão inicial e exemplifiquemos como se podem efectuar o cálculo do peso ideal usando agora o nome no lugar do valor do objecto.

```

1 >>> altura = 1.81
2 >>> (72.7 * altura) - 58
3 73.587000000000018
4 >>>

```

Claro que podíamos ter usado outro nome para a altura, por exemplo **xpto**. Veremos mais à frente que existem algumas limitações e regras para o uso dos nomes. Na realidade não avançámos muito. Será que também podemos associar um nome a uma expressão mais complexa e não apenas a um objecto simples? Vejamos.

```

1 >>> peso = (72.7 * altura) - 58
2 Traceback (most recent call last):
3 File "<string>", line 1, in <fragment>
4 NameError: name 'altura' is not defined
5 >>>

```

Parece que não funciona pois o interpretador dá-nos uma mensagem de erro. A mensagem é clara: estamos a referir um nome que não é conhecido

do sistema, i.e., **altura**, porque não está definido. Como podemos remediar esta situação? Parece óbvio que necessitamos de tornar o nome conhecido. Mas, como o fazer se os nomes apenas existem associados a objectos? Bom, já sabíamos que para a expressão poder ter um valor era preciso conhecer o valor da altura de uma pessoa. Logo,

```

1 >>> altura = 1.81
2 >>> peso = (72.7 * altura) - 58
3 >>> peso
4 73.587000000000018
5 >>>
```

Agora **peso** é um nome associado a um objecto que resultou de termos efectuado os cálculos da expressão depois de substituir o nome **altura** pelo objecto que nomeia.

Os dados de que necessitamos e as associações a efectuar entre nomes e expressões podem obtidos de modo diferente. Vejamos como.

```

1 >>> altura = eval(input("Altura sff: "))
2 Altura sff: 1.81
3 >>> peso = (72.7 * altura) - 58
4 >>> print("Peso ideal para a altura ", altura, "'é'" peso)
5 Peso ideal para a altura 1.81 é 73.58700000000002
6 >>>
```

O que aconteceu? Bem usámos duas novas instruções, uma para entrada de dados (**input**) e, outra, para a saída do resultado (**print**). A instrução **input** permite solicitar ao utilizador a introdução dos dados. Para já, refira-se apenas que o uso da função **eval** permite interpretar o dado introduzido por nós, neste caso como um número real. A instrução **print** permite imprimir o que lhe é dado como argumento, directamente se for um objecto, indirectamente se for um nome ou uma expressão<sup>21</sup>. Os três tipos de instruções referidos, atribuição, entrada e saída, formam um grupo importante de instruções do paradigma procedimental, pois de alguma forma alteram o estado de alguns objectos. O leitor atento terá notado que alguns dos argumentos das funções **input** e **print** são ou nomes (de objectos), ou frases enquadradas por plicas ("Altura sff: ", "Peso ideal para a altura "). Na realidade estes últimos casos não são mais do que outro tipo de objecto, as **cadeias de caracteres**. Mais à frente discutiremos de forma aprofundada estes objectos e as operações que com eles podemos efectuar.

[Cadeias de Caracteres](#)

---

<sup>21</sup>Na realidade, um nome não é mais do que um caso simples de expressão.

## Funções

Aprofundemos a questão relacionada com associar toda uma expressão a um nome. Para quem está habituado a funções matemáticas, ter uma expressão que depende de variáveis, não é uma ideia nova (ver a figura 1.12).

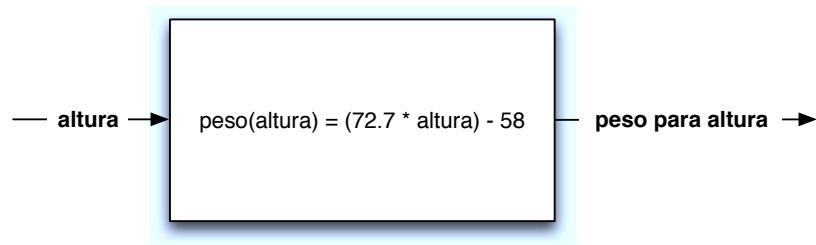


Figura 1.12: A função peso: conhecido o valor de **altura** podemos calcular o peso.

### Definições

Em Python existe também a mesma noção de função. Precisamos de uma nova notação para associar à expressão um nome que depende da variável *altura*, sem que *altura* esteja definida e tal motive um erro, como no caso de uma função matemática. Isso leva-nos à introdução de definições. Uma definição é uma **abstracção** para uma operação mais ao menos complexa.

```

1 >>> def peso(altura):
2 ... return (72.7 * altura) - 58
3 ...
4 >>> peso(1.81)
5 73.587000000000018
6 >>> peso(1.61)
7 59.047000000000011
8 >>>

```

### Definição, chamada e argumento

Com o comando **def** definimos a função **peso**. Na realidade, para sermos mais precisos, associamos o nome **def** à definição. A figura 1.13 retrata a situação.

Mas definir uma função não provoca o aparecimento de nenhum valor. Afinal, para que valor de *altura* devia ser feito o cálculo? Por isso, precisamos **chamar**, ou **usar**, a função, explicitando qual o valor **concreto** de *altura*. Já agora: o nome *altura* que aparece na definição, é tecnicamente designado como sendo um **parâmetro formal**. No exemplo acima chamamos duas vezes a função usando os **parâmetros reais**, 1.81 e 1.61. Fica claro que podemos usar a definição tantas vezes quantas as que quisermos. Se

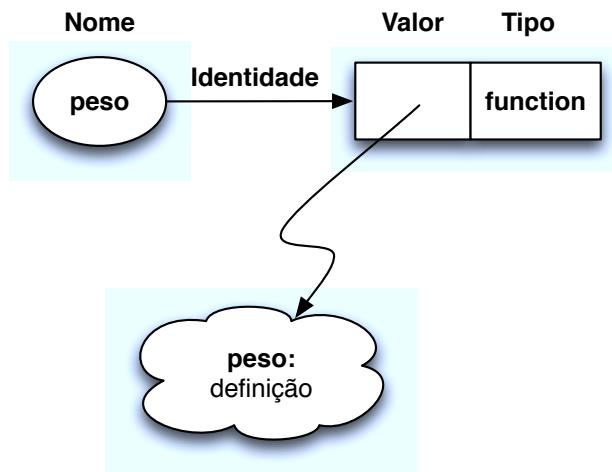


Figura 1.13: Associação entre um nome e uma definição

olharmos para o código verificamos o uso de uma instrução chamada `return` responsável por devolver o resultado da execução da definição a quem lhe solicitar. Neste caso, foi o interpretador que pediu esse valor, e é ele que se responsabiliza por mostrar o resultado, imprimindo-o. Recorde-se: para o interpretador `peso(1.81)` é uma expressão que é lida, o valor associado é calculado, e o resultado visualizado.

Nada obriga a que as nossas definições obtenham os dados através dos parâmetros formais, e devolvam o resultado através da instrução de `return`. Podemos usar dentro das definições as instruções de `input` e de `print`, como já vimos anteriormente:

```

1 >>> def peso():
2 ... altura = eval(input("A sua altura sff: "))
3 ... print("Peso ideal: ", (72.7 * altura) - 58)
4 ...
5 >>> peso()
6 A sua altura sff: 1.81
7 Peso ideal: 73.58700000000002
8 >>>

```

## 1.6 Fazer escolhas

Até aqui os exemplos que vimos o que fazem é um cálculo sequencial seguido da devolução do resultado. Mas a maior parte dos problemas envolve tomada de decisões. Estas por sua vez implicam a execução de testes, que, por seu turno, pressupõem a existência de operações de comparação. É isso que o exemplo muito simples seguinte mostra ser possível de fazer<sup>22</sup>.

```

1 >>> 43 > 27
2 True
3 >>> 4 == 5
4 False
5 >>>

```

**Booleano**

Duas coisas devem ser ditas. Primeiro, o resultado das comparações é um objecto de um novo tipo chamado booleano: **True** ou **False**; em segundo lugar, notar que para comparar a identidade do valor de dois objectos se usam dois sinais de igual<sup>23</sup>. Estamos agora preparados para revisitar o nosso exemplo do peso de uma pessoa conhecida a sua altura. Admitamos que queremos calcular o peso ideal, mas agora para as mulheres. Sabendo que a fórmula não é a mesma, podemos escrever outra função semelhante ao que fizemos para os homens, usando a fórmula adequada a cada caso, homem ou mulher. Mas também podemos juntar tudo, desde que haja uma forma para distinguir os dois casos. Para tal vamos precisar de fazer comparações e usar **instruções de controlo**, que são instruções que não alteram o estado dos objectos e se limitam a indicar qual a parte da nossa definição que deve ser executada.

```

1 >>> def pesohm(altura,genero):
2 ... if genero == 1:
3 ... return (72.7 * altura) - 58
4 ... else:
5 ... return (62.1 * altura) - 44.7
6 ...
7 >>> pesohm(1.81,1)
8 73.587000000000018
9 >>> pesohm(1.61, 0)
10 55.281000000000006
11 >>>

```

---

<sup>22</sup>Trata-se apenas de ilustrar o conceito, existindo muitos mais operadores de comparação.

<sup>23</sup>Mais uma vez este facto dá origem a muitos dos erros de programação cometidos por novatos.

Veja-se como usámos números para codificar o género<sup>24</sup>. A instrução de controlo **if-then-else** faz aquilo que o seu nome indica: se(**if**) o género for masculino então (**then**) usa uma fórmula, senão (**else**), se for feminino usa a outra.

Como anteriormente, os dados necessários podem ser introduzidos pelo utilizador, originando um código ligeiramente diferente.

```

1 >>> def pesohm():
2 ... altura = eval(input("a sua altura por favor: "))
3 ... genero = eval(input("o seu genero por favor: "))
4 ... if genero == 1:
5 ... return (72.7 * altura) - 58
6 ... else:
7 ... return (62.1 * altura) - 44.7
8 ...
9 >>> pesohm()
10 a sua altura por favor: 1.81
11 o seu genero por favor: 0
12 67.701
13 >>> pesohm()
14 a sua altura por favor: 1.61
15 o seu genero por favor:1
16 59.047
17 >>>

```

## 1.7 Repetir

Admitamos agora que queremos repetir estes cálculos para um conjunto de cinco pessoas. Podíamos fazê-lo invocando cinco vezes em sequência o programa **pesohm()**. Mas seria no mínimo um pouco fastidioso. Por isso as linguagens de programação oferecem intruções de controlo de repetição. Neste caso, que sabemos quantas vezes queremos repetir a operação, o nosso programa seria simplesmente o seguinte.

```

1 >>> def pesohm():
2 ... altura = eval(input("a sua altura por favor: "))
3 ... genero = eval(input("o seu genero por favor: "))
4 ... if sexo == 1:

```

---

<sup>24</sup>Quando falarmos de outros tipos de objectos veremos que podemos usar, com mais propósito, cadeias de caracteres.

```

5 ... print(72.7 * altura) - 58
6 ... else:
7 ... print (62.1 * altura) - 44.7
8 ...
9 >>> for i in range(5):
10 ... pesohm()
11 ...
12 a sua altura por favor: 1.81
13 o seu genero por favor: 0
14 67.701
15 a sua altura por favor: 1.90
16 o seu genero por favor: 1
17 80.13
18 a sua altura por favor: 1.45
19 o seu genero por favor: 1
20 47.415
21 a sua altura por favor: 1.60
22 o seu genero por favor: 0
23 54.66
24 a sua altura por favor: 1.75
25 o seu genero por favor: 0
26 63.975
27 >>>

```

O pedaço de código

```

1 >>> for i in range(5):
2 ... pesohm()

```

indica ao interpretador que deve repetir cinco vezes a invocação da função `pesohm()`. É claro que a solução apresentada não é única, nem necessariamente a melhor. Uma alternativa óbvia é colocar a instrução de ciclo no interior da definição `pesohm`.

```

1 >>> def pesohm(n):
2 ... for i in range(n):
3 ... altura = eval(input("a sua altura por favor: "))
4 ... genero = eval(input("o seu genero por favor: "))
5 ... if sexo == 1:
6 ... print(72.7 * altura) - 58
7 ... else:
8 ... print (62.1 * altura) - 44.7
9 ...

```

10 &gt;&gt;&gt;

Notar a introdução de um parâmetro formal (**n**) que nos permite variar o número de vezes que queremos repetir a operação, tornando assim o nosso programa mais geral.



### Indentação

O código **Python** segue regras muito estritas em relação à sua escrita. Em particular, obriga a **alinhar** ou **indentar** cada bloco de código de modo rígido. Esse facto, que no início os programadores não gostam, torna-se um auxiliar precioso na escrita de programas de fácil leitura e depuração. De um modo simples, diremos que precisamos indentar o código cada vez que encontramos **dois pontos**.

## 1.8 Intermezzo

Está na altura de parar um pouco e sistematizar o que estivemos a fazer, quais os conceitos que aprendemos. Em primeiro lugar, falámos de objectos. No caso números de tipos diferentes (inteiros, reais ou em vírgula flutuante)<sup>25</sup>, cadeias de caracteres e booleanos. Em segundo lugar, vimos como podemos associar nomes aos objectos ou, de um modo mais geral, a expressões. Em terceiro lugar, existe um mecanismo de abstracção fundamental que se traduz pela possibilidade de definir funções. Em quarto lugar, as definições são sequências de instruções. Estas podem ter um carácter de manipulação de objectos (atribuição, entrada e saída) ou de controlo (sequência, condicionais ou ciclos) (ver figura 1.14.).

Em quinto lugar, os programas podem envolver mais ou menos interacção com o utilizador, e isso tem implicações quanto ao modo como os objectos são inseridos e/ou extraídos do programa (ver imagem 4.4.).

## 1.9 Módulos

As linguagens de programação devem ser simples e poderosas. Esta dupla característica parece ser contraditória. Em **Python** a questão é resolvida com elegância ao dispormos inicialmente de uma linguagem mínima, mas a que se podem adicionar novas construções e operações que a tornam tão poderosa quanto necessitemos. O conceito que está por detrás desta característica é o conceito de **módulo**.

[Módulos](#)

<sup>25</sup>Existem mais tipos de números como mais tarde se verá.

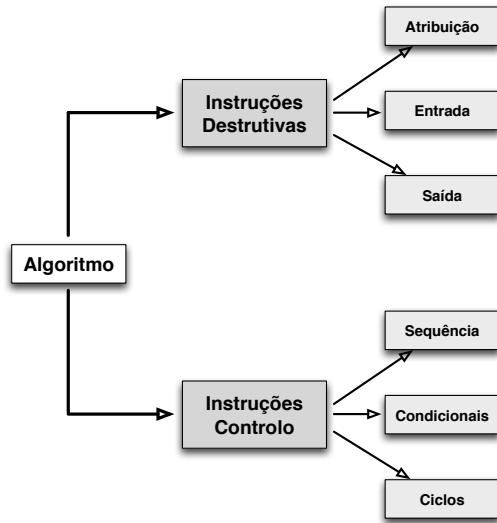


Figura 1.14: Instruções

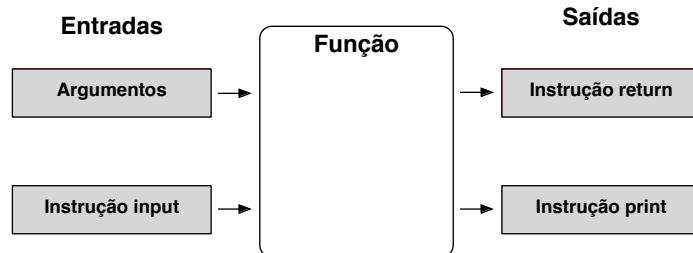


Figura 1.15: Entrada e saída de objectos

Admitamos que queremos calcular o volume de uma esfera, de acordo com a fórmula:

$$\frac{4}{3} \times \pi \times r^3$$

Admitindo que a esfera tem raio 2, qual é o seu volume? Em função do que já sabemos este problema não apresenta ter nenhuma dificuldade. Envolve, no entanto, cálculos com a constante  $\pi$ . Que valor vamos assumir para esta constante? Um valor comum é 3.14. Claro que, usando este valor, o erro no resultado pode ser considerável.

```

1 >>> def volume_esfera(raio):
2 ... return (4/3) * 3.14 * raio ** 3
3 ...
4 >>> volume_esfera(2)
5 33.4933333333
6 >>>

```

Uma maneira de ultrapassar esta questão é recorrer ao valor de  $\pi$  que está definido num **módulo** do sistema. Os módulos são **ficheiros** com código que permite aumentar as capacidades da linguagem de base. Este mecanismo permite que a linguagem base carregada no início da sessão seja simples, necessitando de menos recursos do computador. Caso necessitemos usamos módulos adicionais, que já vem com o sistema ou que são definidos por nós. Um módulo para ser usado tem que ser previamente **importado**. No nosso caso precisamos do módulo **math**, pois é lá que a constante se encontra definida.

```

1 >>> import math
2 >>> math.pi
3 3.1415926535897931
4 >>>

```

Depois de importar o módulo (linha 1 da listagem) temos que usar uma notação especial para **aceder** à constante: começamos com o nome do módulo, seguido de um **ponto**, seguido do nome do objecto (linha 2). Ao inspecionar o objecto  $\pi$  ficamos a saber qual o valor que vai ser usado. Agora só falta a solução para o nosso problema inicial.

```

1 import math
2
3 def volume_esfera(raio):
4 return (4/3) * math.pi * raio ** 3

```

Listagem 1.2: Volume de uma esfera

Os módulos são também objectos. Um modo simples de saber o que nos permitem fazer é inspecionar o objecto recorrendo ao comando **dir**:

```

1 >>> import math
2 >>> dir(math)
3['__doc__', '__file__', '__name__', '__package__', 'acos', ' '
 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', ' '
 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp',
 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', '

```

```

'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', '
pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', '
trunc']
4 >>>

```

Da listagem parece aparente que existe a função **seno**. Verifiquemos isso, por recurso ao comando **help**:

```

1 >>> help(math.sin)
2
3 Help on built-in function sin in module math:
4
5 sin(...)
6 sin(x)
7
8 Return the sine of x (measured in radians).

```

Com este conhecimento podemos efectuar um cálculo específico:

```

1 import math
2 >>> math.sin(45)
3 0.85090352453411844
4 >>>

```

Como veremos mais adiante existem vários modos de importar módulos.

## 1.10 Adivinhar

Suponhamos que queremos um programa para um jogo em que o utilizador procura descobrir um número inteiro, por exemplo entre 0 e 100 que o computador definiu. Com base no que já sabemos uma solução trivial será a seguinte.

```

1 >>> def adivinha(numero):
2 ... if numero == 25:
3 ... return True
4 ... else:
5 ... return False
6 ...
7 >>> adivinha(33)
8 False
9 >>> adivinha(25)
10 True
11 >>>

```

Como solução temos que admitir que não é muito famosa. Vamos procurar melhorar. Em primeiro lugar, vamos mudar o número cada vez que jogamos, efectuando uma escolha aleatória.

```

1 >>> import random
2 >>> def adivinha(numero):
3 ... secreto = random.randint(0,100)
4 ... if numero == secreto:
5 ... return True
6 ... else:
7 ... return False
8 ...
9 >>> adivinha(33)
10 False
11 >>> adivinha(25)
12 False
13 >>>

```

Para resolver a questão tivemos que importar o módulo `random` e usar o método `randint`. De um modo informal, um método é equivalente a uma definição, mas que apenas pode ser aplicado a um certo tipo de objectos. Uma outra melhoria natural será permitir que o utilizador tenha mais do que uma tentativa. Esta é uma situação típica de problemas em que existe interacção com o utilizador, por isso `Python`, com naturalidade, fornece como já sabemos uma instrução para isso mesmo: `input`.

```

1 >>> def adivinha():
2 ... secreto = random.randint(0,100)
3 ... numero = eval(input("O seu palpito: "))
4 ... if numero == secreto:
5 ... return True
6 ... else:
7 ... return False
8 ...
9 >>> adivinha()
10 O seu palpito: 33
11 False
12 >>> adivinha()
13 O seu palpito: 25
14 False
15 >>>

```

[Métodos](#)

Mais uma vez, atente-se no facto de ser possível uma definição não ter parâmetros formais. Mas regressemos à questão da repetição. Como fazer? Suponhamos que apenas damos duas tentativas. Podemos usar a nossa solução com pequenos ajustes.

```

1 >>> def adivinha():
2 ... secreto = random.randint(0,100)
3 ... numero = eval(input("O seu palpito: "))
4 ... if numero == secreto:
5 ... return True
6 ... numero = eval(input("O seu palpito: "))
7 ... if numero == secreto:
8 ... return True
9 ... else:
10 ... return False
11 ...
12 >>> adivinha()
13 O seu palpito: 33
14 O seu palpito: 44
15 False
16 >>>

```

Mas isto não é prático. E se em vez de duas tentativas tivermos dez, ou mesmo cem? Python vai ajudar-nos com recurso à instrução de controlo conhecida por repetição ou **ciclo**, já anteriormente referida.

```

1 >>> def adivinha(tentativas):
2 ... secreto = random.randint(0,100)
3 ... for i in range(tentativas):
4 ... numero = eval(input('O seu palpito: '))
5 ... if numero == secreto:
6 ... return True
7 ... return False
8 ...
9 >>> adivinha(3)
10 O seu palpito: 33
11 O seu palpito: 25
12 O seu palpito: 44
13 False

```

Não pretendemos que o leitor apreenda já toda a complexidade da instrução **for**. Os ciclos serão talvez o conceito mais anti-natural para quem chega pela primeira vez à programação. Serão por isso objecto de exploração

detalhada ao longo do texto. Fazemos apenas notar que a primeira vez que uma instrução de `return` for encontrada e executada o programa termina e devolve o valor que lhe está associado. Deste modo, terminado o ciclo após ser executado o número de vezes por nós definido no parâmetro `tentativas` sem encontrar a solução, podemos devolver `False`. Para concluir, vamos tornar o programa um pouco mais justo para o utilizador, recorrendo agora à instrução `print` e exemplos de objectos do tipo **cadeias de caracteres**.

```
1 >>> import random
2 >>> def adivinha(tentativas):
3 ... secreto = random.randint(0,100)
4 ... for i in range(tentativas):
5 ... numero = eval(input("O seu palpito sff: "))
6 ... if numero == secreto:
7 ... print("Uau, acertou!")
8 ... return True
9 ... else:
10 ... if numero > secreto:
11 ... print("Muito grande...")
12 ... else:
13 ... print("Muito pequeno...")
14 ... print("Lamento, mas esgotou as suas tentativas.")
15 ... return False
16 ...
17 >>> adivinha(3)
18 O seu palpito sff: 33
19 Muito pequeno...
20 O seu palpito sff: 55
21 Muito pequeno...
22 O seu palpito sff: 80
23 Muito grande...
24 Lamento, mas esgotou as suas tentativas.
25 False
26 >>>
```

## 1.11 Modo não interactivo

Os pequenos exemplos de código que vimos até aqui foram todos criados directamente no interpretador. Chamamos a esta forma de usar o sistema, trabalhar em **modo interactivo**. É útil para fazer pequenas coisas. Mas tem

os seus inconvenientes. Por exemplo, sempre que nos enganamos temos que recomeçar tudo de novo. Outro aspecto prende-se com o facto de em geral o que escrevemos no interpretador perder-se quando saímos do interpretador, sendo necessário escrever de novo todos os comandos quando iniciamos nova sessão se pretendemos repetir as coisas. O que normalmente se faz é escrever o código do programa por recurso a um editor de texto, guardar tudo num ficheiro e depois, quando queremos usar o código importar o ficheiro como fizemos com os módulos pré-definidos do sistema. Vejamos um exemplo.

Suponhamos que queremos calcular a raiz quadrada de um número positivo. Um modo simples de o fazer é através do método de Newton. Por definição temos:

$$\begin{cases} x_0 & \approx \sqrt{a} \\ x_{n+1} & = \frac{1}{2} \times (x_n + \frac{a}{x_n}) \end{cases}$$

Com esta definição inductiva o que temos que fazer é definir iterativamente uma sequência de valores para a raiz quadrada, sendo que sucessivos valores aproximam melhor o valor real. Como o processo tem que parar, temos que definir o tamanho da sequência. Suponhamos que abrimos um editor de texto e escrevemos o código como o indicado en [1.3](#).

```

1 # -*- coding: mac-roman -*-
2
3 def raizquad(a):
4 """Cálculo da raiz quadrada de um número positivo pelo
5 método de Newton."""
6 print("Raiz quadrada de: ",a)
7 x = eval(input("Valor inicial sff: "))
8 for i in range(10):
9 x = (1/2.0) * (x + (a/x))
10 return x
11
12 if __name__ == '__main__':
13 print((raizquad(2)))

```

Listagem 1.3: Raiz quadrada

O código tem duas partes: a primeira (linhas 3 a 9), consiste na definição da função que permite o cálculo da raiz quadrada; a segunda (linhas 11 e 12), permite o uso da definição<sup>26</sup>. A listagem abaixo ilustra uma forma de

---

<sup>26</sup>O ficheiro inclui também (linha 1) uma directiva ao sistema sobre a codificação usada.

executar o programa, baseada na chamada explícita do interpretador seguido do nome do ficheiro.

```
1 ernestojfcosta@Ernesto-Costas-Mac-Pro-8 $ python newton.py
2 Raiz quadrada de: 2
3 Valor inicial sff: 1
4 1.414213562373095
```

O ficheiro por nós criado chama-se **newton.py**. Terminar com a extensão **py** é importante. Neste exemplo aparece também uma cadeia de caracteres especial, logo a seguir ao **cabeçalho** da definição. Trata-se de um comentário [Comentários](#) sobre o que faz o programa e a sua colocação naquela zona é importante para efeitos de documentação do programa<sup>27</sup>.

O modo como está escrito o ficheiro permite que este seja também usado em modo interactivo.

```
1 >>> import newton
2 >>> newton.raizquad(4)
3 Valor inicial sff: 5
4 2.0
5 >>> newton.raizquad(2)
6 Valor inicial sff: 3
7 1.41421356237
8 >>>
```

É o código das linhas 11 e 12

```
1 if __name__ == '__main__':
2 print raizquad(2)
```

que permite esta dupla utilização.

## Sumário

Neste capítulo vimos como um computador é um misto de *hardware* e de programas. Descrevemos de forma básica a arquitectura de um computador. Concentrámo-nos nos programas (compiladores e interpretadores) que permitem a execução de outros programas escritos numa linguagem de alto nível. Demos exemplos dos quatro paradigmas de programação principais. Terminámos com alguns exemplos de uso da linguagem Python. No caso de Python demos os primeiros passos que nos levarão à construção de programas de complexidade elevada. Em particular,

---

<sup>27</sup>Incluindo a documentação automática.

- aprendemos os conceitos de objecto, expressões, instruções e módulos,
- os objectos têm atributos: identidade, valor, tipo,
- identificámos alguns tipos de objectos, como inteiros, reais, cadeias de caracteres e booleanos,
- aprendemos a associar nomes a objectos com o operador de atribuição `=`,
- aprendemos a ler e a imprimir valores,
- aprendemos a associar um conjunto de comandos ou instruções a um nome com `def`,
- vimos que as nossas definições podem ter parâmetros,
- aprendemos a controlar escolhas,
- vimos como repetir conjuntos de comandos,
- ficámos a saber importar módulos para expandir a linguagem.

## **Testes os seus conhecimentos**

Tente responder às seguintes questões que foram tratadas neste capítulo.

1. Quais as componentes de base da arquitectura de um computador.
2. Compiladores e interpretadores: o que são e quais as diferenças.
3. Que tipo de paradigmas de programação conhece. O que os distingue.
4. O que são módulos, expressões, instruções, e objectos.
5. O que entende por parâmetros formais.
6. O que faz a instrução de atribuição (`=`).
7. Que instruções de controlo conhece e para que servem.
8. Diga o que entende pelo modelo PCAP.
9. Diga o que entende por ciclo Lê - Avalia - Escreve.

## Exercícios

**Exercício 1.1 MF** Determine como é que na sua plataforma se pode por o interpretador Python a correr.

**Exercício 1.2 MF** Python pode ser usado como uma simples calculadora. Verifique o resultado das seguintes computações.

1.  $2 + 4$
2.  $40 * 300$
3.  $1/2$
4.  $1.0/2$
5.  $1.0 // 2$
6.  $20\text{e}30 * 4$
7.  $20\text{e}50 * 20\text{e}50$
8.  $7 \% 5$
9.  $(5 + 2j) + (3 + 4j)$
10.  $(5 + 2j) * (3 + 4j)$
11.  $(5 + 2j) / (3 + 4j)$

**Exercício 1.3 MF** A galáxia Andrómena está a 2,9 milhões de anos-luz da Terra. Um ano luz equivale  $9.459 \times 10^{12}$  quilómetros. A quantos quilómetros se encontra a galáxia da Terra?

**Exercício 1.4 MF**

Calcule o número de segundos que existe num ano *normal*.

**Exercício 1.5 MF**

Suponha que tem uma sala rectangular de dimensão  $8 \times 6$ . Admitindo que quer cobrir o chão com tijoleira de  $2 \times 2$ , calcule o número de unidades de que vai precisar.

**Exercício 1.6 MF** Escolha objectos numéricos de tipos diferentes e inspecione os seus três atributos.

**Exercício 1.7 MF** A área de um triângulo é igual a metade do produto do comprimento de um dos lados pela distância ao vértice oposto medida perpendicularmente. Diga como podia usar **Python** para calcular o valor concreto da área de um triângulo conhecidos aqueles valores.

**Exercício 1.8 F** Você não gosta de ser enganado e é muito meticuloso. Quando foi comer à sua hamburgeria preferida foi confrontado com uma nova forma: agora o hamburger é um quadrado de lado  $7.62\text{ cm}$  (eu avisei que você era meticuloso!). Para saber se devia protestar (sim porque você adora uma boa luta ...), procurou comparar com o formato antigo, bem redondo como uma circunferência de diâmetro  $8.89\text{ cm}$  (precisa que eu insista em como é meticuloso?). Eu que eu quero saber é se você, consumidor compulsivo de carne picada, tem ou não razões para protestar devido ao design do novo hamburger.

**Exercício 1.9 F** O **Índice de Massa Corporal** é dado pela fórmula:

$$IMC = \frac{\text{peso}(kg)}{\text{altura}^2(m^2)}$$

Refaça os cálculos da secção 1.5 adaptando o exemplo do peso para o caso do IMC.

**Exercício 1.10 F** Escreva um programa que lhe permita converter uma temperatura na escala Celsius ( $T_c$ ) na escala Fahrenheit ( $T_f$ ), baseando-se na fórmula:

$$T_f = \frac{9}{5} \cdot T_c + 32$$

**Exercício 1.11 F** Escreva um programa que lhe permita calcular o volume de um cone, conhecidos o raio da base  $r$  e a altura  $h$ . O volume pode ser calculado pela fórmula:

$$V = \frac{\pi \cdot r^2 \cdot h}{3}$$

**Exercício 1.12 F**

Suponha que tem o seguinte polinómio:  $x^4 + x^3 + 2x^2 - x$ . Socorrendo-se da linguagem **Python** calcule o valor do polinómio nos seguintes pontos:

1.  $x = 1.1$
2.  $x = 5$
3.  $x = \frac{2}{3}$

**Exercício 1.13 F** Importe o módulo **math** e experimente as várias funções fornecidas. O que acontece se tentar calcular a raiz quadrada de um inteiro negativo.

**Exercício 1.14 F** Suponha que quer trocar uma certa quantidade de euros por dólares americanos, conhecida a taxa de câmbio. Diga como pode resolver o problema socorrendo-se de **Python** para um caso concreto. Se por acaso quer uma solução genérica, em que medida a sua solução anterior lhe resolve a questão?

**Exercício 1.15 F** Suponha que tem uma certa quantidade de garrafas vazias de capacidade 5, 1.5, 0.5 e 0.25 litros. Admita que tem um número ilimitado de garrafas de cada tipo. Dado um certa quantidade de água que pretende guardar em garrafas, como ressolveria o problema minimizando o **número** de garrafas a usar. Como poderia usar o computador para lhe calcular o número de garrafas de cada tipo necessárias?

**Exercício 1.16 M** Volte ao exemplo do jogo da adivinha do número e procure definir novas variantes para o jogo, de modo a torná-lo mais interessante e realista.

**Exercício 1.17 Módulo math M**

Um ponto no plano pode ser identificado pelas suas coordenadas cartesianas (par  $(x,y)$ ) ou pelas coordenadas polares (par  $((r,\theta))$ ). Escreva um programa que converte das coordenadas cartesianas para as polares. A relação entre os dois tipos de representação é dada pelas fórmulas que se apresentam na figura 9.4.

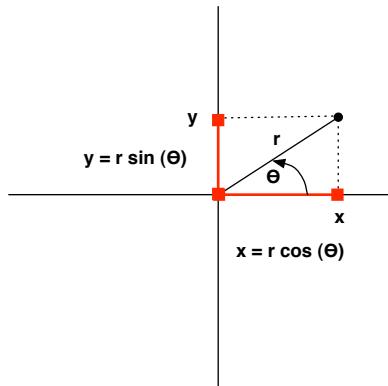


Figura 1.16: De cartesianas a polares

**Exercício 1.18** Módulo math M Johannes Kepler<sup>28</sup> foi um cientista que no início do século XVII formulou três leis relativas ao movimento dos planetas em torno do Sol, com base nas observações do astrónomo Tycho Brahe. A primeira lei, estipula que as órbitas são elipsoidais, com o Sol num dos focos; a segunda lei, estabelece que a linha que une o planeta ao Sol varre áreas iguais durante intervalos de tempo iguais; a terceira lei, diz que o quadrado do período orbital de um planeta é directamente proporcional ao cubo do semi-eixo maior da sua órbita, ou seja que  $p^2 = a^3$ . A distância de um planeta ao Sol é dada em **Unidades Astronómicas**, AU. 1 AU é igual ao valor do semi-eixo maior da órbita da Terra em volta do sol e vale  $149.597890 \times 10^6 \text{ km}$ .

No seguimento de Johannes Kepler, Isaac Newton, publicou em 1687 o seu livro *Principia Mathematica*, onde formulou a sua teoria sobre a gravidade. No contexto da teoria, Newton generaliza a terceira lei de Kepler, que deixa de estar limitada ao sistema solar:

$$p^2 = \frac{4\pi^2}{G(M_1 + M_2)} a^3$$

sendo que  $G = 6.67 \times 10^{-11} \text{ Newton m}^2/\text{Kg}^2$  é a constante gravitacional, e  $M_1$  e  $M_2$  a massa de dois objectos no espaço.

Escreva um programa que permita calcular o período, em anos, da órbita de um planeta, conhecida a sua distância ao Sol em AUs. Para o auxiliar a verificar a correcção do seu programa deve consultar [http://en.wikipedia.org/wiki/Johannes\\_Kepler](http://en.wikipedia.org/wiki/Johannes_Kepler).

<sup>28</sup>[http://en.wikipedia.org/wiki/Johannes\\_Kepler](http://en.wikipedia.org/wiki/Johannes_Kepler)

[org/wiki/Attributes\\_of\\_the\\_largest\\_solar\\_system\\_bodies](https://en.wikipedia.org/wiki/Attributes_of_the_largest_solar_system_bodies), onde encontrará os valores de teste de que necessita.

**Exercício 1.19 M** Sabemos hoje que a formulação de Newton é mais geral que a de Kepler. Vamos tentar resolver o problema de determinar o período orbital de qualquer corpo que orbita em volta de qualquer estrela. Para facilitar a nossa vida vamos escolher uma estrela concreta: **Gliese 581** (ver dados em [http://pt.wikipedia.org/wiki/Gliese\\_581](http://pt.wikipedia.org/wiki/Gliese_581)).



# Capítulo 2

## Visões (I)

### Objectivos

- ✓ Introduzir o módulo gráfico `turtle`
- ✓ Revisitar os conceitos básicos de programação
- ✓ Introduzir a diferença entre função e método
- ✓ Exemplificar o desenho de gráficos de funções

### 2.1 Coisas que mexem

Até agora temos andado ocupados com objectos numéricos (inteiros, vírgula flutuante e complexos), cadeias de caracteres, e booleanos<sup>1</sup>. Sabemos que os objectos têm três características: identidade, valor e tipo. Sabemos ainda que os objectos podem ter outros atributos: por exemplo, os nomes associados. Com estes objectos podemos fazer diferentes manipulações como, por exemplo, operações como soma, multiplicação de números). Está na altura de introduzir um outro **tipo** de objectos, com atributos interessantes, e com os quais se pode fazer operações distintas das aritméticas. Vamos falar de **tartarugas**. Isso mesmo, tartarugas.

As tartarugas são objectos que vivem num mundo a duas dimensões. Têm por isso uma **posição** e uma **direcção**. São várias as operações (i.e., métodos) que podem ser efectuadas sobre objectos do tipo tartaruga. Por

---

<sup>1</sup>Claro que também já sabemos que os módulos e as definições são objectos, porque em Python ... tudo são objectos!

exemplo, as tartarugas podem **mover-se** (para a frente, para trás), e podem **rodar** (para a esquerda, para a direita). Enquanto se deslocam podem deixar (ou não) no chão um **rastro**, isto porque cada tartaruga tem associada uma **caneta**. A caneta também pode ser controlada, levantando-a ou baixando-a, alterando a sua cor ou a espessura do seu traço. A imagem 2.1 mostra uma tartaruga e o rastro que deixou enquanto se deslocou. A ponta da seta simboliza a tartaruga.

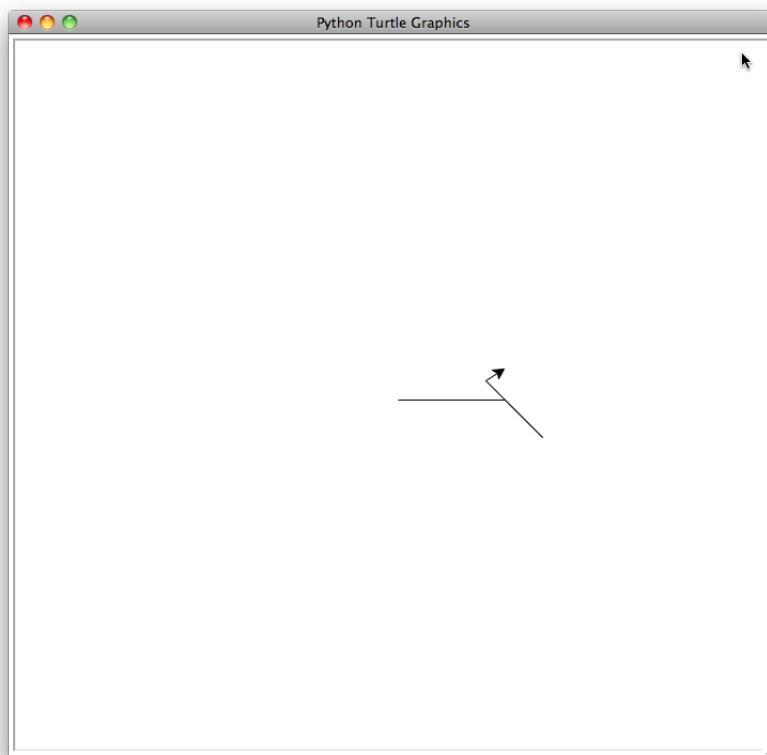


Figura 2.1: O passeio (curto) da tartaruga

Para comandar a tartaruga temos que importar o módulo **turtle**. Quando iniciamos o comando da tartaruga esta encontra-se na posição  $(x,y) = (0,0)$  com uma orientação de 0 graus, isto voltada para leste, paralela a um eixo dos **x** imaginário. O desenho da figura 2.1 foi obtido com a seguinte sessão **interactiva**:

```

1 >>> import turtle
2 >>> turtle.forward(100)
3 >>> turtle.right(45)

```

```
4 >>> turtle.forward(50)
5 >>> turtle.backward(75)
6 >>> turtle.left(80)
7 >>> turtle.forward(20)
8 >>>
```

O comando **forward(n)** faz avançar a tartaruga *n* unidades, **right(a)** faz a tartaruga rodar para a direita um ângulo de *a* graus. Por seu lado, **backward(n)** faz recuar a tartaruga *n* unidades e **left(a)** faz a tartaruga rodar para a esquerda um ângulo de *a* graus.

Depois de executar estes comandos a posição e direcção são diferentes dos valores iniciais e podem ser **consultados**:

```
1 >>> turtle.position()
2 (98.71, 29.15)
3 >>> turtle.heading()
4 35.0
5 >>>
```

Como se vê no exemplo acima, quando importamos o módulo temos que prefixar todos os comandos pelo nome do módulo, como fazemos com qualquer módulo.

Existem outras operações relacionadas com o movimento e com o estado da tartaruga. Eis algumas:

```
1 >>> import turtle
2 >>> turtle.goto(100,100)
3 >>> setx(200)
4 >>> turtle.setx(200)
5 >>> turtle.ycor()
6 100
7 >>> turtle.xcor()
8 200
9 >>> turtle.setheading(225)
10 >>>
```

Podemos controlar a posição (linha 2) e orientação (linha 9) da tartaruga de modo absoluto, e consultar e alterar a posição por componentes (linhas 3 a 8). No anexo A o leitor encontrará um manual breve de referência ao módulo.

## 2.2 Tartarugas e geometria

### Quadrados

Armados deste conhecimento primitivo, podemos começar a fazer pequenos desenhos. Os exemplos serão escritos no editor e vamos fazer uso da importação selectiva. Vamos começar por escrever um programa que permita desenhar um quadrado com um dado comprimento do lado. Trata-se de um problema muito simples. A solução passa por desenhar um lado, rodar  $90^\circ$ , desenhar o segundo lado, rodar  $90^\circ$ , repetindo quatro vezes esta sequência de dois comandos.

```

1 turtle. forward(50)
2 turtle.right(90)
3 turtle.forward(50)
4 turtle.right(90)
5 turtle.forward(50)
6 turtle.right(90)
7 turtle.forward(50)
8 turtle.right(90)

```

A primeira coisa que salta à vista é o facto do comprimento do lado ser fixo. Cada vez que quisermos um quadrado com o lado diferente vamos ter que editar o programa e alterar o valor do comprimento. Temos então uma óptima oportunidade para usar o mecanismo de **abstracção procedimental**, desenvolvendo um programa em que o comprimento do lado é um parâmetro.

```

1 import turtle
2
3 def quadrado(lado):
4 """
5 Desenha um quadrado de lado, lado.
6 """
7 turtle.forward(lado)
8 turtle.right(90)
9 turtle.forward(lado)
10 turtle.right(90)
11 turtle.forward(lado)
12 turtle.right(90)
13 turtle.forward(lado)
14 turtle.right(90)
15

```

```

16
17 if __name__ == '__main__':
18 meu_lado = 50
19 quadrado(meu_lado)

```

O resultado visual não é difícil de antever (Figura 8.5).

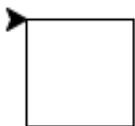


Figura 2.2: Mas que lindo quadrado

Notar que, ao efectuar uma quarta rotação, a direcção da tartaruga á a mesma que a original. Esta solução, estando correcta, não é elegante. Em particular, é evidente que estamos a repetir um número fixo de vezes um par de comandos. Como sabemos existem instruções que nos permitem efectuar repetições, de modo a tornar o código curto, elegante e legível. Para situações como esta, padrão de contagem simples e com um número de repetições fixo, existe uma instrução de controlo que torna o programa mais simples e legível: a instrução repetitiva **for**. Vamos usá-la.

```

1 def quadrado(lado):
2 """
3 Desenha um quadrado de lado lado.
4 """
5 for i in range(4):
6 turtle.forward(lado)
7 turtle.right(90)

```

A variável **i** faz o papel de contador, e a operação **range** gera os sucessivos valores do contador, no nosso caso 0, 1, 2, 3. Será que ainda podemos generalizar mais o nosso código? Claro que sim. Uma possibilidade consiste em tornar variável a posição e a orientação inicial da tartaruga, logo do quadrado. Uma vez mais, a generalização traduz-se pela introdução de dois parâmetros adicionais.

```

1 import turtle
2
3 def quadrado(lado, pos, angulo):
4 """
5 Desenha um quadrado com o comprimento de lado, o vértice
6 inferior
7 esquerdo em pos e direcção inicial angulo.
8 """
9 # Preparação
10 turtle.goto(pos)
11 turtle.setheading(angulo)
12 # Desenha
13 for conta in range(4):
14 turtle.forward(lado)
15 turtle.right(90)
16 turtle.hideturtle()

```

Listagem 2.1: Um quadrado com o rabo de fora

Nesta solução acrescentámos um novo comando, `hideturtle`, que, como o nome deixa antever, permite esconder a tartaruga. Como é natural existe o comando inverso, `showturtle`, que torna visível a tartaruga. Ao executarmos o programa o resultado não é muito famoso como se pode ver na figura 2.3.

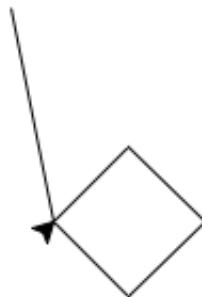


Figura 2.3: Um quadrado defeituoso

Acontece que quando iniciamos a tartaruga a caneta que lhe está associada, por defeito, encontra-se em baixo. Logo, ao executar o comando de

repositionamento da tartaruga ela vai deixar o rastro do seu movimento. Nada que não se possa resolver recorrendo a comandos que levantam e baixam a caneta.

```

1 import turtle
2
3 def quadrado(lado, pos, angulo):
4 """
5 Desenha um quadrado com o comprimento de lado, o vértice
6 inferior
7 esquerdo em pos e direcção inicial angulo.
8 """
9 # Preparação
10 turtle.penup()
11 turtle.goto(pos)
12 turtle.setheading(angulo)
13 turtle.pendown()
14 # Desenha
15 for conta in range(4):
16 turtle.forward(lado)
17 turtle.right(90)
18 turtle.hideturtle()
```

Listagem 2.2: Um quadrado perfeito

### Polígonos regulares

Suponhamos que nos é pedido agora para desenhar um triângulo equilátero. Adaptando a solução anterior facilmente chegamos ao resultado pretendido.

```

1 def tri_equi(lado):
2 """
3 Desenha um triângulo equilátero e lado lado.
4 """
5 for i in range(3):
6 turtle.forward(lado)
7 turtle.right(120)
8 turtle.hideturtle()
```

E se for um pentágono? Sem grande dificuldade chegamos ao código seguinte:

```

1 def pentagono(lado):
2 """
```

```

3 Desenha um pentágono.
4 """
5 for i in range(5):
6 turtle.forward(lado)
7 turtle.right(72)
8 turtle.hideturtle()

```

Olhando para o código dos três polígonos regulares podemos descobrir algum padrão? Sim: o produto do número de lados pelo ângulo é sempre igual a 360 graus. Nada que a geometria não nos tivesse ensinado. A descoberta de um padrão abre a porta à generalização do código<sup>2</sup>. Sempre que generalizamos o nosso código torna-se possível **reutilizá-lo** em diferentes situações. No nosso caso podemos criar um programa para desenhar polígonos regulares.

```

1 def poligono_regular(comp_lado, num_lados):
2 """
3 Desenha um polígono regular.
4 """
5 angulo_viragem = 360 /num_lados
6 # Desenha
7 for i in range(num_lados):
8 turtle.forward(comp_lado)
9 turtle.right(angulo_viragem)
10 turtle.hideturtle()

```

Deixamos ao leitor o cuidado de testar o programa para diferentes polígonos. Mas não resistimos a mostrar-lhe algo. admita que chama o programa com `poligono_regular(1, 360)`. O resultado é o da figura 2.4: um circunferência!

## 2.3 Intermezzo

Até aqui temos admitido que só existe uma tartaruga. Mas sabemos que existem vários objectos de um dado tipo. Por exemplo, existe uma infinidade de números inteiros. Mesmo um tipo tão limitado como os booleanos tem dois objectos. Cada tipo tem associado um **construtor** que permite criar um objecto do tipo. Se a operação não tiver argumento, cria um objecto particular, se tiver tentar criar um objecto do tipo a partir do objecto fornecido. O construtor para as tartarugas chama-se `Turtle`. Com ele podemos criar vários objectos do tipo e associá-los a diversos nomes. Acontece

---

<sup>2</sup>Generalizar é outro modo de dizer abstrair.

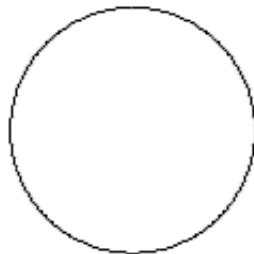


Figura 2.4: Uma bela circunferência

que, até aqui, usámos tartarugas sem criar objectivamente nenhuma. Como é que tal é possível? A resposta reside no facto de o módulo `turtle` ter uma duplicação das operações<sup>3</sup>. Assim, quando não é criada nenhuma tartaruga recorrendo ao construtor, usamos as funções da forma usual:

`módulo.operação(argumentos).`

como em:

`turtle.forward(100).`

Vejamos agora o recurso ao construtor. A listagem abaixo exemplifica como podemos criar dois objectos do tipo tartaruga. Como qualquer outro objecto, eles têm as três características já referenciadas: identidade, valor e tipo. Como se pode ver os objectos têm identidades diferentes.

```

1 >>> import turtle
2 >>> toto = turtle.Turtle()
3 >>> toto
4 <turtle.Turtle object at 0x1007b2910>
5 >>> type(toto)
6 <class 'turtle.Turtle'>
7 >>> id(toto)
8 4303038736
9 >>> titi = turtle.Turtle()
```

---

<sup>3</sup>Tecnicamente, num caso usamos implementações procedimentais, isto é funções, e, no outro caso, usamos uma abordagem orientada aos objectos, isto é métodos. Mais adiante este aspecto será clarificado.

```

10 >>> titi
11 <turtle.Turtle object at 0x1014c8c10>
12 >>> type(titi)
13 <class 'turtle.Turtle'>
14 >>> id(titi)
15 4316761104

```

Agora podemos comandar as tartarugas de modo independente. Para o fazer usamos **métodos** próprios para os objectos do tipo **turtle**<sup>4</sup>.

Os métodos são invocados por:

**objecto.método(argumentos).**

Na realidade, quando **invocamos** um método são executadas **duas** acções: primeiro, vai-se obter o método associado ao objecto através do seu nome; segundo, executa-se o método sobre o objecto, passando-lhe os argumentos, ou seja, fazemos:

**método(objecto,argumentos).**

Quando usamos funções na realidade o que fazemos é apenas este segundo passo, sendo que o objecto é um objecto genérico criado internamente, e que não precisa ser referenciado. É esta criação implícita que permite trabalhar com uma tartaruga sem a ter criado. As duas situações podem estar misturadas como se pode observar na listagem abaixo.

```

1 >>> import turtle
2 >>> toto = turtle.Turtle()
3 >>> toto.pensize(3)
4 >>> toto.forward(100)
5 >>> toto.write('toto', font=("Arial",14,"bold"))
6 >>> turtle.backward(100)
7 >>> turtle.write('anónima',font=("Arial",14,"bold"))

```

No código acima nós criámos uma tartaruga de nome **toto** e o sistema criou uma tartaruga "anónima". Quando indicamos explicitamente o nome do objecto é sobre esse que se aplica o método, quando não indicamos, e referimos apenas o nome do módulo, então o que é aplicado à tartaruga anónima é a função correspondente. O resultado da execução do código acima pode ser visto na figura 2.5 onde se evidencia a existência das duas

---

<sup>4</sup>(Para o leitor curioso) Quando perguntámos pelo tipo a resposta foi `<class 'turtle.Turtle'>`. A razão é porque em Python os tipos são implementados como classes. Usaremos as duas palavras de modo indistinto. Mas isto leva-nos para conceitos mais avançados, da programação orientada aos objectos, que trataremos mais à frente.

tartarugas a funcionar de modo independente. Para tornar a figura de mais fácil interpretação usámos outros comandos, um que altera a espessura do traço e outro que permite escrever texto.



Figura 2.5: Duas tartarugas independentes

Vejamos mais um exemplo de comando independente de duas tartarugas.

```

1 import turtle
2
3 def salta_tartaruga(tarta,distancia):
4 tarta.pu()
5 tarta.forward(distancia)
6 tarta.pendown()
7
8
9 if __name__ == '__main__':
10 toto = turtle.Turtle()
11 toto.color('gray')
12 toto.shape('turtle')
13 titi = turtle.Turtle()
14 titi.shape('triangle')
15 salta_tartaruga(toto,100)
16 titi.right(180)
17 salta_tartaruga(titi,100)
18 turtle.exitonclick()

```

Ao executar obtemos o resultado da figura 2.6. Notar o recurso a outras possibilidades do módulo `turtle`: mudar a cor e a forma da tartaruga. Também usamos o método `exitonclick` que nos permite controlar o fim da execução do programa através do fecho da janela usada para desenhar.

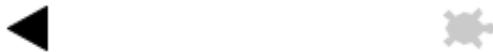


Figura 2.6: De costas voltadas

### Funções e Métodos

Numa linguagem muito simples podemos dizer que os **métodos** de um tipo, são em geral, **funções específicas** para os objectos do tipo. Tecnicamente, os métodos são atributos dos objectos que referenciam funções.

## 2.4 Gráficos de funções

Um problema comum consiste em desenhar o gráfico de uma dada função. Podemos usar o módulo **turtle** para resolver esta questão. A ideia baseia-se em aproximar a função através de uma sequência de segmentos de recta. Se tivermos uma função  $y = f(x)$ , geramos sucessivos valores de  $x = x_0, x_1, \dots$ , calculamos os respectivos  $y_0 = f(x_0), y_1 = f(x_1), \dots$ . Traçamos depois os segmentos que resultam de unir os pontos consecutivos  $((x_i, y_i), (x_{i+1}, y_{i+1}))$ . Vejamos como podemos implementar esta ideia simples. Vamos escolher a função seno, e fazer variar o ângulo entre  $-\pi$  e  $\pi$ .

```

1 import turtle
2 import math
3
4 def grafico(tartaruga, funcao, inicio, fim, n):
5 """ Desenha o gráfico da função f entre inf e sup usando n
6 segmentos."""
7 tam_seg = (fim - inicio)/n
8 # Posiciona-se
9 x = inicio
10 tartaruga.up()

```

```

11 tartaruga.goto(x, funcao(x))
12 tartaruga.down()
13 # Desenha
14 for conta in range(n):
15 x = x + tam_seg
16 tartaruga.goto(x, funcao(x))
17
18 if __name__ == '__main__':
19 turtle.setworldcoordinates(-math.pi, -2, math.pi, 2)
20 toto = turtle.Turtle()
21 toto.pen(pensize=5, pencolor='gray')
22 grafico(toto,math.sin, -math.pi,math.pi,50)
23 toto.hideturtle()
24 turtle.exitonclick()

```

Esta solução precisa saber os valores inicial e final e o número de segmentos que vamos usar. Com base nestes três valores é possível calcular a separação entre cada par de valores consecutivos (linha 7). De seguida posicionamos a tartaruga na posição inicial sem deixar rasto (linhas 9 a 12). Entramos depois num ciclo em que se obtém a segunda extremidade de cada segmento e se usa a função `goto` para efectuar o desenho (linhas 14 a16). Ao correr o programa (linhas 19 a 24), obtemos o gráfico da figura 2.7.

O número de segmentos usado determina a qualidade da aproximação. Experimente com diferentes valores e verifique esse fenómeno. O leitor atento terá notado o uso de uma operação chamada `setworldcoordinates`. Esta operação é fundamental para a visualização pois adapta a janela à escala de valores com que estamos a trabalhar. Se não a usarmos na prática não se vê o gráfico. Tem quatro argumentos: os dois primeiros definem o canto inferior esquerdo da janela, e os dois últimos o canto superior direito. No desenho de gráficos de funções estamos habituados a ver os eixos. Não é difícil juntar essa informação. Para tal definimos uma operação auxiliar que nos permita traçar um segmento de recta entre dois pontos. Com esta alteração o programa será agora o da listagem 2.3.

```

1 import turtle
2 import math
3
4 def linha(x1,y1,x2,y2):
5 """ Traça uma linha entre dois pontos."""
6 turtle.up()
7 turtle.goto(x1,y1)
8 turtle.pd()

```

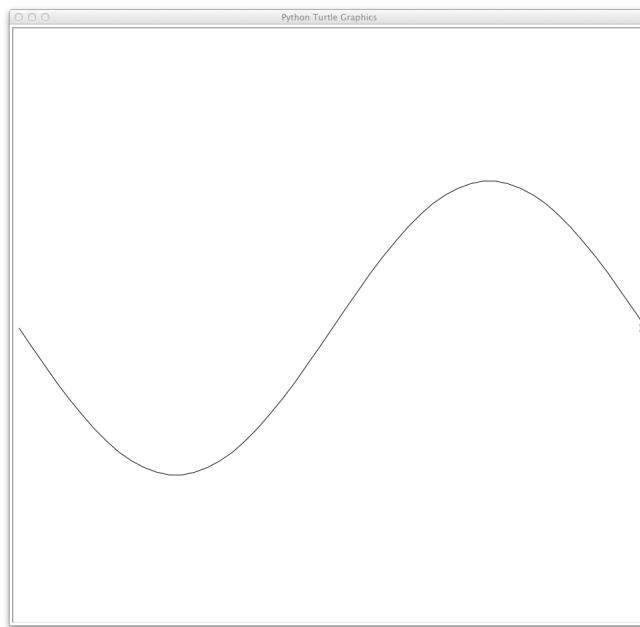


Figura 2.7: A função seno

```
9 turtle.goto(x2,y2)
10 turtle.up()
11 turtle.hideturtle()
12
13 def grafico(tartaruga,funcao, inicio,fim, n):
14 """ Desenha o gráfico da função f entre inf e sup usando n
15 segmentos."""
16 # Tamanho dos segmentos
17 tam_seg = (fim - inicio)/n
18 x = inicio
19 # Posiciona-se
20 tartaruga.up()
21 tartaruga.goto(x, funcao(x))
22 tartaruga.down()
23 # Desenha
24 for conta in range(n):
25 x = x + tam_seg
26 tartaruga.goto(x, funcao(x))
27
```

```

28 if '__name__' == '__main__':
29 turtle.setworldcoordinates(-math.pi, -2, math.pi,2)
30 linha(-math.pi,0,math.pi,0)
31 linha(0,-2,0,2)
32 toto = turtle.Turtle()
33 toto.pen(pensize=3, pencolor='gray')
34 grafico(toto,math.sin, -math.pi,math.pi,50)
35 toto.hideturtle()
36 turtle.exitonclick()

```

Listagem 2.3: A função seno

Podemos adicionar outros elementos ao gráfico, como seja os pontos calculados e o nome. Podemos ainda, desenhar mais do que um gráfico. O programa final, completo, é o da listagem 2.4.

```

1 import turtle
2 import math
3
4 def linha(x1,y1,x2,y2):
5 """ Traça uma linha entre dois pontos."""
6 turtle.up()
7 turtle.goto(x1,y1)
8 turtle.pd()
9 turtle.goto(x2,y2)
10 turtle.up()
11 turtle.hideturtle()
12
13
14 def grafico(tartaruga,funcao, inicio,fim, n):
15 """ Desenha o gráfico da função f entre inf e sup usando n
16 segmentos."""
17 # Tamanho dos segmentos
18 tam_seg = (fim - inicio)/n
19 x = inicio
20 # Posiciona-se
21 tartaruga.up()
22 tartaruga.goto(x, funcao(x))
23 tartaruga.dot(6)
24 tartaruga.down()
25 # Desenha
26 for conta in range(n):
27 x = x + tam_seg

```

```

27 tartaruga.goto(x, funcao(x))
28 tartaruga.dot(6)
29
30 if __name__ == '__main__':
31 # Inicializa
32 turtle.setworldcoordinates(-math.pi, -2, math.pi,2)
33 linha(-math.pi,0,math.pi,0)
34 linha(0,-2,0,2)
35 # Seno
36 toto = turtle.Turtle()
37 toto.pen(pensize=3, pencolor='gray')
38 grafico_1(toto,math.sin, -math.pi,math.pi,50)
39 toto.up()
40 toto.goto(0.5,1)
41 toto.write('SENO(X)', font=('ARIAL', 14, 'bold'))
42 toto.hideturtle()
43 # Coseno
44 titi = turtle.Turtle()
45 titi.pen(pensize=3)
46 grafico_1(titi,math.cos, -math.pi,math.pi,50)
47 titi.up()
48 titi.goto(-0.7,1)
49 titi.write('COSENO(X)', font=('ARIAL', 14, 'bold'))
50 titi.hideturtle()
51 # Termina
52 turtle.exitonclick()

```

Listagem 2.4: Seno e coseno

Ao executar o programa veremos o resultado retratado na figura 2.8.

## Sumário

Neste capítulo introduzimos o módulo `turtle` e alguns das operações que podem ser feitas sobre objectos do tipo `Turtle`. Introduzimos o conceito de construtor de um tipo e clarificámos a diferença entre o conceitos de função e de método. Exemplificámos o uso do módulo para resolver problemas de geometria e para o gráfico de funções.

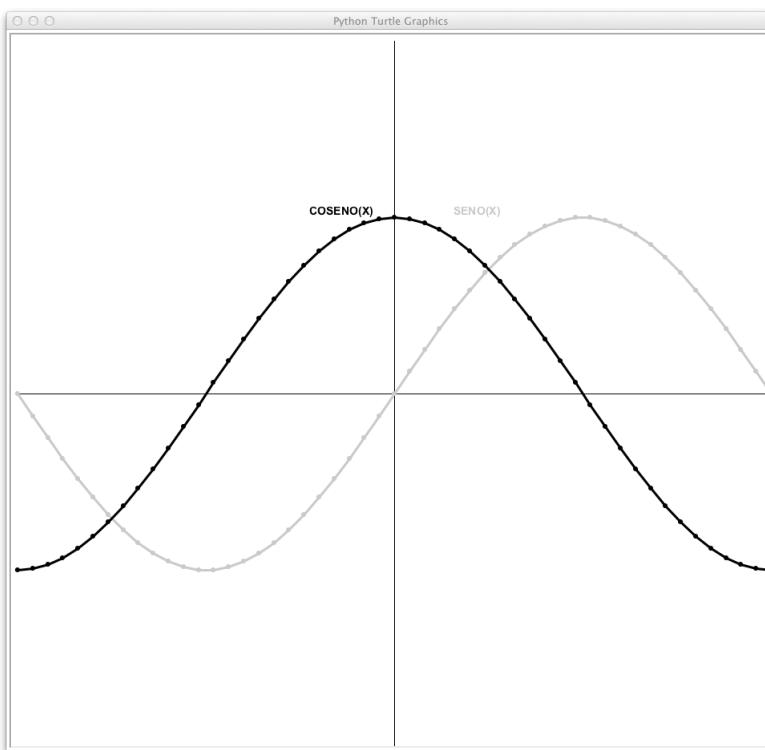


Figura 2.8: Seno e cosseno

## Teste os seus conhecimentos

Tente responder às seguintes questões que foram tratadas neste capítulo.

1. O que entende por construtor
2. Quais são as operações básicas sobre tartarugas
3. Quais as operações básicas sobre a caneta
4. O que distingue uma função de um método
5. Como podemos controlar a escala do nosso desenho

## Exercícios

**Exercício 2.1 F** Neste capítulo desenvolvemos um programa para desenhar polígonos regulares. No centro do programa está um ciclo que repete um certo número de vezes duas operações: avançar e rodar. Retome o programa e faça variar estes três elementos por forma a obter outro tipo de figuras. A figura 2.9 ilustra a situação de repetir 5 vezes o avanço de 100 unidades e uma rotação de 144 graus.



Figura 2.9: Uma estrela

**Exercício 2.2 F** Faça um programa que lhe permita simular um passeio aleatório (*random walk*). A figura 2.10 ilustra o que se pretende. Procure alterar a cor de cada segmento de modo aleatório.

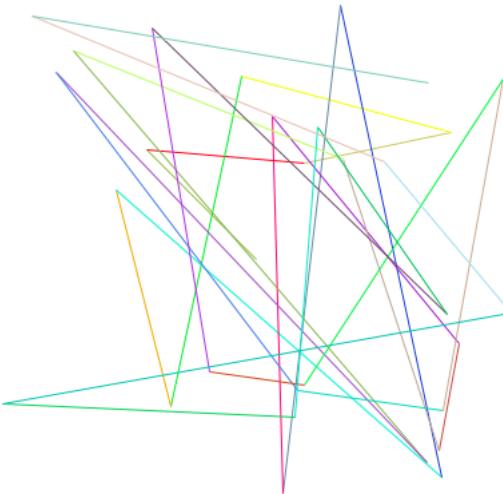


Figura 2.10: Sem rei nem roque

**Exercício 2.3 F** No texto mostrámos como se podia desenhar uma circunferência usando a função `poligono_regular`. Mas o desenho é sempre o mesmo, isto é, não podemos controlar o **raio** da circunferência. Diga como pode usar na mesma a função `poligono_regular` mas tendo em linha de conta o valor do raio.

**Exercício 2.4 M** O módulo `turtle` tem um método `circle` pré-definido que lhe permite desenhar circunferências. O método tem um parâmetro obrigatório que é o raio da circunferência. Adicionalmente podemos indicar qual a extensão que vamos desenhar, tudo (por defeito) ou apenas um arco. como a circunferência é aproximada através a um polígono regular inscrito, existe um terceiro parâmetro que torna possível o uso de `circle` para desenhar polígonos regulares. Explore essa facilidade para desenhar diversos polígonos regulares.

**Exercício 2.5 D** Desenvolva um programa que lhe permita desenhar quadrados concêntricos como, por exemplo, indicado na figura 2.11.

**Exercício 2.6 D** Desenvolva um programa que lhe permita desenhar a figura 2.12. Se reparar bem a figura é formada por quadrados cujos lados têm

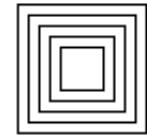


Figura 2.11: Tanto quadrado...

dimensão diferentes e os ângulos iniciais têm orientações diferentes.

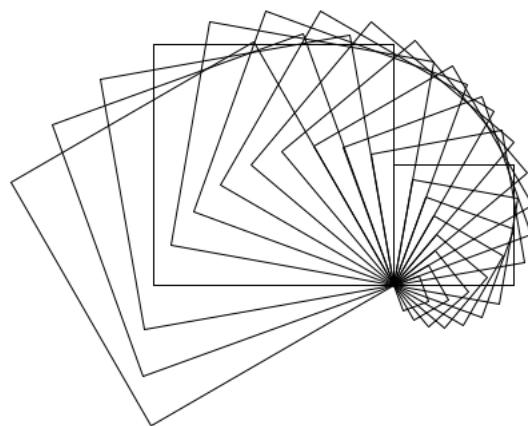


Figura 2.12: Parece um náutilus...

# Capítulo 3

## Objectos (I)

### Objectivos

- ✓ Entender os conceitos de literal, objecto e tipo
- ✓ Introduzir os conceitos de precedência e de precisão
- ✓ Explorar os tipos básicos e operações associadas: números e sequências

### 3.1 Generalidades

Os computadores utilizam e manipulam coisas que designamos por **objectos**. Numa primeira aproximação diremos que os objectos são entidades que têm:

- identidade
- valor
- tipo

A **identidade** é o que torna o objecto único podendo ser consultada mas não modificada. Em termos informáticos falamos normalmente da identidade como uma referência ou apontador para uma zona da memória onde se encontra a descrição do objecto. O **valor** traduz o estado do objecto num dado momento. Este valor pode ser acedido e em certas circunstâncias alterado. O **tipo** determina o conjunto de valores que o objecto pode assumir e as operações que com ele podemos fazer. Em **Python** o tipo é uma propriedade do objecto pode ser consultado mas não alterado.

### Tipagem

As linguagens têm restrições diferentes relativamente ao tipo dos objectos. Umas, como ADA, C++ ou JAVA, dizem-se **fortemente tipadas** e a associação entre os objectos e o seu tipo não pode ser alterada. Questão diferente é o problema da tipagem ser estática ou dinâmica. Neste último caso o que está em jogo não é um problema de **consistência** entre o objecto e o seu tipo mas antes o **tempo** em que se faz a associação entre o objecto e o tipo. Quando a este último aspecto as linguagens podem adoptar uma ligação **estática** (a ligação fica definida antes da compilação) ou uma ligação **dinâmica** (a ligação é conhecida em tempo de execução). São possíveis diversas situações. Por exemplo, existem linguagens fortemente tipadas com ligação estática (e.g., ADA), fortemente tipadas e mas suportando ligação dinâmica (e.g., C++, Java) ou ainda não tipadas e ligação dinâmica (e.g., Python ).

Em Python os tipos são implementados como **classes**. Antes de discutir de modo aprofundado o conceito de classe, podemos dizer, de forma simples que uma classe é um modelo para descrever o que o conjunto dos objectos da classe partilham, em particular os seus atributos, que definem o **estado** do objecto e as operações que podem ser feitas com o objecto, definidoras do seu **comportamento**.

Vejamos uma sessão simples no interpretador como podemos **inspecionar** objectos em Python .

```

1 >>> id(5)
2 4563119552
3 >>> 5
4 5
5 >>> type(5)
6 <class 'int'>
7 >>>
```

Ficamos a saber que 5 é um objecto guardado na posição de memória 4563119552, tem valor 5 (!) e tipo<sup>1</sup> inteiro (**int**). Já sabemos que podemos associar nomes aos objectos. Por isso não nos estranhará o resultado da sessão seguinte, efectuada na sequência da sessão anterior.

```

1 >>> a = 5
2 >>> id(a)
3 4563119552
```

<sup>1</sup>Não se esqueça que os tipos são implementados como classes.

```

4 >>> a
5
6 >>> type(a)
7 <class 'int'>
8 >>>

```

Como se pode ver o nome *a* está associado ao objecto 5, pelo que se obtêm exactamente os mesmos resultados. Podemos visualizar (ver figura 3.1) a situação da memória que resulta da sessão anterior.

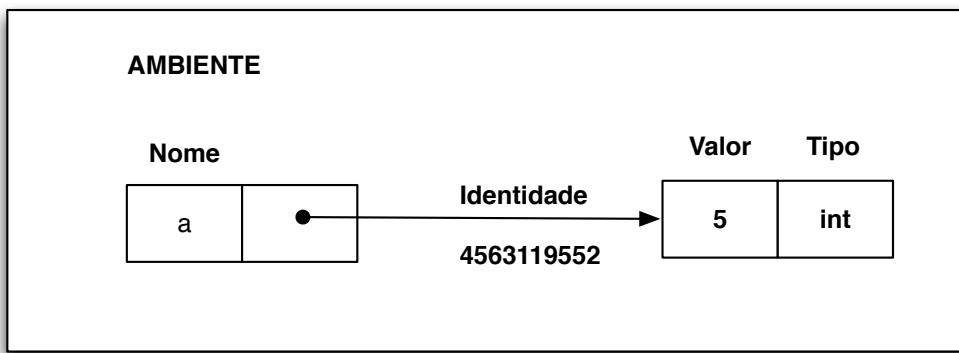


Figura 3.1: Ambiente, nomes e objectos

Independentemente do aprofundamento que faremos mais adiante desta questão, importa salientar desde já alguns aspectos. Em primeiro lugar, sempre que se executa algo existe um **ambiente** que permite identificar os objectos activos e os seus atributos. Em segundo lugar, os nomes dentro do ambiente activo formam o que se designa por **espaço de nomes**. Em terceiro lugar, o nome permite para aceder aos restantes atributos de um objecto.

Quando iniciamos o sistema existe um ambiente inicial simples, que pode ser inspeccionado. Qualquer alteração posterior modifica o ambiente.

```

1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
 2336.1.00)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
 information.
4 >>> dir()
5 ['__builtins__', '__doc__', '__name__', '__package__']

```

```

6 >>> a = 5
7 >>> dir()
8 ['__builtins__', '__doc__', '__name__', '__package__', 'a']
9 >>> dir(a)
10 ['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__
 class__', '__delattr__', '__divmod__', '__doc__', '__eq__',
 '__float__', '__floor__', '__floordiv__', '__format__',
 '__ge__', '__getattribute__', '__getnewargs__', '__gt__',
 '__hash__', '__index__', '__init__', '__int__', '__invert__',
 '__le__', '__lshift__', '__lt__', '__mod__', '__mul__',
 '__ne__', '__neg__', '__new__', '__or__', '__pos__',
 '__pow__', '__radd__', '__rand__', '__rdivmod__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
 '__rlshift__', '__rmod__', '__rmul__', '__ror__',
 '__round__', '__rpow__', '__rrshift__', '__rshift__',
 '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
 '__sizeof__', '__str__', '__sub__', '__subclasshook__',
 '__truediv__', '__trunc__', '__xor__', 'bit_length',
 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator',
 'real', 'to_bytes']
11 >>>

```

Esta listagem mostra que inicialmente existe um conjunto reduzido de (quatro) nomes conhecidos. Quando criamos um objecto e o associamos a um nome (*a*), se voltarmos a **inspeccionar** o ambiente lá encontraremos o novo nome. Igualmente, o uso do comando **dir** sobre um objecto permite saber que operações podemos efectuar com esse objecto. Na realidade, o sistema mostra todas as operações que podemos fazer com qualquer objecto do mesmo tipo.

`__builtins__` é o nome de um módulo que contém as constantes e funções embutidas. Algumas dessas constantes são `True`, `False` e `None`. As duas primeiras correspondem aos membros do tipo `boolean` e a última ao tipo `NoneType`. `None` denota a ausência de valor. `__name__` permite conhecer o valor associado ao ambiente activo.

```

1 >>> dir(__builtins__)
2 ['ArithError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'None',
 'NotImplemented', 'NotImplementedError', 'NotADirectoryError',
 'OverflowError', 'PendingDeprecationWarning', 'RuntimeError',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'TypeError',
 'ValueError', 'Warning']

```

```

IOError', 'ImportError', 'ImportWarning', 'IndentationError',
', 'IndexError', 'KeyError', 'KeyboardInterrupt', '
LookupError', 'MemoryError', 'NameError', 'None', ,
NotImplemented', 'NotImplementedError', 'OSerror', ,
OverflowError', 'PendingDeprecationWarning', '
ReferenceError', 'ResourceWarning', 'RuntimeError', ,
RuntimeWarning', 'StopIteration', 'SyntaxError', ,
SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', ,
True', 'TypeError', 'UnboundLocalError', ,
UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', ,
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', ,
ValueError', 'Warning', 'ZeroDivisionError', '_', ,
__build_class__', '__debug__', '__doc__', '__import__', ,
__name__', '__package__', 'abs', 'all', 'any', 'ascii', ,
bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', ,
classmethod', 'compile', 'complex', 'copyright', 'credits', ,
'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', ,
exec', 'exit', 'filter', 'float', 'format', 'frozenset', ,
getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id',
', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len',
', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', ,
print', 'property', 'quit', 'range', 'repr', 'reversed', ,
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
3 >>> __name__
4 '__main__'
5 >>>

```

Podemos ainda pedir ajuda para saber o que determinados objectos fazem, recorrendo ao comando `help`.

```

1 help(pow)
2 Help on built-in function pow in module builtins:
3
4 pow(...)
5 pow(x, y[, z]) -> number
6
7 With two arguments, equivalent to x**y. With three
8 arguments,
equivalent to (x**y) % z, but may be more efficient (e.g.
 for longs).

```

Como se pode observar a função `pow` pode ter dois ou três argumentos. No primeiro caso, devolve a potência do primeiro elevada ao segundo. No segundo caso, obtemos esse valor módulo o terceiro argumento.

São vários os tipos pré-definidos em Python , que podem ser agrupados por características comuns. Os principais são os tipos numéricos, as sequências, e os mapeamentos<sup>2</sup>. Neste capítulo iremos introduzir com algum detalhe os tipos mais primitivos de entre os numéricos (inteiros, reais e complexos) e sequências (cadeias de caracteres, tuplos e *range*) (ver figura 3.2)<sup>3</sup>.

## 3.2 Números

Os objectos primitivos mais simples são os números. Existem fundamentalmente três tipos de números: inteiros, reais (ou vírgula flutuante)<sup>4</sup> e complexos. Como todos os objectos os números são construídos por recurso a literais, i.e., expressões cuja sintaxe gera (ou denota) um objecto. Números sem qualquer adorno são inteiros, com um ponto decimal são reais e com um 'j' ou 'J' são complexos. Na tabela 3.1 encontram-se literais para diferentes objectos numéricos<sup>5</sup>.

| Literal                         | Interpretação     |
|---------------------------------|-------------------|
| 733, -15, 0                     | Inteiros          |
| 2.46, 3.14e-20, 6E100, 4.0e+120 | Vírgula flutuante |
| 4 + 5j, 4.0 + 5.0j, 6J          | Números Complexos |

Tabela 3.1: Literais Numéricos

Em relação aos literais podemos desde já observar que existem várias notações sintáticas (i.e., literais) para os números em vírgula flutuante. Por outro lado, os números inteiros podem ser representados com precisão ilimitada, o mesmo não sendo verdade para reais e complexos. Devemos ter em atenção de que uma coisa são os objectos e outra a sua representação externa. Olhemos para a sessão que se segue.

<sup>2</sup>Por exemplo, também existe o tipo `module`, restrito basicamente à operação de acesso (aos objectos do módulo).

<sup>3</sup>No texto será explicado o porquê da inclusão do tipo `bool` no subgrupo dos tipos numéricicos.

<sup>4</sup>Do inglês *float*.

<sup>5</sup>Os inteiros também podem ser representados em notação binária, octal e hexadecimal.

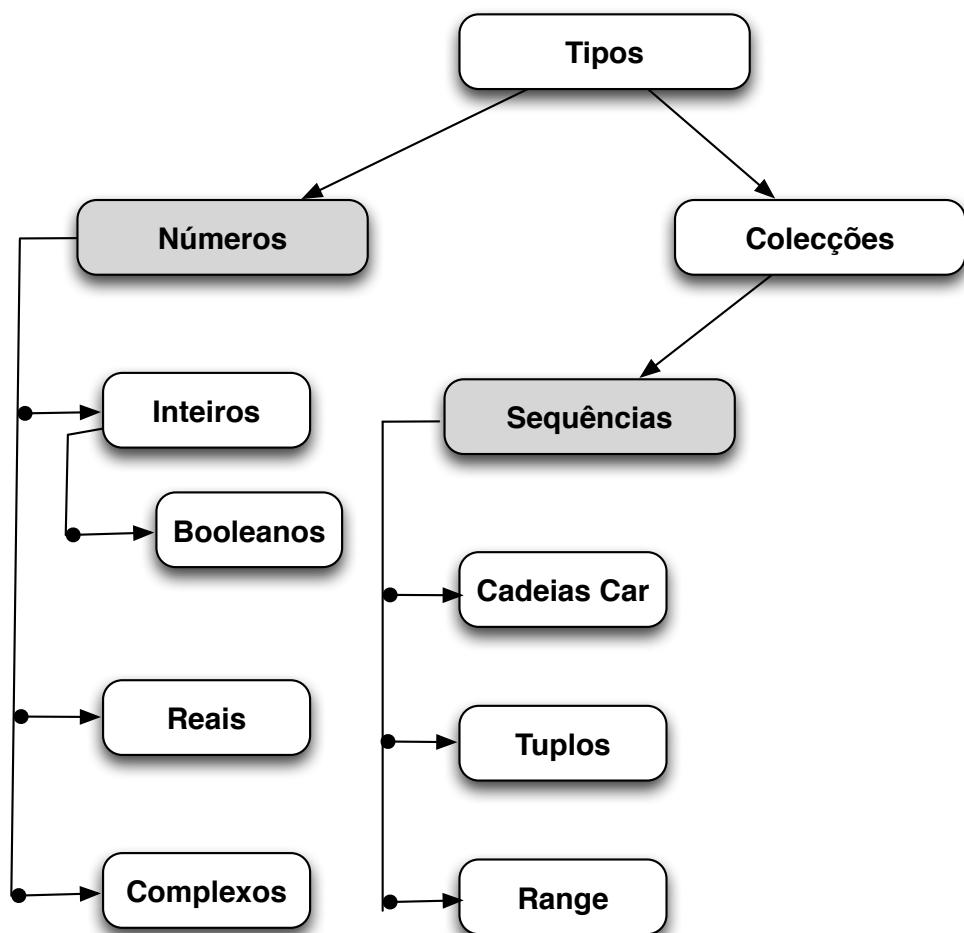


Figura 3.2: Tipos básicos

```

1 >>> 543
2 543
3 >>> 4.0e3
4 4000.0
5 >>> 4E100
6 4.000000000000001e+100
7 >>> 5.0e+50
8 5e+50
9 >>> 4.0e50
10 4.000000000000003e+50
11 >>> 2.4 + 5.2J
12 (2.4+5.2j)
13 >>> 0.0 + 5j
14 5j
15 >>>

```

Como vemos os dados introduzidos são "echoados" de modo não necessariamente idêntico. Como anteriormente explicado, quando se recorre a um interpretador este funciona com base num ciclo normalmente designado por **ciclo lê-avalia-escreve**: uma expressão é lida, o valor que lhe está associado é calculado e o resultado desse cálculo é enviado para o exterior. Na aparência é um ciclo com três passos. No entanto são na realidade **cinco** pois a seguir à leitura há uma conversão para um formato interno e antes da impressão há a escolha do formato de saída. Um outro aspecto que devemos notar é que no caso de algumas expressões, por exemplo da **4E100**, o resultado enviado é diferente do esperado: existe uma imprecisão na máquina!

## Operações

Se não pudéssemos efectuar operações e cálculos com os números estes não teriam muito interesse. Desde tempos remotos que com eles fazemos operações básicas que nos permitem contar, medir ou calcular. Na escola fomos progredindo dos números inteiros até aos complexos passando pelos racionais e pelos reais. Na tabela 6.2 encontram-se as operações elementares que podemos efectuar, e que são comuns aos três tipos de números, excepto nalguns casos óbvios envolvendo números complexos.

Vejamos uma pequena sessão de cálculos (ver listagem 3.1).

```

1 >>> 4 + 7
2 11

```

Tabela 3.2: Operações básicas sobre números

| Operações        | Descrição                   |
|------------------|-----------------------------|
| +                | Adição                      |
| -                | Subtração (dois argumentos) |
| *                | Multiplicação               |
| /                | Divisão                     |
| //               | Divisão truncada            |
| %                | resto                       |
| -                | Negação (um argumento)      |
| <b>abs</b>       | Valor absoluto              |
| <b>conjugate</b> | o conjugado de um complexo  |
| <b>divmod</b>    | Divisão truncada e resto    |
| <b>pow</b>       | Exponenciação               |
| <b>**</b>        | Exponenciação               |

```

3 >>> 4 - 7
4 -3
5 >>> 4.3 + 2.4 # oops, mais um problema de precisão!
6 6.699999999999999
7 >>> (4.3 + 2.4j) + (3.4 + 4.2j)
8 (7.69999999999999+6.6j)
9 >>> 34 * 14
10 476
11 >>> 1 / 2
12 0.5
13 >>> 1 / 3
14 0.3333333333333333
15 >>> 1 / 3.14
16 0.3184713375796178
17 >>> import math
18 >>> 1 / math.pi
19 0.3183098861837907
20 >>> 1 // 2
21 0
22 >>> -1 // 2
23 -1
24 >>> 1 // -2

```

```

25 -1
26 >>> 4.3 // 2.4
27 1.0
28 >>> 4.3 / 2.4
29 1.7916666666666667
30 >>> 4 % 5
31 4
32 >>> 7 % 5
33 2
34 >>> -7 % 5
35 3
36 >>> 7.0 % 5.0
37 2.0
38 >>> 4+3j * 2+5j
39 (4+11j)
40 >>> (4+3j) * (2+5j)
41 (-7+26j)
42 >>> (4+3j) / (2+5j)
43 (0.793103448275862-0.48275862068965514j)
44 >>> (4+3j).conjugate()
45 (4-3j)
46 >>> divmod(7,5)
47 (1, 2)
48 >>> pow(2,3)
49 8
50 >>> pow(2,3,2)
51 0
52 >>> 2 ** 3
53 8
54 >>> 0 ** 0 # zero levantado a zero é igual a um!
55 1
56 >>> (4+2j)**2
57 (12+16j)
58 >>> (4+3j) / (2+5j)
59 (0.793103448275862-0.48275862068965514j)
60 >>>

```

Listagem 3.1: Operações com números

Alguns aspectos sobressaem desta pequena sessão. Desde logo, o facto de o mesmo **símbolo** de operador ser usado para efectuar operações com números de tipos diferentes. Dizemos que o operador está **sobre carregado**. Operadores Sobrecarregados

Por outro lado, quando os objectos são de tipo diferente existe uma **conversão** automática de tipos. Por exemplo, quando dividimos um inteiro por um número em vírgula flutuante o resultado é um número em vírgula flutuante. A conversão é feita para o tido dito mais *geral*: complexos são mais *gerais* do que os reais, e estes mais gerais do que os inteiros. A figura 3.3 mostra o modo de proceder de **Python**. Também se torna de novo aparente as questões de precisão já antes referidas.

[Conversão de Tipos](#)

### Precisão

Os computadores são máquinas com limitações. Cada número tem que ter uma representação única no computador. No caso dos números inteiros em **Python** podemos representar qualquer número, estando apenas limitados pela memória do computador. Já no caso dos números reais o problema é mais delicado. Como existem uma infinidade de números reais, nunca poderemos representar todos, mas apenas um seu subconjunto<sup>1</sup>. Por outro lado, uma vez mais devido a limitações da máquina, usam-se representações de comprimento fixo. Isso implica, por exemplo, que um número com muitos dígitos tenha que ser aproximado. Daqui decorre que os números reais têm representações imprecisas e as operações que com eles fazemos são inexatas. Para saber alguma informação sobre a precisão o leitor interessado pode fazer o indicado na listagem. Por **precisão**, de um sistema em vírgula flutuante, entende-se o número *epsilon* ( $\epsilon$ ) mais pequeno para o qual se verifica  $1 + \epsilon > 1$ .

```

1>>> import sys
2>>> sys.float_info
3sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
 max_10_exp=308, min=2.2250738585072014e-308, min_exp
 ==-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon
 =2.220446049250313e-16, radix=2, rounds=1)
4>>>
```

Da listagem<sup>a</sup> podemos retirar, entre outras coisas, os valores máximo e mínimo que se pode representar bem como o valor de *epsilon*

---

<sup>a</sup>Os valores listados dependem da máquina em que está a correr o interpretador.

Na tabela 6.2 afirmamos que // representa a divisão truncada. Na realidade ela pretende designar a divisão inteira por recurso à operação **floor**. É como se a divisão com inteiros fosse executada em dois passos: primeiro, é feita uma divisão como se se tratasse de números reais; depois, o valor é aproximado ao maior inteiro **menor** do que o real obtido. É por isso que te-

mos o resultado nas linhas 20 a 27. Existe uma diferença subtil entre truncar ou fazer do modo referido. A listagem ilustra a diferença.

```

1 >>> import math
2 >>> math.floor(4.5)
3 4
4 >>> math.floor(-4.5)
5 -5
6 >>> math.trunc(4.5)
7 4
8 >>> math.trunc(-4.5)
9 -4
10 >>>

```

Como se vê só no caso de os números serem positivos é que a truncagemv e a operação `floor` coincidem.

### Coerção e construtores

O programador pode forçar a conversão de tipos numéricos. A tabela 3.3 mostra os operadores que podemos usar.

Tabela 3.3: Operadores de conversão de tipos

| Operadores                           | Descrição                                 |
|--------------------------------------|-------------------------------------------|
| <code>int(obj, base=10)</code>       | Converte para inteiro na base <i>base</i> |
| <code>float(obj)</code>              | Converte para float                       |
| <code>complex(real, imag=0.0)</code> | Converte para complexo                    |

Vejamos agora alguns exemplos ilustrativos.

```

1 >>> int(4.5)
2 4
3 >>> int(8/3)
4 2
5 >>> int('123')
6 123
7 >>> float('123')
8 123.0
9 >>> float('123e10')

```

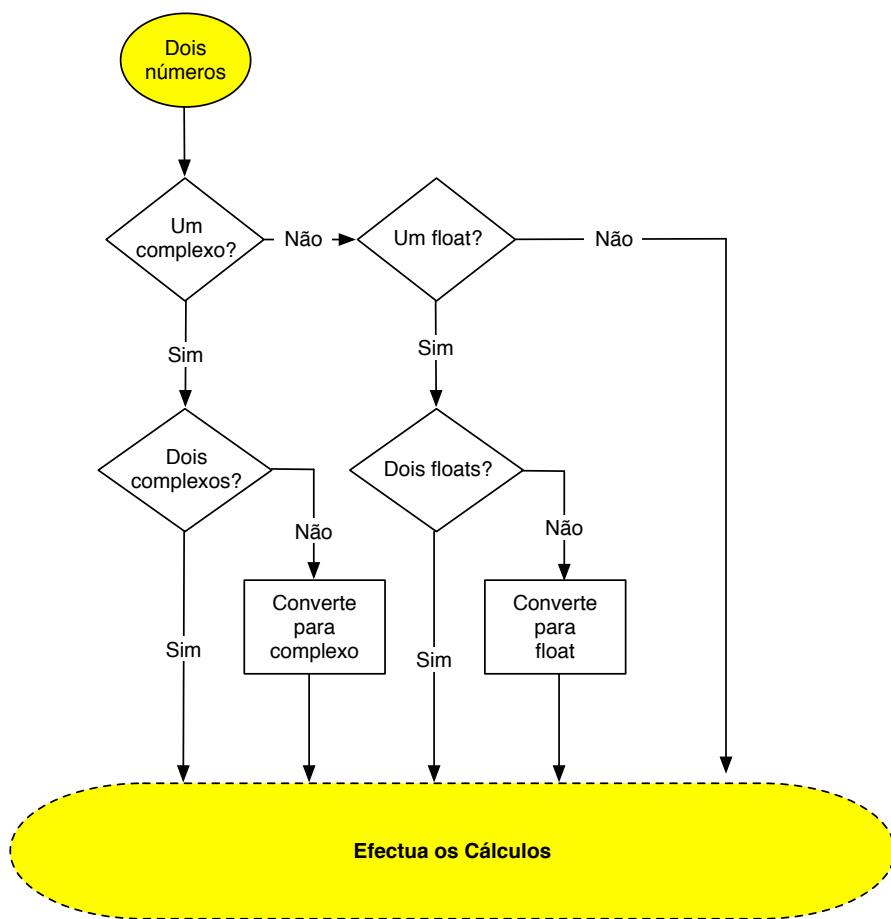


Figura 3.3: Conversão automática de tipos

```

10 1230000000000.0
11 >>>
12 >>> float(34)
13 34.0
14 >>> float('123')
15 123.0
16 >>> float('123e10')
17 1230000000000.0
18 >>>
19 >>> complex(5,3)
20 (5+3j)
21 >>> complex('4+5j')
22 (4+5j)
23 >>> complex(5)
24 (5+0j)
25 >>> float(4+5j)
26 Traceback (most recent call last):
27 File "<stdin>", line 1, in <module>
28 TypeError: can't convert complex to float; use abs(z)
29 >>> abs(4+5j)
30 6.4031242374328485

```

Estes exemplos mostram que nem tudo é possível fazer. Por exemplo, não podemos converter um número complexo num real<sup>6</sup>.

### Construtores

As funções indicadas na tabela 3.3 são conhecidas como **construtores** do tipo<sup>7</sup>. Podem também ser usadas sem argumento nenhum, criando neste caso objectos particulares do tipo.

```

1 >>> int()
2 0
3 >>> float()
4 0.0
5 >>> complex()
6 0j
7 >>>

```

Os números complexos têm a particularidade de terem dois atributos: a parte real e a parte imaginária. Em certas aplicações estamos interessados

---

<sup>6</sup>Mas, como veremos mais tarde, podemos aceder às suas componentes e depois efectuar a conversão

<sup>7</sup>Não é por acaso que têm o nome do tipo.

em obter separadamente cada uma destas partes. No caso dos complexos, tal pode ser feito usando a **notação por ponto** e o nome dado a cada um dos atributos, **real** e **imag**.

```

1 >>> (4.5 + 3.2j).real
2 4.5
3 >>> (4.5 + 3.2j).imag
4 3.2
5 >>>

```

## Precedência

Os exemplos que temos vindo a mostrar são bastante simples, envolvendo apenas um operador em cada expressão. O que acontece se tivermos mais operadores, do mesmo tipo ou de tipos diferentes, ou seja, por que ordem são feitas as operações? Esta questão resolve-se devido à existência de **prioridade** ou **precedência** entre os operadores. A tabela 6.2 mostra os operadores por ordem crescente de prioridade<sup>8</sup>. Vejamos um exemplo ilustrativo.

```

1 >>> 2 + 3 * 5
2 17
3 >>> (2 + 3) * 5
4 25
5 >>> 2 ** 3 ** 2
6 512
7 >>> (2 ** 3) ** 2
8 64
9 >>> 2 * 3 / 4
10 1.5
11 >>> 2 / 3 * 4
12 2.6666666666666665
13 >>>

```

## Outros Casos

À semelhança de outras linguagens também existem operações ao nível do bit. Só funcionam com argumentos inteiros e os números negativos estão representados em complemento para dois. A tabela 3.4 mostra as operações por ordem crescente de prioridade.

Vejamos exemplos ilustrativos.

---

<sup>8</sup>Podemos forçar a ordem pela qual as operações são efectuadas recorrendo ao uso de parênteses.

Tabela 3.4: Operadores ao nível do Bit

| Operadores | Descrição                                    |
|------------|----------------------------------------------|
| $x   y$    | O <i>ou</i> dos bits                         |
| $x ^ y$    | O <i>ou exclusivo</i> dos bits               |
| $x & y$    | O <i>and</i> dos bits                        |
| $x \ll n$  | O <i>deslocamento à esquerda</i> de $n$ bits |
| $x \gg n$  | O <i>deslocamento à direita</i> de $n$ bits  |
| $\sim x$   | A <i>inversão</i> dos bits                   |

```

1 >>> 5 >> 2
2 1
3 >>> 4 << 3
4 32
5 >>> 5 & 3
6 1
7 >>> 5 ^ 2
8 7
9 >>> 5 | 3
10 7
11 >>> ~8
12 -9
13 >>> ~~7
14 6
15 >>>

```

## Exemplos

Vejamos dois exemplos simples de programas envolvendo cálculos com números.

### Exemplo 3.1

---

Suponhamos que queremos calcular o declive de uma recta dados dois pontos.

A fórmula para o fazer é conhecida:

$$\text{declive}(x_1, y_1, x_2, y_2) = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

Daqui decorre um programa muito simples.

```

1 def declive(x1,y1,x2,y2):
2 """ Usa a forma habitual para calcular o declive, dados dois
 pontos.
3 Cuidado: não podem ter a mesma abcissa!
4 """
5 return (y2 - y1)/(x2 - x1)

```

O único problema com este programa é que não prevê o caso de os pontos terem a mesma abcissa, isto é, estarmos na presença de uma recta perpendicular ao eixo dos  $x$ . Mas tal pode ser remediado facilmente efectuando um teste a essa situação e tomando a acção apropriada.

```

1 def declive(x1,y1,x2,y2):
2 """ Usa a forma habitual para calcular o declive, dados dois
 pontos.
3 Cuidado: não podem ter a mesma abcissa!
4 """
5 if x1 != x2:
6 return (y2 - y1)/(x2 - x1)
7 else:
8 return float('Inf')

```

Note-se como em Python podemos introduzir uma constante que simboliza o infinito (linha 8). Caso pretendamos menos infinito usamos `float('-Inf')`.

**3.1**

### Exemplo 3.2

Passemos ao caso do cálculo das raízes de um polinómio do segundo grau.

Também aqui a fórmula é conhecida:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

O código é directo.

```

1 import cmath
2
3 def poli_2(a,b,c):
4 """ Calcula as raízes de um polinomio do segundo grau."""
5 delta = cmath.sqrt(b**2 - 4 * a * c)

```

```

6 raiz_1 = (- b + delta)/ (2 * a)
7 raiz_2 = (- b - delta) / (2 * a)
8 return raiz_1, raiz_2

```

Neste exemplo fomos obrigados a usar o módulo `cmath`, semelhante ao módulo `math`, mas que disponibiliza operações para números complexos. Notar ainda que é possível uma função devolver mais do que um valor, neste caso as duas raízes do polinómio<sup>9</sup>.

3.2

---

### 3.3 Booleanos

Os booleanos foram representados durante muito tempo por números, mais concretamente, o 1 para verdadeiro e o 0 para falso. Actualmente têm o seu próprio tipo, `bool` e valores (as constantes `True` e `False`). Existem três operações com objectos do tipo booleano, como se indica na tabela 3.5.

Tabela 3.5: Operadores Booleanos

| Operadores           | Descrição          |
|----------------------|--------------------|
| <code>x and y</code> | O <i>e</i> lógico  |
| <code>x or y</code>  | O <i>ou</i> lógico |
| <code>not x</code>   | A negação lógica   |

Em termos de implementação convém referir que estas operações e seguem o princípio dito de *curto circuito*: no caso do `and` a operação termina mal um dos operandos tenha valor `False` e, no caso do `or`, termina mal um dos operandos seja `True`. A existência de booleanos é fundamental em programação pois é o que necessitamos usar nas instruções de controlo condicionais ou nos ciclos de execução condicional.

Eis alguns exemplos simples de utilização.

```

1 >>> True and False
2 False
3 >>> True and True
4 True

```

---

<sup>9</sup>Na realidade o que é devolvido é um **tuplo**, objecto de um tipo de que falaremos na secção 3.6.

```

5 >>> True or False
6 True
7 >>> False or False
8 False
9 >>> not True
10 False

```



### Booleanos e inteiros

Tecnicamente o tipo classe `bool` é uma subclasse da classe `int`. `True` e `False` comportam-se exactamente como os inteiros 1 e 0, respectivamente. Daí ser possível observar coisas bizarras como se indica na listagem.

```

1 >>> True + 2
2 3
3 >>> False * 3
4 0
5 >>> True + False
6 1
7 >>>

```

Encontrar situações destas em código é sinal de má programação.

Existe um conjunto grande de operadores de comparação cujo resultado é um objecto do tipo booleano. A tabela 3.6 identifica-os.

Tabela 3.6: Operadores de Comparação

| Operadores | Descrição           |
|------------|---------------------|
| >          | Maior               |
| >=         | Maior ou igual      |
| <          | Menor               |
| >=         | Menor ou igual      |
| ==         | Igual valor         |
| !=         | Desigual (valor)    |
| is         | Igual identidade    |
| is not     | Desigual identidade |

Vejamos alguns exemplos de utilização.

```

1 >>> 4 >= 5
2 False

```

```

3 >>> 3 <= 4
4 True
5 >>> 4 != 5
6 True
7 >>> 3 == 3
8 True
9 >>> 4 == 4.0
10 True
11 >>> 3 is 3
12 True
13 >>> 4 is 4.0
14 False
15 >>> 5 == 5E0
16 True
17 >>>

```

Notar a diferença entre as operações `==` e `is`. A primeira verifica se os objectos têm o mesmo valor, enquanto a segunda só é `True` se se tratar do mesmo objecto, isto é, tiverem a mesma identidade.

O construtor deste tipo chama-se ...`bool`.

```

1 >>> bool()
2 False
3 >>> bool(0)
4 False
5 >>> bool(0.0)
6 False
7 >>> bool(0j)
8 False
9 >>> bool(45)
10 True
11 >>> bool(43.5)
12 True
13 >>> bool((3+4j))
14 True
15 >>>

```

Sem argumento, cria o objecto `False`. Com argumentos específicos (linhas 3 a 8) cria ainda o objecto `False`<sup>10</sup>. Com argumentos genéricos (linhas

---

<sup>10</sup>Como veremos mais tarde há outros objectos que são reconhecidos como `False`. Tipicamente trata-se dos objectos de outros tipos criados pelo respectivo construtor.

9 a 14) cria o objecto `True`.

## 3.4 Cadeia de Caracteres

Cada vez mais nesta sociedade nós trocamos mensagens uns com os outros, por exemplo sob a forma de correio electrónico, sms, no twitter ou no facebook, guardamos informação em grandes bases de dados, procuramos e manipulamos informação, por exemplo encriptando-a. O que têm em comum estas diferentes maneiras de interagir com, ou por meio de, informação é o facto de esta ser representada por texto, ou seja, por uma sequência de caracteres de um dado alfabeto. A própria vida pode ser entendida a partir de uma (grande) cadeia de caracteres, a cadeia de ADN, formada a partir de apenas quatro letras. Essas quatro letras correspondem às quatro bases distintas existentes no nosso ADN, identificadas por letras: T(imina), A(denina), C(itosina) e G(uanina).

Em **Python** é fácil representar uma cadeia de ADN usando uma **cadeia de caracteres** que representa a sequência de bases.

```

1 >>> meu_adn = 'ATTCGGTATGGTAC'
2 >>> meu_adn
3 'ATTCGGTATGGTAC'
```

As cadeias de caracteres são outro tipo primitivo da linguagem **Python**. As cadeias de caracteres são colecções ordenadas, isto é sequências, de caracteres<sup>11</sup>, homogéneas (todos os seus elementos são do mesmo tipo, no caso caracteres). Como no caso dos números estas cadeias são construídas por recurso a literais como se ilustra na tabela 3.7.

Dos exemplos da tabela resulta claro que a **marca sintáctica** para construir cadeias de caracteres é o recurso às plicas. Estas podem ser simples, duplas ou triplas. As últimas permitem escrever longas cadeias de caracteres que se propagam por mais do que uma linha sendo por isso muito usadas para comentar o nosso código. Existem cadeias *brutas* que são prefixadas com `r`. As cadeias são sensíveis ao caso, pelo que "`alma`" e "`Alma`" são objectos diferentes. Na listagem 3.2 mostramos um exemplo trivial. A existência de mais do que uma marca é útil para as situações das cadeias de caracteres em que existem no seu interior plicas, como é visível em alguns exemplos da tabela 3.7. As cadeias de caracteres podem ainda ter associadas marcas, que alteram o modo como estas são interpretadas, como se apresenta no último

---

<sup>11</sup>Em **Python** ao contrário de outras linguagens não existe o tipo caracter.

Tabela 3.7: Literais Para Cadeias de Caracteres

| Literal                              | Interpretação       |
|--------------------------------------|---------------------|
| 'alma'                               | Só um plica         |
| "lamA"                               | Duas plicas         |
| """" Mala""""                        | Três plicas duplas  |
| '''MaLa'''                           | Três plicas simples |
| '''Sporting's''                      | Mistura             |
| ' Ele gritou "Golo"!'                | Mais mistura        |
| '''Sim!'' disse ele. 'É a praxe...'' | Ainda mais mistura  |
| <u>r'\t\nesto\n costa'</u>           | Cadeia bruta        |

exemplo.

### Caracteres de controlo<sup>12</sup>

Como já aconteceu com os números, as cadeias introduzidas não são sempre exactamente ecoadas. No exemplo da listagem 3.2 exemplificamos essa situação. Em particular, no exemplo com três plicas a cadeia "**gosto muito de jogar futebol**" é transformada em "**gosto muito de jogar\nfutebol**", porque o utilizador carregou na tecla de **return**. O uso de **\n** tem a função de um caracter de controlo, que ao ser interpretado faz mudar de linha para continuar a visualização.

```

1 >>> 'bola'
2 'bola'
3 >>> "Bola"
4 'Bola'
5 >>> """BoLa"""
6 'BoLa'
7 >>> """gosto muito de jogar
8 ... futebol"""
9 'gosto muito de jogar\nfutebol'
10 >>>

```

Listagem 3.2: Uso das plicas

<sup>12</sup>Do inglês *Escape characters*.

O uso destes caracteres permite formatar, de modo simples, os nossos textos. Existe uma multiplicidade de caracteres de controlo parcialmente ilustrada na tabela ??.

Tabela 3.8: Caracteres de controlo em cadeias de carateres

| Escape | Interpretação                |
|--------|------------------------------|
| \\\    | Armazena uma barra inclinada |
| \b     | Espaçamento atrás            |
| \n     | Muda de linha                |
| \t     | Tabulação horizontal         |
| \v     | Tabulação vertical           |

A listagem 3.3 ilustra o uso dos caracteres de controlo.

```

1 >>> "spam"
2 'spam'
3 >>> "s\tpa\nm"
4 's\tpa\nm'
5 >>> r"s\tpa\nm"
6 's\\tpa\\\\nm'
7 >>> "C:\\spam\\\\toto.exe"
8 'C:\\spam\\\\toto.exe'
9 >>> "\tsp\\vam"
10 '\tsp\x0bam'
11 >>>

```

Listagem 3.3: Marcas em cadeias de caracteres

## Operações de conversão

Cada caractere, que aparece numa cadeia, é representado por meio de um código<sup>13</sup>. Existe uma operação que nos permite dado um caractere obter o seu código (**ord**), e outra para dado um código obter o respectivo caractere (**chr**).

[Codificação](#)

---

<sup>13</sup>Existem vários códigos, como o ASCII, Latin-1, Unicode. Estes códigos coincidem nos primeiros 127 caracteres (basicamente dígitos, letras e alguns sinais). Os códigos que aparecem após o código ASCII, apareceram para dar expressão a caracteres especiais de várias línguas, como o português. Actualmente o código que melhor garante a portabilidade é o Unicode.

```

1 >>> car_1 = 'A'
2 >>> ord(car_1)
3 65
4 >>> chr(65)
5 'A'
6 >>> len(car_1)
7 1
8 >>> car_3 = 'ó'
9 >>> car_3
10 'ó'
11 >>> ord(car_3)
12 243
13 >>> chr(243)
14 'ó'
15 >>> len(car_3)
16 1
17 >>>

```

### Comparando cadeias de caracteres

As cadeias de caracteres podem ser comparadas usando os operadores convencionais de comparação: `<`, `<=`, `==`, `!=`, `>`, `>=`. Para se obter o resultado destas comparações usam-se os códigos dos seus caracteres. Quando existe apenas um caractere o método de comparação é trivial. Quando existe mais do que um caractere é preciso percorrer as duas cadeias a partir da posição zero. Se os caracteres numa dada posição são iguais, passamos para a posição seguinte e repetimos. Se são diferentes, o resultado final é o que decorre da comparação desses caracteres. Se as duas cadeias têm comprimento diferente mas são iguais até à última posição da menor então a de maior comprimento é sempre maior.

```

1 >>> 'a' == 'a'
2 True
3 >>> 'a' > 'A'
4 True
5 >>> 'amo' < 'ama'
6 False
7 >>> 'pa' < 'para'
8 True
9 >>> 'a' != 'A'
10 True

```

11 &gt;&gt;&gt;

## Operadores

Existe um conjunto de operações comuns os vários tipos de sequências, logo que podem ser usadas com cadeias de caracteres. A tabela ?? apresenta as básicas.

Tabela 3.9: Operações básicas Para Cadeias de Caracteres

| Literal          | Interpretação                        |
|------------------|--------------------------------------|
| +                | Concatenação de cadeias de carateres |
| *                | Cópias de superfície de uma cadeia   |
| <code>len</code> | Comprimento da cadeia                |

O leitor atento terá logo notado que os **símbolos** para as operações de concatenação e de repetição são os mesmos que os utilizados para as operações de soma e produto de números, respectivamente. Uma vez mais dizemos que os operadores estão **sobrecurregados**. Sendo o símbolo (ou nome) da operação o mesmo o que faz a diferença é o **tipo** do objecto.

Alguns exemplos simples estão ilustrados na listagem 3.4.

```
1 >>> 'GAATTC' + 'GGATCC'
2 'GAATTCTGGATCC'
3 >>> 'GAATTC' * 2
4 'GAATTCGAATTC'
5 >>> 2 * 'GAATTC'
6 'GAATTCGAATTC'
7 >>> len('GAATTC')
8 6
```

Listagem 3.4: Caracteres: operações básicas

## Indexação

As cadeias de caracteres são **sequências**, logo são ordenadas. Cada posição tem associada um **índice**. Os índices crescem da esquerda para a direita a partir de 0 até `len(<cadeia>) - 1`, e decrescem de `-1` até `-len(<cadeia>)` (ver figura 3.4).

Os índices permitem aceder a um caractere particular, usando para tal o operador de indexação `[ ]`.

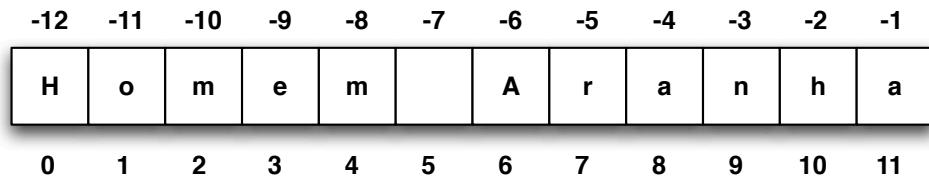


Figura 3.4: Cadeia de caracteres: indexação

```
1 >>> cadeia = 'Homem Aranha'
2 >>> cadeia[3]
3 'e'
4 >>> cadeia[-1]
5 'a'
6 >>> cadeia[0]
7 'H'
8 >>> cadeia[4] == cadeia[-8]
9 True
10 >>>
```

## Fatiamento

Podemos generalizar a operação de indexação por forma a obter uma fatia (elementos contíguos) da cadeia ou para obter a sequência de elementos espaçados regularmente. Usamos agora a notação `[inf:sup]`, no primeiro caso, e `[inf:sup:step]`, no segundo caso.

```
1 >>> cadeia = 'Homem Aranha'
2 >>> cadeia[1:4]
3 'ome'
4 >>> cadeia[-6:-2]
5 'Aran'
6 >>> cadeia[:5]
7 'Homem'
8 >>> cadeia[6:]
9 'Aranha'
10 >>> cadeia[:]
11 'Homem Aranha'
12 >>> cadeia[::-2]
13 'HmAAh'
14 >>> cadeia[1:7:2]
```

```
15 'oe '
16 >>>
```

Notará que o fatiamento é feito entre a posição **inf** inclusive e a posição **sup** exclusive. Podemos usar os índices positivos ou negativos. Caso o índice inferior seja maior que o índice superior o resultado será uma cadeia vazia. Podemos não colocar os índices. Se não pusermos o inferior significa desde o início; se não pusermos o superior, significa até ao fim, se não pusermos ambos, significa do início ao fim do objecto. Os últimos exemplos ilustram a obtenção da subsequência de elementos igualmente espaçados.

### Exemplo 3.3

---

Obtenha a cadeia inversa e uma dada cadeia de caracteres.

A solução é trivial.

```
1 >>> cadeia[::-1]
2 'ahnarA memoH'
3 >>>
```

**3.3**

---

## Cadeias de Caracteres, **print** e formatação

Em exemplos anteriores já vimos como podemos usar a função **print** para nos devolver o valor associado a uma expressão ou para imprimir mensagens simples. Neste último caso as mensagens eram codificadas como cadeias de caracteres sempre iguais. Acabámos de ver que podemos colocar caracteres especiais numa cadeia de modo a obter uma impressão de acordo com um certo formato. Será que podemos usar algo semelhante para gerar mensagens, frases, cujo resultado final depende do valor de certos objectos? A resposta é positiva.

```
1 >>> num_1 = 1
2 >>> num_2 = 3
3 >>> mensagem_1 = "A soma de " + str(num_1) + " com " + str(
 num_2) + " dá " + str(num_1 + num_2)
4 >>> print(mensagem_1)
5 A soma de 1 com 3 dá 4
6 >>>
```

**Construtor**

A listagem acima mostra como podemos fabricar uma mensagem usando a operação de concatenação para juntar os pedaços a partir dos quais a mensagem é composta. Como os objectos variáveis de que precisamos são números e a operação de concatenação actua sobre cadeias de caracteres é preciso converter os números para cadeias de caracteres usando o **construtor str**.

Temos que convir que este modo de construção não é muito prático. Para nos ajudar existe o **operador de formatação** de cadeias de caracteres. Vejamos um exemplo de aplicação.

```

1 >>> mensagem_2 = "A soma de %d com %d dá %d" % (num_1,num_2,
2 num_1 + num_2)
3 >>> print(mensagem_2)
4 A soma de 1 com 3 dá 4
5 >>>

```

A estrutura desta construção é muito simples:

```

1 <frase_com_formato> % <valores>

```

A frase com formato não é mais do que uma cadeia de caracteres com marcas de formatação no seu interior que recorrem ao mesmo símbolo %. Os valores são os elementos que vão substituir as especificações de conversão. Quando é só um valor basta usar uma expressão, no caso de ser mais do que um valor temos que fornecer os valores separados por vírgulas e enquadrados por parênteses<sup>14</sup>. O leitor atento terá reparado que estamos perante mais um exemplo de operador sobre carregado. Com efeito, o operador % pode ser usado como acabamos de ver mas também como o operador que nos dá o resto da divisão inteira de dois números.

Vejamos um outro exemplo.

```

1 >>> num_3 = 1.0
2 >>> num_4 = 3.0
3 >>> mensagem_2 = "A divisão de %f por %f é igual a %f" % (
4 num_3, num_4, num_3 / num_4)
5 >>> print(mensagem_2)
6 A divisão de 1.000000 por 3.000000 é igual a 0.333333
7 >>> mensagem_2 = "A divisão de %.0f por %.0f é igual a %.2f" % (
8 num_3, num_4, num_3 / num_4)
9 >>> print(mensagem_2)
10 A divisão de 1 por 3 é igual a 0.33
11 >>>

```

---

<sup>14</sup>Tecnicamente, neste caso usamos um objecto do tipo **tuplo** de que falaremos mais adiante.

Neste exemplo vemos como podemos controlar o que é mostrado. Mais um exemplo para terminar<sup>15</sup>.

```

1 >>> nome = "Ernesto Costa"
2 >>> mensagem = "Exmo. Senhor\n%s" % nome
3 >>> print(mensagem)
4 Exmo. Senhor
5 Ernesto Costa
6 >>>

```

Neste último exemplo usamos cadeias de caracteres como objectos o que obriga a usar uma marca de conversão diferente. Aprece também um carácter especial para colocar o texto em duas linhas.

## Mais operações

Existem operações adicionais comuns às sequências. A tabela 3.10 identifica-as.

Tabela 3.10: Operações adicionais para cadeias de caracteres

| Literal            | Interpretação                                 |
|--------------------|-----------------------------------------------|
| <code>in</code>    | Determina se uma cadeia é sub cadeia de outra |
| <code>max</code>   | Qual o maior elemento da cadeia               |
| <code>min</code>   | Qual o menor elemento da cadeia               |
| <code>index</code> | O índice da primeira ocorrência               |
| <code>count</code> | O número de ocorrências                       |

A listagem 3.5 ilustra algumas aplicações das operações da tabela 3.10.

```

1 >>> cadeia = 'Homem Aranha'
2 >>> 'ma' in cadeia
3 False
4 >>> max(cadeia)
5 'r'
6 >>> min(cadeia)
7 ','
8 >>> cadeia.count('a')

```

---

<sup>15</sup>O leitor deve consultar o material para ver todas as possibilidades oferecidas pela linguagem.

```

9 2
10 >>> cadeia.index('a')
11 8
12 >>>

```

Listagem 3.5: Mais operações

Refira-se que para as operações `max` e `min` é usado o valor dado por `ord` a cada caractere.

## Métodos

Para além das operações já mencionadas, e que podem ser usadas com todo o tipo de sequências e não apenas cadeias de caracteres, existem outros tipos de operações específicas das cadeias de caracteres, ou seja **métodos**. A tabela 3.11 ilustram **alguns** desses métodos. O leitor interessado deve procurar no manual da linguagem a lista completas dos métodos.

| Método                              | Significado                             |
|-------------------------------------|-----------------------------------------|
| <code>s.find(sub)</code>            | O índice da primeira ocorrência ou -1   |
| <code>s.isalpha()</code>            | Verdadeiro se só letras                 |
| <code>s.isdigit()</code>            | Verdadeiro se só dígitos                |
| <code>s.center(comprimento)</code>  | Centra numa cadeia de comprimento       |
| <code>s.lower()</code>              | Converte para letras minúsculas         |
| <code>s.upper()</code>              | Converte para maiúsculas                |
| <code>s.strip()</code>              | Retira brancos à esquerda e direita     |
| <code>s.replace(velho, novo)</code> | Substitui ocorrências de velho por novo |

Tabela 3.11: Métodos das cadeias de caracteres

Vejamos exemplos concretos da sua utilização.

```

1 >>> cadeia = "TACGAUGGGTCAAUGTCGAT"
2 >>> cadeia.find('AUG')
3 4
4 >>> cadeia.find('AUG', 6)
5 12
6 >>> cadeia.find('AUT', 6)
7 -1
8 >>> cadeia.isalpha()
9 True
10 >>> cadeia.lower()

```

```

11 'tacgaugggtcaaugtcgat'
12 >>> cadeia.replace('T', 'U')
13 'UACGAUGGGUCAAAUGUCGAU'
14 >>> nome = 'Ernesto J. F. Costa'
15 >>> nome.center(50)
16 ' Ernesto J. F. Costa
17 >>> titulo = nome.center(30)
18 >>> titulo
19 ' Ernesto J. F. Costa '
20 >>> titulo.strip()
21 'Ernesto J. F. Costa'
22 >>> titulo.upper()
23 ' ERNESTO J. F. COSTA '
24 >>> '1234'.isdigit()
25 True
26 >>> '12.34'.isdigit()
27 False
28 >>>

```

Algumas notas. O método `find` pode ter argumentos opcionais que indicam o início e o fim da zona a pesquisar<sup>16</sup>. No caso do `find` se não encontra devolve -1. Chama-se a atenção para a operação semelhante a `find`, `index`. No caso desta última se o elemento não se encontrar na cadeia dá um erro.

## Construtor

À semelhança dos outros tipos de objectos, a cadeia de caracteres também tem um construtor, o método `str`.

```

1 >>> str()
2 ''
3 >>> bool('')
4 False
5 >>> str(123)
6 '123'
7 >>> str('abc')
8 'abc'
9 >>> str(12e3)
10 '12000.0'

```

---

<sup>16</sup>A procura é feita entre o primeiro valor inclusive e o último exclusive. Se este não for indicado procura-se até ao final da cadeia.

```

11 >>> str(4+5j)
12 '(4+5j)'
13 >>> str(True)
14 'True'
15 >>> a = 3
16 >>> str(a)
17 '3'
18 >>> str(4+5)
19 '9'
20 >>>

```

Quando usado sem argumento o construtor devolve a cadeia vazia. Com argumento procura transformar o objecto associado numa cadeia de caracteres. Veja-se ainda como a cadeia vazia pode ser interpretada como `False` (linhas 3 e 4).

### Exemplos

Como referimos a cadeia de ADN é uma longa sequência formada por caracteres provenientes de um alfabeto de quatro letras (A,T,C,G). A cadeia de ADN é fundamental para a construção de um organismo biológico a partir de um ovo fertilizado. A etapa inicial da construção envolve a expressão dos genes contidos no ADN, comportando ela própria dois momentos: num primeiro tempo, denominada transcrição, o ADN é transformado no ARN por substituição da base Timina por outra base, Uracil; num segundo tempo, denominada tradução, o ARN produz as proteínas que vão ser responsáveis por alimentar o processo de construção do organismo.

#### Exemplo 3.4

---

Desenvolva um programa que dada uma cadeia de ADN fabrique a correspondente cadeia de ARN.

A solução é directa e baseia-se na utilização do método `replace`.

```

1 def transcreve(adn):
2 """Tranforma o ADN em ARN."""
3 return adn.replace('T', 'U')

```

Esta solução apenas pressupõe que a cadeia de ADN é dada com caracteres maiúsculos.

3.4

**Exemplo 3.5**

Um problema importante é o da identificação dos genes numa cadeia de ARN. Felizmente existe um marcador para o seu início, formado pela subsequência de bases 'AUG', conhecido como codão de início. Sabendo isto o código para resolver esta questão é o seguinte:

```

1 def gene_pos(arn):
2 """Indica a posição do início do primeiro gene na cadeia
 de ARN."""
3 return arn.find('AUG')
4
5 def gene_pos_b(arn, pos):
6 """Indica a posição do início do primeiro gene na cadeia
 de ARN,
 a partir da posição pos."""
7 return arn.find('AUG',pos)
8

```

Agora é a vez de usar o método `find`. Ter em atenção que em caso de inexistência o resultado devolvido é `-1`. Notar ainda que na segunda versão podemos definir a posição a partir da qual nos interessa identificar o gene.

**3.5**

**Exemplo 3.6**

Sabemos que as duas fitas do ADN se encontram ligadas através das respectivas bases. Mas esta ligação não é qualquer. Assim, só podemos ter uma Adenina ligada com uma Timina, e uma Citosina com uma Guanina. Esta propriedade permite reconstruir uma das fitas conhecida a outra. É isso que o programa da listagem faz.

```

1 def complemento(adn):
2 """Fabrica o complemento da cadeia de ADN."""
3 bases = 'TACG'
4 par = 'ATGC'
5 comp = ''
6 for base in adn:
7 indice = bases.index(base)
8 comp = comp + par[indice]
9 return comp

```

Este exemplo é importante a vários títulos. Mostra o uso de operações e de métodos, mas sobretudo ilustra um **padrão** de programação. Este padrão recorre a uma ciclo e a uma variável que funciona como **acumulador** dos resultados que o interior do ciclo vai gerando. Este padrão é tão recorrente que a linguagem **Python** fornece construções que permitem a sua implementação de modo sucinto, como veremos mais adiante. Refira-se também que o controlo da execução do ciclo recorre, como é usual, a um objecto iterável, a cadeia de carácter, que é percorrida não pelos índices mas pelo seu **conteúdo**.

3.6

### Mais exemplos

Transmitir e receber textos codificados é uma tarefa que é executada em diversas situações. Os métodos para encriptar e desencriptar as mensagens são variados, mas todos supõe a existência de uma regra, implícita ou explícita, para transformar os textos.

#### Exemplo 3.7

Um método relativamente básico de codificação consiste em isolar os caracteres nas posições pares para um lado, e os caracteres nas posições ímpares para outro. Depois juntam-se as duas partes anteriormente obtidas. Podemos então resumir a nossa solução de acordo com o modelo seguinte:

```
1 def encripta(texto_normal):
2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5
6 # caracteres nas posições pares
7
8 # caracteres nas posições ímpares
9
10 # junta tudo
11
12 return texto_encriptado
```

Deste modo, dividimos o problema inicial em três sub-problemas que agora vamos ter que resolver. A ordem pela qual vamos proceder não é única. Por exemplo, podemos começar pelo terceiro sub-problema.

```
1 def encripta(texto_normal):
```

```

2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5
6 # caracteres nas posições pares
7
8 # caracteres nas posições ímpares
9
10 texto_encriptado = car_impares + car_pares
11 return texto_encriptado

```

Avançámos pouco. Mas pelo menos temos a garantia que o programa está correcto, **desde que** as variáveis `car_pares` e `car_impares`, sejam iguais às cadeias com os caracteres da cadeia inicial nas posições pares e nas posições ímpares, respectivamente. Passemos aos dois sub-problemas por resolver. Na realidade eles são semelhantes e podem ser resolvidos de modo directo recorrendo ao operador de fatiamento usado com sequências.

```

1 def encripta(texto_normal):
2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5 comp = len(texto_normal)
6 # pares
7 car_pares = texto_normal[0:comp:2]
8 # ímpares
9 car_impares = texto_normal[1:comp:2]
10 # junta
11 texto_encriptado = car_impares + car_pares
12 return texto_encriptado

```

Podemos, em alternativa, recorrer ao padrão **ciclo - acumulador - contador**.

```

1 def encripta(texto_normal):
2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5 car_pares = ""
6 car_impares = ""
7 car_conta = 0
8 for car in texto_normal:
9 if car_conta % 2 == 0: # par ou ímpar?

```

```

9 car_pares = car_pares + car
10 else:
11 car_impar = car_impar + car
12 car_conta = car_conta + 1
13 texto_encriptado = car_impar + car_pares
14 return texto_encriptado

```

## Programação Descendente

Nesta solução temos duas variáveis que assumem o papel de acumuladores, e uma (**conta**) que faz o papel de **contador**. No caso **conta** conta os caracteres já analisados e serve ainda para determinar se a posição é par ou ímpar. . Este problema permitiu-nos ilustrar uma metodologia de programação conhecida por **programação descendente**, também chamada de construção do topo para a base: decompõe-se um dado problema em subproblemas, cada um dos quais é resolvido usando a mesma abordagem reducionista, até chegarmos a problemas que se podem resolver de modo directo com as construções da linguagem.

**3.7**

---

Os métodos mais comuns de encriptação baseiam-se na ideia de chave. No método se **substituição** a chave traduz a ligação entre os caracteres. Por exemplo, uma chave pode fazer corresponder a um **a** um **h**, a um **b** um **j**, e do mesmo modo para os restantes caracteres. Claro que uma chave para ser usável deve ser uma permutação dos caracteres que podem aparecer no texto.

### Exemplo 3.8

---

Vejamos uma implementação possível do método.

```

1 def codifica(texto_normal,chave):
2 """Codifica um texto pelo método de substituição. A chave
3 é
4 dada por uma correspondência um a um entre caracteres.
5 Supõe que
6 os caracteres são as 26 letras (minúsculas) do alfabeto
7 mais o espaço em branco"""
8 alfabeto = 'abcdefghijklmnopqrstuvwxyz '
9 texto_encriptado = ''
10 for car in texto_normal:
11 indice = alfabeto.find(car)
12 texto_encriptado = texto_encriptado + chave[indice]

```

10 **return** texto\\_criptado

Como entrada temos o texto e a chave. Esta não é mais do que uma permutação dos 27 símbolos que podem aparecer no nosso texto. Para obter o texto codificado recorremos ao padrão ciclo - acumulador. O texto é percorrido pelo conteúdo. Para cada carácter do texto inicial, procuramos qual o seu índice na variável **alfabeto**. De seguida vamos buscar na chave o carácter na posição corresponde. s

**3.8**

---

## 3.5 Range

Carl Gauss foi um grande matemático e físico, tendo vivido entre os finais do século 17 e a primeira metade do século 18. Para além da sua inteligência era também muito irrequieto. Reza a lenda que um dia, na escola primária, para o manter entretido e não perturbar a aula, o seu professor mandou-o somar todos os inteiros de 1 até 100. Pegou na sua lousa e, segundos depois, respondeu 5050. Tinha acabado de descobrir a fórmula para poder efectuar o cálculo. Se fosse nos dias de hoje, e não conhecendo a fórmula, qualquer um de nós o que faria era pegar no computador para que este o ajudasse na tarefa. Mesmo assim, seria necessário introduzir os números um a um, uma tarefa um pouco fastidiosa e sujeita a erros. A menos que seja possível que o computador gere a sequência dos inteiros pretendidos. No caso de Python temos a tarefa hiperfacilitada graças a um novo tipo de objectos: **range**. Os objectos do tipo **range** são colecções ordenadas, homogéneas (todos os seus elementos são do tipo inteiro). Como sempre estes objectos têm identidade, valor e tipo.

```

1 >>> r = range(4)
2 >>> r
3 range(0, 4)
4 >>> type(r)
5 <class 'range'>
6 >>> id(r)
7 4406725824
8 >>>
```

**range** é um **iterador** que devolve os elementos de uma sequência à medida que eles são necessários, evitando deste modo que estes estejam todos em memória. Pode ter um, dois ou três argumentos. Se tiver apenas um

argumento, gera a sequência de inteiros entre 0 e esse número **exclusivé**; se tiver dois argumentos, gera a sequência de inteiros entre esses dois números, incluindo o de início e excluindo o de fim; se tiver três argumentos, gera os inteiros a partir do primeiro, até ao último **exclusivé** por saltos do valor do terceiro argumento. Os objectos criados por `range` são usados basicamente nos ciclos `for`. Se retomarmos o exemplo de criptografia (ver secção 3.4) podemos chegar a soluções diferentes das anteriormente apresentadas, recorrendo ao **padrão ciclo - acumulador** acima referido: usamos um ciclo que a cada iteração vai acumular numa variável o resultado anterior com o da iteração corrente. Dito isto, olhemos para o código.

```

1 def encripta(texto_normal):
2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5 comp = len(texto_normal)
6 # caracteres nas posições pares
7 car_pares = ""
8 for i in range(0,comp,2):
9 car_pares = car_pares + texto_normal[i]
10 # caracteres nas posições ímpares
11 car_impares = ""
12 for i in range(1,comp,2):
13 car_impares = car_impares + texto_normal[i]
14 # junta tudo
15 texto_encriptado = car_impares + car_pares
16 return texto_encriptado

```

O iterador `range` foi usado para gerar os índices nas posições que nos interessam. Podemos juntar os dois ciclos num só.

```

1 def encripta_a(texto_normal):
2 """Encriptação por separação dos caracteres nas posições
3 pares
4 e nas posições ímpares."""
5 comp = len(texto_normal)
6 car_pares = ""
7 car_impares = ""
8 for indice in range(comp):
9 if indice % 2 == 0: # par ou ímpar?
10 car_pares = car_pares + texto_normal[indice]
11 else:
12 car_impares = car_impares + texto_normal[indice]

```

```

12 texto_encriptado = car_impar + car_pares
13 return texto_encriptado

```

Agora percorremos o texto por índice. Como se vê um mesmo problema pode ter várias soluções alternativas. A nossa escolha dever ser a que apresente o melhor compromisso entre legibilidade e eficiência<sup>17</sup>.

Sendo os objectos criados com `range` sequências é natural que possamos usar as operações sobre sequências. No entanto, devido à natureza destes objectos apenas algumas operações são permitidas. Eis exemplos do que pode ser feito.

```

1 >>> r = range(10)
2 >>> 3 in r
3 True
4 >>> r[2]
5 2
6 >>> r[3:]
7 range(3, 10)
8 >>> len(r)
9 10
10 >>> r.index(3)
11 3
12 >>> r[-5]
13 5
14 >>> max(r)
15 9
16 >>> min(r)
17 0
18 >>>

```

Se pretendermos obter explicitamente **todos** os elementos da sequência podemos fazê-lo usando um construtor de sequências como `tuple`, de que falaremos mais profundadamente na secção 3.6.

```

1 >>> tuple(range(10))
2 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
3 >>> tuple(range(5,10))
4 (5, 6, 7, 8, 9)
5 >>> tuple(range(1,10,2))
6 (1, 3, 5, 7, 9)

```

---

<sup>17</sup>Existem outros aspectos a ter em conta, como por exemplo, a possibilidade de reutilizar o código.

```

7 >>> tuple(range(-10))
8 ()
9 >>> tuple(range(-10,1))
10 (-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0)
11 >>> tuple(range(-10, 1, 2))
12 (-10, -8, -6, -4, -2, 0)
13 >>>

```

## 3.6 Tuplos

Com as cadeias de caracteres introduzimos sequências cujos elementos podem ser acedidos numa base individual se o pretendermos. Existem no entanto situações em que este tipo não é de grande ajuda. Com efeito, em muitas aplicações do mundo real os objectos que manipulamos têm uma estrutura mais ou menos complexa, como por exemplo, as coordenadas de um objecto num espaço a  $n$  dimensões, os dados de uma conta bancária, ou ainda os valores das cotações de acções e a sua flutuação ao longo do dia. Imaginemos um jogo em que dois agentes habitam um espaço bidimensional, cada um deles possuindo uma arma com um certo alcance. Para saber se a arma pode ser usada, cada agente precisa calcular a distância que o separa do outro agente. Com o que já sabemos podemos efectuar esse cálculo de modo muito simples, recorrendo à formula:

$$dist(x_1, y_1, x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

que podemos traduzir num programa:

```

1 def dist(x_1,y_1, x_2,y_2):
2 """Cálculo da distância euclidiana entre os pontos 1 e 2.
3
4
5 return math.sqrt((x_2 - x_1)** 2 + (y_2 - y_1)** 2)

```

Não há nada de mal com esta solução, mas se pudéssemos tornar o código mais natural fazendo realçar o facto se tratar de pontos no espaço, seria melhor. Para isso podemos recorrer aos **tuplos**. Os **tuplos** são colecções ordenadas e heterogéneas (os seus elementos podem ser de qualquer tipo e diferentes). Quando usamos tuplos passamos a usar o objecto como um todo, mas temos que poder aceder às suas componentes.

```

1 def distancia(ponto_1, ponto_2):
2 """Cálculo da distância euclidiana entre os pontos 1 e 2.
3

```

```
3 return math.sqrt((ponto_2[0] - ponto_1[0])** 2 + (ponto_2
 [1] - ponto_1[1])** 2)
```

Cada ponto terá agora que ser representado por um tuplo<sup>18</sup>. A marca **sintática** dos tuplos são os parênteses curvos:

```
1 ponto = (1,2,3) # criar um tuplo
```

Neste exemplo, os objectos são estruturados, mas homogéneos, isto é todas as componentes são do mesmo tipo, ou de tipos compatíveis, neste exemplo números. No caso de uma conta bancária, podemos ter, por exemplo, o nome, a idade, a morada, o saldo da conta. Recorrendo a tuplos podemos armazenar essa informação:

```
1 >>> conta_1 = ('Ernesto Costa', 59, 'Coimbra', 123.45)
2 >>> nome = conta_1[0]
3 >>> nome
4 'Ernesto Costa'
5 >>>
```

## Operações

Sendo sequências, os tuplos partilham as operações básicas sobre sequências que já vimos para as cadeias de caracteres e que reproduzimos na tabela 3.12

Tabela 3.12: Operações para tuplos

| Literal | Interpretação                                 |
|---------|-----------------------------------------------|
| +       | Concatenação de cadeias de caracteres         |
| *       | Cópias de superfície de uma cadeia            |
| len     | Comprimento da cadeia                         |
| in      | Determina se uma cadeia é sub cadeia de outra |
| max     | Qual o maior elemento da cadeia               |
| min     | Qual o menor elemento da cadeia               |
| index   | O índice da primeira ocorrência               |
| count   | O número de ocorrências                       |
| [i:j:k] | Fatiamento                                    |

Alguns exemplos de utilização.

---

<sup>18</sup>Tuplos formados apenas por sequências de números são um modo natural de representar **vectores**.

```

1 >>> t_1 = (1,2,3)
2 >>> t_2 = (4,5,6,7,8,9)
3 >>> t_1 + t_2
4 (1, 2, 3, 4, 5, 6, 7, 8, 9)
5 >>> t_1 * 2
6 (1, 2, 3, 1, 2, 3)
7 >>> len(t_2)
8 6
9 >>> 4 in t_1
10 False
11 >>> max(t_1)
12 3
13 >>> min(t_2)
14 4
15 >>> t_2[2:5]
16 (6, 7, 8)
17 >>> t_1.count(3)
18 1
19 >>> t_2.index(5)
20 1
21 >>> t_1[-1]
22 3
23 >>> t_2[0:6:2]
24 (4, 6, 8)
25 >>>

```

## Empacotamento

Em certas situações, é possível referirmo-nos a tuplos sem usar a sua marca sintática, os parênteses. Isso pode acontecer quando criamos um tuplo com a instrução de atribuição, ou quando uma função devolve com **return** mais do que um resultado.

```

1 >>> t_3 = 1,2,3,4,5
2 >>> t_3
3 (1, 2, 3, 4, 5)
4 >>> def toto(n):
5 ... return n, n**2, n**3
6 ...
7 >>> res = toto(4)
8 >>> res

```

```

9 (4, 16, 64)
10 >>>

```

Chamamos a este processo empacotamento<sup>19</sup>. também é possível o processo inverso.

```

1 >>> conta_1 = ('Ernesto Costa', 59, 'Coimbra', 123.45)
2 >>> nome, idade,morada,saldo = conta_1
3 >>> nome
4 'Ernesto Costa'
5 >>> morada
6 'Coimbra'
7 >>> saldo
8 123.45
9 >>> idade
10 59
11 >>>

```

É preciso ter em atenção que temos que ter um número de variáveis à esquerda igual ao número de elementos do tuplo. Claro que podemos fazer coisas mais especializadas, como no exemplo seguinte.

```

1 >>> dados,saldo = conta_1[:3],conta_1[-1]
2 >>> dados
3 ('Ernesto Costa', 59, 'Coimbra')
4 >>> saldo
5 123.45
6 >>>

```

## Construtor

O construtor dos tuplos tem o nome da classe, ou seja `tuple`. Pode ser usado com ou sem argumento.

```

1 >>> tuplo_4 = tuple()
2 >>> tuplo_4
3 ()
4 >>> tuplo_5 = tuple('abc')
5 >>> tuplo_5
6 ('a', 'b', 'c')
7 >>> tuplo_6 = tuple('123')
8 >>> tuplo_6

```

---

<sup>19</sup>Do inglês *packing*.

```

9 ('1', '2', '3')
10 >>> tuplo_7 = tuple(123)
11 Traceback (most recent call last):
12 File "<stdin>", line 1, in <module>
13 TypeError: 'int' object is not iterable
14 >>> tuplo_8 = 4,
15 >>> tuplo_8
16 (4,)
17 >>> tuplo_9 = (4,)
18 >>> tuplo_9
19 (4,)
20 >>>

```

Como a listagem mostra, sem argumento devolve o tuplo vazio, com argumento, tenta transformar o objecto fornecido num tuplo. Nem sempre é possível, pois o objecto dado como argumento quem tem que ser **iterável** (ver linhas 10 a 13). As linhas 12 a 19 mostram a particularidade da representação de um tuplo só com um elemento.

## Representação

Já sabemos o que acontece quando criamos um objecto: ele é armazenado na memória. Quando associamos um nome ao objecto passamos a usar o nome para referir o objecto. No caso dos objectos estruturados como os tuplos, com componentes autónomas, o que guardamos na memória é uma **referência** para cada uma das componentes. A figura 3.5 mostra de forma simplificada o que acontece quando criamos o tuplo `(1, 2, 3)` e o associamos ao nome `t`.

Um aspecto importante da organização dos objectos na memória é a **partilha de objectos**, que permite optimizar o espaço ocupado. A listagem abaixo ilustra a situação. `a` e `t` estão ligados a 2 e, por isso, a identidade de `a` e `t[1]` é a mesma.

```

1 >>> t = (1,2,3)
2 >>> id(t)
3 4303573232
4 >>> id(2)
5 4299455840
6 >>> id(t[1])
7 4299455840
8 >>> a = 2
9 >>> id(a)

```

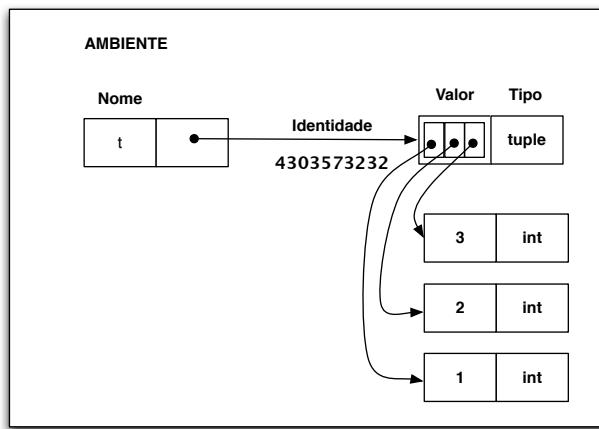


Figura 3.5: Organização dos tuplos na memória

```

10 4299455840
11 >>>

```

A figura 3.6 ilustra a situação.

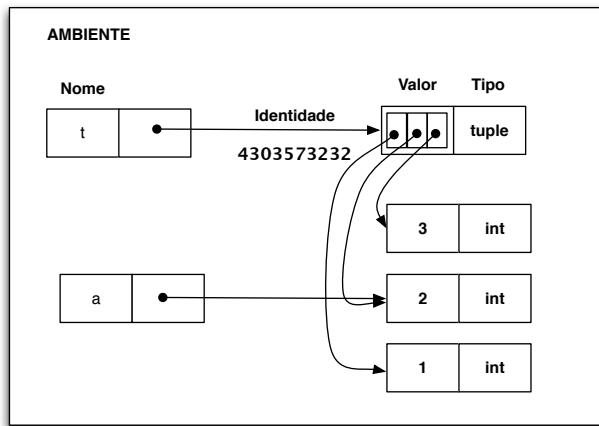


Figura 3.6: Partilha da memória

## Embrincamento

Os tuplos são colecções ordenadas de objectos heterogéneos. quando dizemos *objectos* queremos dizer que um tuplo pode ter como elemento um objecto

qualquer, incluindo um tuplo! Aceder a uma componente que é também um objecto estruturado obriga a maiores cuidados, como se ilustra abaixo.

```

1 >>> t = ('Coimbra', (40.15,8.27))
2 >>> t[1][0]
3 40.15
4 >>> t[0][3]
5 'm'
6 >>> tt = ((1,2), (3, ((4,5),6),7))
7 >>> tt[1][1][1]
8 6
9 >>> tt[1][1][0][1]
10 5
11 >>>

```

A figura 3.7 ilustra a situação para o tuplo `(1, (1,2),3)`.

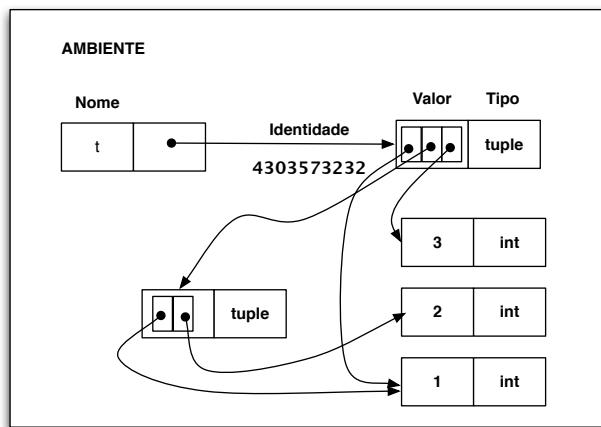


Figura 3.7: Embricamento

### Tuplos com nome

Imaginemos uma situação em que temos fichas de cidades com a respectiva localização (latitude, longitude), ou a ficha de um cliente bancário, ou qualquer outro tipo de informação em que as componentes têm um nome associado. Podemos representar informação com estas características por meio de um tuplo. Por exemplo:

```

1 >>> ficha_1 = ('Coimbra', 40.15, 8.27)

```

```

2 >>> ficha_cb_1 = ('Ernesto Costa', 'Coimbra', 59, 100)
3 >>> ficha_1[1]
4 40.15
5 >>> ficha_cb_1[3]
6 100
7 >>>

```

Ao representar a informação deste modo resulta claro que perdemos alguma informação, i.e., perdemos a noção do que representa cada uma das componentes (latitude? longitude?). Seria interessante se fosse possível associar um nome representativo a cada um dos elementos e o acesso puder ser feito pelo nome, e não pela posição (índice). Em Python existe um módulo, `collections` que implementa tuplos com nome.

```

1 >>> import collections
2 >>> Cidades = collections.namedtuple('Cidades', 'Nome lat long')
3 >>> cid_1 = Cidades(Nome='Coimbra', lat=40.15, log=8.27)
4 >>> type(cid_1)
5 <class '__main__.Cidades'>
6 >>> cid_1.Nome
7 'Coimbra'
8 >>> cid_1.lat
9 40.15
10 >>> cid_1.long
11 8.27
12 >>> cid_2 = Cidades('Lisboa', 38.42, 9.10)
13 >>> cid_3 = Cidades(long=8.40, Nome='Porto', lat=41.08)
14 >>> cid_2[1]
15 38.42
16 >>> cid_3.Nome
17 'Porto'
18 >>>

```

A listagem ilustra um aspecto importante: o utilizador pode criar novos tipos, que são implementadas como classes. A linha 2 mostra a criação do construtor do tipo, sendo que neste caso o nome do construtor é o mesmo que o nome do tipo. Este modo de proceder não sendo obrigatório é no entanto boa prática. Fica também claro que podemos aceder aos atributos pelo nome ou da forma usual, e que na criação dos objectos não é obrigatório usar o nome do atributos (linha 12). No entanto quando usamos os nomes a flexibilidade é total, ou seja, podemos indicar as componentes pela ordem que quisermos (linha 13). Esta possibilidade é muito importante em aplicações

em que o número de componentes de uma ficha é muito elevado.



## Tuplos com nome e classes

Podemos criar um tuplo com nome e visualizar a respectiva classe.

```

1 >>> OutroPonto = collections.namedtuple('OutroPonto', 'x y
2 z', verbose=True)
3 from builtins import property as _property, tuple as
4 _tuple
5 from operator import itemgetter as _itemgetter
6 from collections import OrderedDict
7
8 class OutroPonto(tuple):
9 'OutroPonto(x, y, z)'
10
11 __slots__ = ()
12
13 _fields = ('x', 'y', 'z')
14
15 def __new__(cls, x, y, z):
16 'Create new instance of OutroPonto(x, y, z)'
17 return _tuple.__new__(cls, (x, y, z))
18
19 @classmethod
20 def _make(cls, iterable, new=tuple.__new__, len=len):
21 'Make a new OutroPonto object from a sequence or
22 iterable'
23 result = new(cls, iterable)
24 if len(result) != 3:
25 raise TypeError('Expected 3 arguments, got %d'
26 % len(result))
27 return result
28
29 def __repr__(self):
30 'Return a nicely formatted representation string'
31 return self.__class__.__name__ + '(x=%r, y=%r, z=%
32 r)' % self
33
34 def _asdict(self):
35 'Return a new OrderedDict which maps field names
36 to their values'
37 return OrderedDict(zip(self._fields, self))
38
39 __dict__ = property(_asdict)
40
41 def _replace(_self, **kwds):
42 'Return a new OutroPonto object replacing
43 specified fields with new values'
44 result = _self._make(map(kwds.pop, ('x', 'y', 'z')
45 , _self))
46 if kwds:
47 raise ValueError('Got unexpected field names:
48 %s' % kwds)
49
50 return result
51
52 _fields = ('x', 'y', 'z')
53
54 def __init__(self, x, y, z):
55 'Initialize self. x, y, z'
56 self.x = x
57 self.y = y
58 self.z = z
59
60 def __getattribute__(self, name):
61 if name in self.__dict__:
62 return self.__dict__[name]
63 else:
64 return _itemgetter(name)(self)
65
66 def __setattr__(self, name, value):
67 if name in self.__dict__:
68 self.__dict__[name] = value
69 else:
70 _itemgetter(name).__set__(self, value)
71
72 def __eq__(self, other):
73 if isinstance(other, OutroPonto):
74 return self._asdict() == other._asdict()
75 return NotImplemented
76
77 def __ne__(self, other):
78 if isinstance(other, OutroPonto):
79 return self._asdict() != other._asdict()
80 return NotImplemented
81
82 def __hash__(self):
83 return hash(self._asdict())
84
85 def __reduce__(self):
86 return OutroPonto, (self._asdict(),)
87
88 def __str__(self):
89 return str(self._asdict())
90
91 def __repr__(self):
92 return '%s(%s)' % (self.__class__.__name__, self._asdict())
93
94 def __format__(self, format_spec):
95 if format_spec == '':
96 return str(self._asdict())
97 else:
98 return self._asdict().__format__(format_spec)
99
100 def __getstate__(self):
101 return self._asdict()
102
103 def __setstate__(self, state):
104 self._asdict().__setstate__(state)
105
106 def __getnewargs__(self):
107 return self._asdict()
108
109 def __getnewargs_ex__(self):
110 return self._asdict()
111
112 def __getstate__(self):
113 return self._asdict()
114
115 def __setstate__(self, state):
116 self._asdict().__setstate__(state)
117
118 def __getnewargs__(self):
119 return self._asdict()
120
121 def __getnewargs_ex__(self):
122 return self._asdict()
123
124 def __getstate__(self):
125 return self._asdict()
126
127 def __setstate__(self, state):
128 self._asdict().__setstate__(state)
129
130 def __getnewargs__(self):
131 return self._asdict()
132
133 def __getnewargs_ex__(self):
134 return self._asdict()
135
136 def __getstate__(self):
137 return self._asdict()
138
139 def __setstate__(self, state):
140 self._asdict().__setstate__(state)
141
142 def __getnewargs__(self):
143 return self._asdict()
144
145 def __getnewargs_ex__(self):
146 return self._asdict()
147
148 def __getstate__(self):
149 return self._asdict()
150
151 def __setstate__(self, state):
152 self._asdict().__setstate__(state)
153
154 def __getnewargs__(self):
155 return self._asdict()
156
157 def __getnewargs_ex__(self):
158 return self._asdict()
159
160 def __getstate__(self):
161 return self._asdict()
162
163 def __setstate__(self, state):
164 self._asdict().__setstate__(state)
165
166 def __getnewargs__(self):
167 return self._asdict()
168
169 def __getnewargs_ex__(self):
170 return self._asdict()
171
172 def __getstate__(self):
173 return self._asdict()
174
175 def __setstate__(self, state):
176 self._asdict().__setstate__(state)
177
178 def __getnewargs__(self):
179 return self._asdict()
180
181 def __getnewargs_ex__(self):
182 return self._asdict()
183
184 def __getstate__(self):
185 return self._asdict()
186
187 def __setstate__(self, state):
188 self._asdict().__setstate__(state)
189
190 def __getnewargs__(self):
191 return self._asdict()
192
193 def __getnewargs_ex__(self):
194 return self._asdict()
195
196 def __getstate__(self):
197 return self._asdict()
198
199 def __setstate__(self, state):
200 self._asdict().__setstate__(state)
201
202 def __getnewargs__(self):
203 return self._asdict()
204
205 def __getnewargs_ex__(self):
206 return self._asdict()
207
208 def __getstate__(self):
209 return self._asdict()
210
211 def __setstate__(self, state):
212 self._asdict().__setstate__(state)
213
214 def __getnewargs__(self):
215 return self._asdict()
216
217 def __getnewargs_ex__(self):
218 return self._asdict()
219
220 def __getstate__(self):
221 return self._asdict()
222
223 def __setstate__(self, state):
224 self._asdict().__setstate__(state)
225
226 def __getnewargs__(self):
227 return self._asdict()
228
229 def __getnewargs_ex__(self):
230 return self._asdict()
231
232 def __getstate__(self):
233 return self._asdict()
234
235 def __setstate__(self, state):
236 self._asdict().__setstate__(state)
237
238 def __getnewargs__(self):
239 return self._asdict()
240
241 def __getnewargs_ex__(self):
242 return self._asdict()
243
244 def __getstate__(self):
245 return self._asdict()
246
247 def __setstate__(self, state):
248 self._asdict().__setstate__(state)
249
250 def __getnewargs__(self):
251 return self._asdict()
252
253 def __getnewargs_ex__(self):
254 return self._asdict()
255
256 def __getstate__(self):
257 return self._asdict()
258
259 def __setstate__(self, state):
260 self._asdict().__setstate__(state)
261
262 def __getnewargs__(self):
263 return self._asdict()
264
265 def __getnewargs_ex__(self):
266 return self._asdict()
267
268 def __getstate__(self):
269 return self._asdict()
270
271 def __setstate__(self, state):
272 self._asdict().__setstate__(state)
273
274 def __getnewargs__(self):
275 return self._asdict()
276
277 def __getnewargs_ex__(self):
278 return self._asdict()
279
280 def __getstate__(self):
281 return self._asdict()
282
283 def __setstate__(self, state):
284 self._asdict().__setstate__(state)
285
286 def __getnewargs__(self):
287 return self._asdict()
288
289 def __getnewargs_ex__(self):
290 return self._asdict()
291
292 def __getstate__(self):
293 return self._asdict()
294
295 def __setstate__(self, state):
296 self._asdict().__setstate__(state)
297
298 def __getnewargs__(self):
299 return self._asdict()
300
301 def __getnewargs_ex__(self):
302 return self._asdict()
303
304 def __getstate__(self):
305 return self._asdict()
306
307 def __setstate__(self, state):
308 self._asdict().__setstate__(state)
309
310 def __getnewargs__(self):
311 return self._asdict()
312
313 def __getnewargs_ex__(self):
314 return self._asdict()
315
316 def __getstate__(self):
317 return self._asdict()
318
319 def __setstate__(self, state):
320 self._asdict().__setstate__(state)
321
322 def __getnewargs__(self):
323 return self._asdict()
324
325 def __getnewargs_ex__(self):
326 return self._asdict()
327
328 def __getstate__(self):
329 return self._asdict()
330
331 def __setstate__(self, state):
332 self._asdict().__setstate__(state)
333
334 def __getnewargs__(self):
335 return self._asdict()
336
337 def __getnewargs_ex__(self):
338 return self._asdict()
339
340 def __getstate__(self):
341 return self._asdict()
342
343 def __setstate__(self, state):
344 self._asdict().__setstate__(state)
345
346 def __getnewargs__(self):
347 return self._asdict()
348
349 def __getnewargs_ex__(self):
350 return self._asdict()
351
352 def __getstate__(self):
353 return self._asdict()
354
355 def __setstate__(self, state):
356 self._asdict().__setstate__(state)
357
358 def __getnewargs__(self):
359 return self._asdict()
360
361 def __getnewargs_ex__(self):
362 return self._asdict()
363
364 def __getstate__(self):
365 return self._asdict()
366
367 def __setstate__(self, state):
368 self._asdict().__setstate__(state)
369
370 def __getnewargs__(self):
371 return self._asdict()
372
373 def __getnewargs_ex__(self):
374 return self._asdict()
375
376 def __getstate__(self):
377 return self._asdict()
378
379 def __setstate__(self, state):
380 self._asdict().__setstate__(state)
381
382 def __getnewargs__(self):
383 return self._asdict()
384
385 def __getnewargs_ex__(self):
386 return self._asdict()
387
388 def __getstate__(self):
389 return self._asdict()
390
391 def __setstate__(self, state):
392 self._asdict().__setstate__(state)
393
394 def __getnewargs__(self):
395 return self._asdict()
396
397 def __getnewargs_ex__(self):
398 return self._asdict()
399
400 def __getstate__(self):
401 return self._asdict()
402
403 def __setstate__(self, state):
404 self._asdict().__setstate__(state)
405
406 def __getnewargs__(self):
407 return self._asdict()
408
409 def __getnewargs_ex__(self):
410 return self._asdict()
411
412 def __getstate__(self):
413 return self._asdict()
414
415 def __setstate__(self, state):
416 self._asdict().__setstate__(state)
417
418 def __getnewargs__(self):
419 return self._asdict()
420
421 def __getnewargs_ex__(self):
422 return self._asdict()
423
424 def __getstate__(self):
425 return self._asdict()
426
427 def __setstate__(self, state):
428 self._asdict().__setstate__(state)
429
430 def __getnewargs__(self):
431 return self._asdict()
432
433 def __getnewargs_ex__(self):
434 return self._asdict()
435
436 def __getstate__(self):
437 return self._asdict()
438
439 def __setstate__(self, state):
440 self._asdict().__setstate__(state)
441
442 def __getnewargs__(self):
443 return self._asdict()
444
445 def __getnewargs_ex__(self):
446 return self._asdict()
447
448 def __getstate__(self):
449 return self._asdict()
450
451 def __setstate__(self, state):
452 self._asdict().__setstate__(state)
453
454 def __getnewargs__(self):
455 return self._asdict()
456
457 def __getnewargs_ex__(self):
458 return self._asdict()
459
460 def __getstate__(self):
461 return self._asdict()
462
463 def __setstate__(self, state):
464 self._asdict().__setstate__(state)
465
466 def __getnewargs__(self):
467 return self._asdict()
468
469 def __getnewargs_ex__(self):
470 return self._asdict()
471
472 def __getstate__(self):
473 return self._asdict()
474
475 def __setstate__(self, state):
476 self._asdict().__setstate__(state)
477
478 def __getnewargs__(self):
479 return self._asdict()
480
481 def __getnewargs_ex__(self):
482 return self._asdict()
483
484 def __getstate__(self):
485 return self._asdict()
486
487 def __setstate__(self, state):
488 self._asdict().__setstate__(state)
489
490 def __getnewargs__(self):
491 return self._asdict()
492
493 def __getnewargs_ex__(self):
494 return self._asdict()
495
496 def __getstate__(self):
497 return self._asdict()
498
499 def __setstate__(self, state):
500 self._asdict().__setstate__(state)
501
502 def __getnewargs__(self):
503 return self._asdict()
504
505 def __getnewargs_ex__(self):
506 return self._asdict()
507
508 def __getstate__(self):
509 return self._asdict()
510
511 def __setstate__(self, state):
512 self._asdict().__setstate__(state)
513
514 def __getnewargs__(self):
515 return self._asdict()
516
517 def __getnewargs_ex__(self):
518 return self._asdict()
519
520 def __getstate__(self):
521 return self._asdict()
522
523 def __setstate__(self, state):
524 self._asdict().__setstate__(state)
525
526 def __getnewargs__(self):
527 return self._asdict()
528
529 def __getnewargs_ex__(self):
530 return self._asdict()
531
532 def __getstate__(self):
533 return self._asdict()
534
535 def __setstate__(self, state):
536 self._asdict().__setstate__(state)
537
538 def __getnewargs__(self):
539 return self._asdict()
540
541 def __getnewargs_ex__(self):
542 return self._asdict()
543
544 def __getstate__(self):
545 return self._asdict()
546
547 def __setstate__(self, state):
548 self._asdict().__setstate__(state)
549
550 def __getnewargs__(self):
551 return self._asdict()
552
553 def __getnewargs_ex__(self):
554 return self._asdict()
555
556 def __getstate__(self):
557 return self._asdict()
558
559 def __setstate__(self, state):
560 self._asdict().__setstate__(state)
561
562 def __getnewargs__(self):
563 return self._asdict()
564
565 def __getnewargs_ex__(self):
566 return self._asdict()
567
568 def __getstate__(self):
569 return self._asdict()
570
571 def __setstate__(self, state):
572 self._asdict().__setstate__(state)
573
574 def __getnewargs__(self):
575 return self._asdict()
576
577 def __getnewargs_ex__(self):
578 return self._asdict()
579
580 def __getstate__(self):
581 return self._asdict()
582
583 def __setstate__(self, state):
584 self._asdict().__setstate__(state)
585
586 def __getnewargs__(self):
587 return self._asdict()
588
589 def __getnewargs_ex__(self):
590 return self._asdict()
591
592 def __getstate__(self):
593 return self._asdict()
594
595 def __setstate__(self, state):
596 self._asdict().__setstate__(state)
597
598 def __getnewargs__(self):
599 return self._asdict()
600
601 def __getnewargs_ex__(self):
602 return self._asdict()
603
604 def __getstate__(self):
605 return self._asdict()
606
607 def __setstate__(self, state):
608 self._asdict().__setstate__(state)
609
610 def __getnewargs__(self):
611 return self._asdict()
612
613 def __getnewargs_ex__(self):
614 return self._asdict()
615
616 def __getstate__(self):
617 return self._asdict()
618
619 def __setstate__(self, state):
620 self._asdict().__setstate__(state)
621
622 def __getnewargs__(self):
623 return self._asdict()
624
625 def __getnewargs_ex__(self):
626 return self._asdict()
627
628 def __getstate__(self):
629 return self._asdict()
630
631 def __setstate__(self, state):
632 self._asdict().__setstate__(state)
633
634 def __getnewargs__(self):
635 return self._asdict()
636
637 def __getnewargs_ex__(self):
638 return self._asdict()
639
640 def __getstate__(self):
641 return self._asdict()
642
643 def __setstate__(self, state):
644 self._asdict().__setstate__(state)
645
646 def __getnewargs__(self):
647 return self._asdict()
648
649 def __getnewargs_ex__(self):
650 return self._asdict()
651
652 def __getstate__(self):
653 return self._asdict()
654
655 def __setstate__(self, state):
656 self._asdict().__setstate__(state)
657
658 def __getnewargs__(self):
659 return self._asdict()
660
661 def __getnewargs_ex__(self):
662 return self._asdict()
663
664 def __getstate__(self):
665 return self._asdict()
666
667 def __setstate__(self, state):
668 self._asdict().__setstate__(state)
669
670 def __getnewargs__(self):
671 return self._asdict()
672
673 def __getnewargs_ex__(self):
674 return self._asdict()
675
676 def __getstate__(self):
677 return self._asdict()
678
679 def __setstate__(self, state):
680 self._asdict().__setstate__(state)
681
682 def __getnewargs__(self):
683 return self._asdict()
684
685 def __getnewargs_ex__(self):
686 return self._asdict()
687
688 def __getstate__(self):
689 return self._asdict()
690
691 def __setstate__(self, state):
692 self._asdict().__setstate__(state)
693
694 def __getnewargs__(self):
695 return self._asdict()
696
697 def __getnewargs_ex__(self):
698 return self._asdict()
699
700 def __getstate__(self):
701 return self._asdict()
702
703 def __setstate__(self, state):
704 self._asdict().__setstate__(state)
705
706 def __getnewargs__(self):
707 return self._asdict()
708
709 def __getnewargs_ex__(self):
710 return self._asdict()
711
712 def __getstate__(self):
713 return self._asdict()
714
715 def __setstate__(self, state):
716 self._asdict().__setstate__(state)
717
718 def __getnewargs__(self):
719 return self._asdict()
720
721 def __getnewargs_ex__(self):
722 return self._asdict()
723
724 def __getstate__(self):
725 return self._asdict()
726
727 def __setstate__(self, state):
728 self._asdict().__setstate__(state)
729
730 def __getnewargs__(self):
731 return self._asdict()
732
733 def __getnewargs_ex__(self):
734 return self._asdict()
735
736 def __getstate__(self):
737 return self._asdict()
738
739 def __setstate__(self, state):
740 self._asdict().__setstate__(state)
741
742 def __getnewargs__(self):
743 return self._asdict()
744
745 def __getnewargs_ex__(self):
746 return self._asdict()
747
748 def __getstate__(self):
749 return self._asdict()
750
751 def __setstate__(self, state):
752 self._asdict().__setstate__(state)
753
754 def __getnewargs__(self):
755 return self._asdict()
756
757 def __getnewargs_ex__(self):
758 return self._asdict()
759
760 def __getstate__(self):
761 return self._asdict()
762
763 def __setstate__(self, state):
764 self._asdict().__setstate__(state)
765
766 def __getnewargs__(self):
767 return self._asdict()
768
769 def __getnewargs_ex__(self):
770 return self._asdict()
771
772 def __getstate__(self):
773 return self._asdict()
774
775 def __setstate__(self, state):
776 self._asdict().__setstate__(state)
777
778 def __getnewargs__(self):
779 return self._asdict()
780
781 def __getnewargs_ex__(self):
782 return self._asdict()
783
784 def __getstate__(self):
785 return self._asdict()
786
787 def __setstate__(self, state):
788 self._asdict().__setstate__(state)
789
790 def __getnewargs__(self):
791 return self._asdict()
792
793 def __getnewargs_ex__(self):
794 return self._asdict()
795
796 def __getstate__(self):
797 return self._asdict()
798
799 def __setstate__(self, state):
800 self._asdict().__setstate__(state)
801
802 def __getnewargs__(self):
803 return self._asdict()
804
805 def __getnewargs_ex__(self):
806 return self._asdict()
807
808 def __getstate__(self):
809 return self._asdict()
810
811 def __setstate__(self, state):
812 self._asdict().__setstate__(state)
813
814 def __getnewargs__(self):
815 return self._asdict()
816
817 def __getnewargs_ex__(self):
818 return self._asdict()
819
820 def __getstate__(self):
821 return self._asdict()
822
823 def __setstate__(self, state):
824 self._asdict().__setstate__(state)
825
826 def __getnewargs__(self):
827 return self._asdict()
828
829 def __getnewargs_ex__(self):
830 return self._asdict()
831
832 def __getstate__(self):
833 return self._asdict()
834
835 def __setstate__(self, state):
836 self._asdict().__setstate__(state)
837
838 def __getnewargs__(self):
839 return self._asdict()
840
841 def __getnewargs_ex__(self):
842 return self._asdict()
843
844 def __getstate__(self):
845 return self._asdict()
846
847 def __setstate__(self, state):
848 self._asdict().__setstate__(state)
849
850 def __getnewargs__(self):
851 return self._asdict()
852
853 def __getnewargs_ex__(self):
854 return self._asdict()
855
856 def __getstate__(self):
857 return self._asdict()
858
859 def __setstate__(self, state):
860 self._asdict().__setstate__(state)
861
862 def __getnewargs__(self):
863 return self._asdict()
864
865 def __getnewargs_ex__(self):
866 return self._asdict()
867
868 def __getstate__(self):
869 return self._asdict()
870
871 def __setstate__(self, state):
872 self._asdict().__setstate__(state)
873
874 def __getnewargs__(self):
875 return self._asdict()
876
877 def __getnewargs_ex__(self):
878 return self._asdict()
879
880 def __getstate__(self):
881 return self._asdict()
882
883 def __setstate__(self, state):
884 self._asdict().__setstate__(state)
885
886 def __getnewargs__(self):
887 return self._asdict()
888
889 def __getnewargs_ex__(self):
890 return self._asdict()
891
892 def __getstate__(self):
893 return self._asdict()
894
895 def __setstate__(self, state):
896 self._asdict().__setstate__(state)
897
898 def __getnewargs__(self):
899 return self._asdict()
900
901 def __getnewargs_ex__(self):
902 return self._asdict()
903
904 def __getstate__(self):
905 return self._asdict()
906
907 def __setstate__(self, state):
908 self._asdict().__setstate__(state)
909
910 def __getnewargs__(self):
911 return self._asdict()
912
913 def __getnewargs_ex__(self):
914 return self._asdict()
915
916 def __getstate__(self):
917 return self._asdict()
918
919 def __setstate__(self, state):
920 self._asdict().__setstate__(state)
921
922 def __getnewargs__(self):
923 return self._asdict()
924
925 def __getnewargs_ex__(self):
926 return self._asdict()
927
928 def __getstate__(self):
929 return self._asdict()
930
931 def __setstate__(self, state):
932 self._asdict().__setstate__(state)
933
934 def __getnewargs__(self):
935 return self._asdict()
936
937 def __getnewargs_ex__(self):
938 return self._asdict()
939
940 def __getstate__(self):
941 return self._asdict()
942
943 def __setstate__(self, state):
944 self._asdict().__setstate__(state)
945
946 def __getnewargs__(self):
947 return self._asdict()
948
949 def __getnewargs_ex__(self):
950 return self._asdict()
951
952 def __getstate__(self):
953 return self._asdict()
954
955 def __setstate__(self, state):
956 self._asdict().__setstate__(state)
957
958 def __getnewargs__(self):
959 return self._asdict()
960
961 def __getnewargs_ex__(self):
962 return self._asdict()
963
964 def __getstate__(self):
965 return self._asdict()
966
967 def __setstate__(self, state):
968 self._asdict().__setstate__(state)
969
970 def __getnewargs__(self):
971 return self._asdict()
972
973 def __getnewargs_ex__(self):
974 return self._asdict()
975
976 def __getstate__(self):
977 return self._asdict()
978
979 def __setstate__(self, state):
980 self._asdict().__setstate__(state)
981
982 def __getnewargs__(self):
983 return self._asdict()
984
985 def __getnewargs_ex__(self):
986 return self._asdict()
987
988 def __getstate__(self):
989 return self._asdict()
990
991 def __setstate__(self, state):
992 self._asdict().__setstate__(state)
993
994 def __getnewargs__(self):
995 return self._asdict()
996
997 def __getnewargs_ex__(self):
998 return self._asdict()
999
1000 def __getstate__(self):
1001 return self._asdict()
1002
1003 def __setstate__(self, state):
1004 self._asdict().__setstate__(state)
1005
1006 def __getnewargs__(self):
1007 return self._asdict()
1008
1009 def __getnewargs_ex__(self):
1010 return self._asdict()
1011
1012 def __getstate__(self):
1013 return self._asdict()
1014
1015 def __setstate__(self, state):
1016 self._asdict().__setstate__(state)
1017
1018 def __getnewargs__(self):
1019 return self._asdict()
1020
1021 def __getnewargs_ex__(self):
1022 return self._asdict()
1023
1024 def __getstate__(self):
1025 return self._asdict()
1026
1027 def __setstate__(self, state):
1028 self._asdict().__setstate__(state)
1029
1030 def __getnewargs__(self):
1031 return self._asdict()
1032
1033 def __getnewargs_ex__(self):
1034 return self._asdict()
1035
1036 def __getstate__(self):
1037 return self._asdict()
1038
1039 def __setstate__(self, state):
1040 self._asdict().__setstate__(state)
1041
1042 def __getnewargs__(self):
1043 return self._asdict()
1044
1045 def __getnewargs_ex__(self):
1046 return self._asdict()
1047
1048 def __getstate__(self):
1049 return self._asdict()
1050
1051 def __setstate__(self, state):
1052 self._asdict().__setstate__(state)
1053
1054 def __getnewargs__(self):
1055 return self._asdict()
1056
1057 def __getnewargs_ex__(self):
1058 return self._asdict()
1059
1060 def __getstate__(self):
1061 return self._asdict()
1062
1063 def __setstate__(self, state):
1064 self._asdict().__setstate__(state)
1065
1066 def __getnewargs__(self):
1067 return self._asdict()
1068
1069 def __getnewargs_ex__(self):
1070 return self._asdict()
1071
1072 def __getstate__(self):
1073 return self._asdict()
1074
1075 def __setstate__(self, state):
1076 self._asdict().__setstate__(state)
1077
1078 def __getnewargs__(self):
1079 return self._asdict()
1080
1081 def __getnewargs_ex__(self):
1082 return self._asdict()
1083
1084 def __getstate__(self):
1085 return self._asdict()
1086
1087 def __setstate__(self, state):
1088 self._asdict().__setstate__(state)
1089
1090 def __getnewargs__(self):
1091 return self._asdict()
1092
1093 def __getnewargs_ex__(self):
1094 return self._asdict()
1095
1096 def __getstate__(self):
1097 return self._asdict()
1098
1099 def __setstate__(self, state):
1100 self._asdict().__setstate__(state)
1101
1102 def __getnewargs__(self):
1103 return self._asdict()
1104
1105 def __getnewargs_ex__(self):
1106 return self._asdict()
1107
1108 def __getstate__(self):
1109 return self._asdict()
1110
1111 def __setstate__(self, state):
1112 self._asdict().__setstate__(state)
1113
1114 def __getnewargs__(self):
1115 return self._asdict()
1116
1117 def __getnewargs_ex__(self):
1118 return self._asdict()
1119
1120 def __getstate__(self):
1121 return self._asdict()
1122
1123 def __setstate__(self, state):
1124 self._asdict().__setstate__(state)
1125
1126 def __getnewargs__(self):
1127 return self._asdict()
1128
1129 def __getnewargs_ex__(self):
1130 return self._asdict()
1131
1132 def __getstate__(self):
1133 return self._asdict()
1134
1135 def __setstate__(self, state):
1136 self._asdict().__setstate__(state)
1137
1138 def __getnewargs__(self):
1139 return self._asdict()
1140
1141 def __getnewargs_ex__(self):
1142 return self._asdict()
1143
1144 def __getstate__(self):
1145 return self._asdict()
1146
1147 def __setstate__(self, state):
1148 self._asdict().__setstate__(state)
1149
1150 def __getnewargs__(self):
1151 return self._asdict()
1152
1153 def __getnewargs_ex__(self):
1154 return self._asdict()
1155
1156 def __getstate__(self):
1157 return self._asdict()
1158
1159 def __setstate__(self, state):
1160 self._asdict().__setstate__(state)
1161
1162 def __getnewargs__(self):
1163 return self._asdict()
1164
1165 def __getnewargs_ex__(self):
1166 return self._asdict()
1167
1168 def __getstate__(self):
1169 return self._asdict()
1170
1171 def __setstate__(self, state):
1172 self._asdict().__setstate__(state)
1173
1174 def __getnewargs__(self):
1175 return self._asdict()
1176
1177 def __getnewargs_ex__(self):
1178 return self._asdict()
1179
1180 def __getstate__(self):
1181 return self._asdict()
1182
1183 def __setstate__(self, state):
1184 self._asdict().__setstate__(state)
1185
1186 def __getnewargs__(self):
1187 return self._asdict()
118
```

### 3.7 Intermezzo

Todo o programador se confronta com a questão de escolher, de entre várias soluções para um dado problema, aquela que lhe parece melhor. A escolha é geralmente a que resulta de um compromisso de vários objectivos, por vezes contraditórios: correcção (!), legibilidade, simplicidade, potencial de reutilização, facilidade de manutenção, economia de recursos computacionais, são alguns dos aspectos a considerar. Mesmo quando o **algoritmo** é o mesmo, as características da linguagem podem sugerir diferentes soluções. Vejamos um exemplo concreto de problema, em que o objectivo primeiro é o de determinar qual o melhor elemento numa colecção.

```

1 def max_elem_1(elementos,funcao):
2 """Qual o elemento de maior valor de acordo com a função.
3 """
4
5 # Inicializa
6 melhor_valor = None
7 melhor_elem = None
8
9 # Testa e actualiza
10 for elem in elementos:
11 valor = funcao(elem)
12 if melhor_valor == None or valor > melhor_valor:
13 # actualiza
14 melhor_valor = valor
15 melhor_elem = elem
16
17 return melhor_elem

```

Esta é uma solução clássica: analiso todos os elementos de modo ordenado e sempre que tenho um novo melhor actualizo. Notar como se inicializam as variáveis, o que nos obriga a um teste adicional comparando o valor do melhor com `None`. Mantendo o mesmo princípio podemos efectuar pequenas melhorias, admitindo que a colecção dos elementos tem pelo menos um.

```

1 def max_elem_2(elementos,funcao):
2 """Qual o elemento de maior valor de acordo com a função.
3 """
4
5 # Inicializa
6 melhor_elem = elementos[0]
7 melhor_valor = funcao(melhor_elem)
8
9 # Testa e actualiza
10 for elem in elementos[1:]:
11 valor = funcao(elem)
12 if valor > melhor_valor:
13 melhor_elem = elem
14 melhor_valor = valor

```

```

10 # actualiza
11 melhor_valor = valor
12 melhor_elem = elem
13
14 return melhor_elem

```

Conhecendo bem a linguagem Python podemos propor uma solução mais legível, reduzindo a solução a uma linha de código.

```

1 def max_elem_3(elementos,funcao):
2 """Qual o elemento de maior valor de acordo com a função.
3 """
4
5 return max(elementos,key=funcao)

```

Nesta solução reencontramos a função `max` e ficamos a saber que se pode usar um segundo argumento que explicita qual o critério para a comparação.

```

1 def max_elem_4(elementos,funcao):
2 """Qual o elemento de maior valor de acordo com a função.
3 """
4
5 valores = (funcao(elem) for elem in elementos)
6
7 return max(elementos,key=funcao)

```

Esta quarta solução também depende de conhecimento profundo da linguagem Python , em particular da existência de **expressões geradoras** (ver linha 3). Este é um tema mais avançado a que voltaremos mais à frente no livro. Uma outra questão que se pode colocar, é a de saber em que medida estas soluções são reutilizáveis? Por exemplo, se pretendermos não apenas o elemento de maior valor mas também o seu valor. As alterações neste caso são mínimas. Mostramos dois exemplos.

[Expressões Geradoras](#)

```

1 def max_elem_valor_1(elementos,funcao):
2 """Devolve o elemento de valor máximo e o respectivo valor.
3 """
4
5 # Inicializa
6 melhor_elem = elementos[0]
7 melhor_valor = funcao(melhor_elem)
8
9 # Testa e actualiza
10 for elem in elementos[1:]:
11 valor = funcao(elem)
12 if valor > melhor_valor:
13 # actualiza
14 melhor_valor = valor
15 melhor_elem = elem
16
17 return melhor_elem, melhor_valor

```

```

14
15
16 def max_elem_valor_2(elementos,funcao):
17 """Qual o elemento de maior valor de acordo com a função.
18
19 melhor_elem = max(elementos,key=funcao)
20 melhor_valor = funcao(melhor_elem)
21 return melhor_elem, melhor_valor

```

## 3.8 Mutabilidade

Na secção 3.6 referimos a possibilidade de os objectos partilharem zonas da memória. Quais são as consequências possíveis dessa partilha? Consideremos a situação da listagem 3.6.

```

1 >>> t_1 = (1,2,3)
2 >>> t_2 = t_1
3 >>> t_3 = (1,2,3)
4 >>> t_4 = (1,(1,2,3),3)
5 >>> a = 2
6 >>> b = t_4[1]
7 >>>

```

Listagem 3.6: Partilha

Criamos 5 tuplos (`t_1`, `t_2`, `t_2`, `t_2`, e `b`) e um objecto numérico (`a`). Temos várias situações de partilha, sendo que esta é feita através das **referências**, ou seja, das identidades dos objectos. Notar com atenção o que acontece nas três primeiras situações. Do ponto de vista da memória, e de modo muito simplificado, esta fica organizada como ilustra a figura 3.8.

Dada a ilustração não nos podemos admirar do resultado de efectuarmos várias consultas às identidades dos objectos.

```

1 >>> id(t_1)
2 4303573392
3 >>> id(t_2)
4 4303573392
5 >>> id(t_3)
6 4303574512
7 >>> id(t_4)
8 4303574352
9 >>> id(a)

```

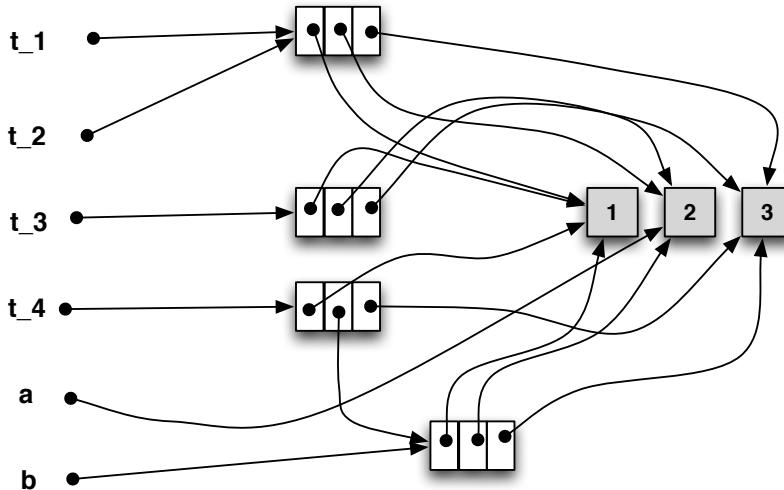


Figura 3.8: Partilha da memória

```

10 4299455840
11 >>> id(b)
12 4303573472
13 >>> id(t_1[1])
14 4299455840
15 >>> id(t_3[1])
16 4299455840
17 >>> id(b[1])
18 4299455840
19 >>>

```

Vamos então proceder a algumas alterações e analisar as consequências. Se alterarmos **t\_2**, **t\_1** também se altera? Se alterarmos o segundo elemento de **t\_4**, o que acontece a **b**? Alterando **a** ou **t\_3** o que acontece ao valor dos outros objectos? A listagem 3.8 dá-nos a resposta.

```

1 >>> t_2 = (4,5,6)
2 >>> t_1
3 (1, 2, 3)
4 >>> id(t_2)
5 4303574672
6 >>> t_4 = (1, (7,8,9),3)
7 >>> b
8 (1, 2, 3)

```

```

9 >>> a = 10
10 >>> t_1
11 (1, 2, 3)
12 >>> t_3 = (11, 12, 13)
13 >>>

```

**Imutabilidade**

A razão porque estas mudanças não provocam efeitos indesejados, isto é, alterando um objecto alteramos indirectamente outro se partilharem o que foi alterado, reside no facto de os tuplos serem objectos **imutáveis**: não é possível alterar o seu valor sem alterar a sua identidade, criando assim um **novo** objecto. Esta propriedade também permite explicar o modo alteramos parte do objecto. A listagem abaixo mostra o que acontece quando tentamos fazer essa alteração directamente sobre a componente a modificar.

```

1 >>> t_4[1] = (14,15,16)
2 Traceback (most recent call last):
3 File "<stdin>", line 1, in <module>
4 TypeError: 'tuple' object does not support item assignment
5 >>>

```

**Imutabilidade**

Não são só os tuplos que são imutáveis. Os números, as cadeias de caracteres, os *range*, também são imutáveis.

Analisemos agora o caso das cadeias de caracteres.

```

1 >>> cadeia = 'Homem Aranha'
2 >>> cadeia[6] = 'I'
3 Traceback (most recent call last):
4 File "<string>", line 1, in <fragment>
5 TypeError: 'str' object does not support item assignment
6 >>>

```

Não é possível a alteração! No entanto podemos ultrapassar em parte esta dificuldade construindo uma **nova** cadeia e associar o objecto resultante ao mesmo nome.

```

1 >>> cadeia = cadeia[:6] + 'I' + cadeia[7:]
2 >>> cadeia
3 'Homem Iranha'
4 >>>

```

O mesmo processo serve para inserir ou eliminar caracteres ou, generalizando, sub-cadeias de caracteres.

```

1 >>> cadeia = 'Homem Aranha'
2 >>> cadeia = cadeia[:6] + 'Quase ' + cadeia[6:]
3 >>> cadeia
4 'Homem Quase Aranha'
5 >>> id(cadeia)
6 4303038768
7 >>> cadeia = cadeia[:6] + cadeia[12:]
8 >>> cadeia
9 'Homem Aranha'
10 >>> id(cadeia)
11 4303023944
12 >>>

```

Como seria de esperar as identidades são diferentes.

### Mutabilidade e memória

Vejamos agora como as coisas se passam ao nível da memória, usando exemplos simples.

```

1 >>> t_1 = (1,2,3)
2 >>> t_2 = t_1
3 >>> id(t_1)
4 4303574752
5 >>> id(t_2)
6 4303574752
7 >>> t_2 = (4,5,6)
8 >>> t_1
9 (1, 2, 3)
10 >>> id(t_1)
11 4303574752
12 >>> id(t_2)
13 4303573392
14 >>>

```

Neste caso é todo o objecto que é alterado e a partilha terminou (ver figura 3.9).

Mas se só alterarmos uma parte, o que acontece?

```

1 >>> t_1 = (1,2,3)
2 >>> t_2 = t_1
3 >>> t_2 = (1,4,3)
4 >>> id(t_1[1])

```

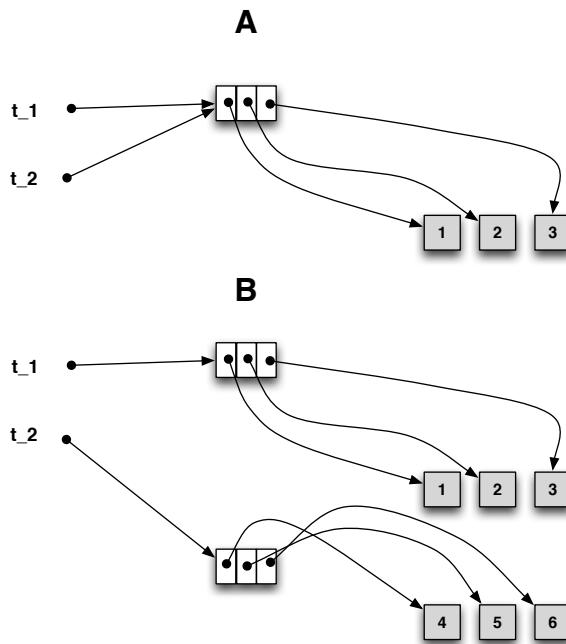


Figura 3.9: Mutabilidade (I): antes (A) e depois (B)

```

5 4299455840
6 >>> id(t_2[1])
7 4299455904
8 >>> id(t_1[0])
9 4299455808
10 >>> id(t_2[0])
11 4299455808
12 >>>

```

Mantém-se a partilha do que não foi alterado!

## Sumário

Neste capítulo introduzimos um grupo de objectos primitivos (números e booleanos) e objectos estruturados simples (cadeias de caracteres, tuplos e range). Identificámos as marcas sintáticas (i.e., os literais) para cada tipo e as operações sobre esses objectos, tendo realçado o construtor de cada tipo. Referimos a existência de critérios de classificação para os tipos (coleção, ordem, homogeneidade, mutabilidade). Também foi tratada a questão da

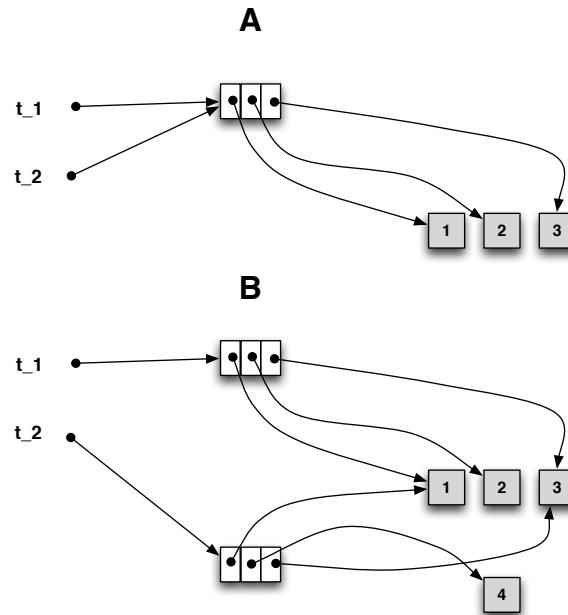


Figura 3.10: Mutabilidade (II): antes (A) e depois (B)

representação em memória dos objectos, simples e estruturados. Foi introduzido o padrão de programação ciclo - acumulador, e foi feita uma breve referência a aspectos de metodologia da programação

## Teste os seus conhecimentos

Tente responder às seguintes questões que foram tratadas neste capítulo.

1. O que entende por programação descendente
2. O que são objectos imutáveis
3. Qual a diferença entre a função `range` com um argumento, dois argumentos, ou três argumentos?
4. O que é uma variável com o papel de contador?
5. O que é uma variável com o papel de acumulador?
6. Como se pode modificar uma cadeia de caracteres obtendo uma nova?

7. O que significa fatiamento?
8. Qual é o construtor do tipo cadeia de caracteres?
9. Que diferença existem entre os métodos `find` e `index`?

## Exercícios

### Exercício 3.1 MF

Arranque o interpretador **Python** e associe um nome com um objecto do tipo inteiro. Recorra ao comando `help` para saber mais coisas sobre o objecto. Observe o resultado e tire conclusões.

### Exercício 3.2 F

Escreva um programa que calcula a área de um triângulo por recurso à fórmula de Heron. Neste caso, se o triângulo tiver como lados  $a$ ,  $b$  e  $c$  a área é dada por:

$$\text{área} = \sqrt{s \times (s - a) \times (s - b) \times (s - c)}$$

com:

$$s = \frac{a + b + c}{2}$$

### Exercício 3.3 F

Suponha que quer colocar uma escada encostada a uma parede de sua casa por forma que ela alcance uma dada altura  $alt$ . Por razões de segurança a escada deve fazer um dado ângulo  $ang$  com o solo. Escreva um programa que determine o comprimento  $comp$  da escada. A relação entre as três variáveis é dada por:

$$comp = \frac{alt}{\operatorname{seno}(ang)}$$

O cálculo do seno é feito em radianos mas admita que o ângulo é dado pelo utilizador em graus. A relação é dada por:

$$radianos = \frac{\pi}{180} \times graus$$

### Exercício 3.4 F

O valor do batimento cardíaco máximo tem sido objecto de vários estudos, existindo várias fórmulas que dão o seu valor médio. Uma delas é:

$$163 + 1.16 * \text{idade} - 0.018 * \text{idade}^2$$

Desenvolva um programa que dada a idade calcule o valor médio do batimento cardíaco máximo.

### Exercício 3.5 F

Admitamos que colocamos uma certa quantidade de dinheiro a render. A fórmula que nos permite calcular o valor ao fim de vários anos, conhecida a taxa de juro fixa é:

$$v * (1 + t)^a$$

Escreva um programa que conhecido o valor inicial ( $v$ ), a taxa de juro ( $t$ ), e os anos decorridos ( $a$ ), calcula o valor ao fim desses anos. Use este programa para saber ao fim de quanto tempo consegue duplicar o seu dinheiro.

### Exercício 3.6 F

Problema semelhante a 3.8 só que agora a taxa pode ser composta várias vezes ao ano. Neste caso a fórmula passa a ser:

$$v * \left(1 + \frac{t}{n}\right)^{n*a}$$

Faça alguns testes para o caso em que a composição é mensal ( $n = 12$ ) e compare com os resultados obtidos no caso da acumulação ser anual ( $n = 1$ ).

### Exercício 3.7 F

Escreva um programa que permita descodificar um texto que foi codificado com o método da separação entre caracteres nas posições pares e caracteres nas posições ímpares.

### Exercício 3.8 F

Escreva um programa que lhe permita descodificar um texto codificado pelo método da chave de substituição.

### Exercício 3.9 F

Apresentámos um programa que nos permite calcular a cadeia complementar de uma dada cadeia de ADN. Apresente uma solução alternativa ao problema. **Sugestão:** Pense em usar a instrução condicional **if**.

### Exercício 3.10 F

Desenvolva um programa que lhe permitir gerar uma cadeia de ADN. O tamanho deve ser um parâmetro do seu programa. **Sugestão:** Pense em usar o padrão ciclo - acumulador.

### Exercício 3.11 F

Desenvolva um programa que substitua as ocorrências de vogais numa cadeia de caracteres por espaços em branco.

### Exercício 3.12 F

Escreva um programa que dada uma cadeia de caracteres e um número inteiro positivo, imprima todas as suas **sub-cadeias** de comprimento igual ao número. Por exemplo, para a cadeia de caracteres 'Monty Python' e comprimento 3, o resultado será o apresentado na listagem.

```

1 Mon
2 ont
3 nty
4 ty
5 y P
6 Py
7 Pyt
8 yth
9 tho
10 hon
```

### Exercício 3.13 M

Associe uma variável à cadeia de caracteres 'Monty Python'. Escreva um programa que permita obter a saída da listagem abaixo. Trata-se de todos os **prefixos** da cadeia.

```

1 M
2 Mo
3 Mon
4 Mont
5 Monty
6 Monty
7 Monty P
8 Monty Py
9 Monty Pyt
10 Monty Pyth
11 Monty Python
```

12 Monty Python

### Exercício 3.14 M

Semelhante ao problema 3.8, só que agora pretende-se obter na saída todos os **sufixos**.

```
1 Monty Python
2 Monty Pytho
3 Monty Pyth
4 Monty Pyt
5 Monty Py
6 Monty P
7 Monty
8 Monty
9 Mont
10 Mon
11 Mo
12 M
```

### Exercício 3.15 Módulo turtle F

À semelhança dos humanos a nossa amiga tartaruga **tarta** tem um **código genético** baseado num alfabeto de quatro letras:  $\{f', t', e', d'\}$ . Ainda como no caso dos humanos, aquilo que ela é (faz) resulta da *expressão* do seu ADN. Para tal, cada letra está ligada a uma acção simples : 'f', move-se para a frente, 't', move-se para trás, 'd' roda à direita e, 'e' roda à esquerda. Por exemplo, se o seu ADN for 'feftd' **tarta** passa o tempo a executar as acções: para a frente ('f'), roda à esquerda ('e'), para a frente ('f'), para trás ('t') e roda à direita ('d').

Escreva um **simulador** que permita mostrar o percurso da nossa amiga, conhecido o seu ADN. Admita que os movimentos são de valor constante, o mesmo sucedendo com as rotações.

### Exercício 3.16 Módulo random Módulo turtle F

Pretendemos um simulador semelhante ao problema 3.8, mas em que, agora, os movimentos e as rotações tenham valores aleatórios. Escolha no entanto os valores possíveis dentro de uma gama razoável!

### Exercício 3.17 Módulo random Módulo turtle F

Nos problemas 3.8 e 3.8 o ADN da tartaruga é determinado à partida.

Escreva um simulador em que o ADN é definido primeiro de modo aleatório e depois executado. Também neste caso suponha que os valores das deslocações e rotações podem variar aleatoriamente. O único argumento da função deve ser o **comprimento** do ADN.

**Exercício 3.18 M**

Um método para codificar/descodificar um texto baseia-se na ideia de substituir um carácter pelo carácter que está a uma certa **distância** dele. Por exemplo, se a distância escolhida for 2, então o **c** substitui o **a**, o **d** substitui o **b** e assim sucessivamente. Escreva um programa para codificar e outro para descodificar recorrendo a este método. A distância deve ser um parâmetro do problema e pode ser positiva ou negativa. Pense como vai resolver o caso dos caracteres nas extremidades do alfabeto.

# Instruções Destrutivas

## Objectivos

- ✓ Compreender o conceito de instrução destrutiva
- ✓ Aprofundar os conceitos de entrada e saída

Um programa encontra-se normalmente organizado em vários módulos sendo que cada um deles é formado por uma sequência de **comandos**. Esses comandos dividem-se em **expressões**, **instruções** ou ainda **definições**. Já vimos exemplos de cada um destes três tipos de comandos. Vamos agora concentrarmo-nos nas instruções. De um modo simples podemos dizer que as instruções **fazem** coisas. Quando executamos um programa algumas instruções alteram ou criam objectos e associam esses objectos a nomes, enquanto que outras apenas servem para definir a próxima instrução a ser executada. Às primeiras chamamos **instruções destrutivas** e às segundas **instruções de controlo**. No modelo que temos vindo a explorar num dado instante os objectos têm um determinado conjunto de características (em particular têm um dado valor) e o programa encontra-se a executar uma dada instrução. Dizemos que o programa se encontra num dado **estado**. A sequência de estados por que vai passando o programa ao longo do tempo constitui uma **computação**. Neste capítulo iremos trabalhar o conceito de instruções destrutivas que, como veremos, se subdividem em três categorias:

- atribuição
- leitura
- escrita

Alguns dos conceitos apresentados serão apenas aprofundamentos do que já foi dito nos capítulos anteriores.

## 4.1 Generalidades

Os objectos manipulados pelas instruções destrutivas podem ter um atributo que designámos por **nome**. Informaticamente o nome costuma ser apelidado **variável**. Usaremos ambos de modo indistinto. Através da variável acedemos ao objecto.

```

1 >>> a = 5
2 >>> a
3 5
4 >>> import math
5 >>> math.pi
6 3.141592653589793
7 >>>

```

A construção dos nomes em Python obedece a regras: um nome válido tem que começar ou por uma letra ou pelo carácter `_`, podendo seguir-se depois letras, dígitos e o carácter `_`, pela ordem e em qualquer número. Python é sensível ao caso: Alma e alma são nomes diferentes e, por isso, e, geral remetem para objectos também eles distintos.

```

1 >>> Alma = 4
2 >>> alma = 3
3 >>> id(Alma)
4 4367850912
5 >>> id(alma)
6 4367850880
7 >>> a = 5
8 >>> _a = 3
9 >>> a
10 5
11 >>> _a
12 3
13 >>> carro_amarelo = 1953
14 >>> carro_amarelo
15 1953
16 >>> idade?
17 File "<stdin>", line 1
18 idade?
19 ^
20 SyntaxError: invalid syntax
21 >>> ida de
22 File "<stdin>", line 1

```

```

23 ida de
24 ^
25 SyntaxError: invalid syntax
26 >>> 53idade
27 File "<stdin>", line 1
28 53idade
29 ^
30 SyntaxError: invalid syntax
31 >>>

```

Mesmo respeitando as regras de sintaxe para os nomes, nem todos os nomes sintaticamente válidos podem ser usados.

```

1 >>> def = 23
2 File "<stdin>", line 1
3 def = 23
4 ^
5 SyntaxError: invalid syntax
6 >>>

```

A razão para o erro assinalado resulta do facto de **def** ser uma **palavra reservada** da linguagem. A listagem completa das palavras reservadas é feita na tabela 4.1.

[Palavras Reservadas](#)

Tabela 4.1: Palavras Reservadas

|        |          |         |        |          |        |       |
|--------|----------|---------|--------|----------|--------|-------|
| and    | continue | except  | global | lambda   | pass   | while |
| as     | def      | False   | if     | None     | raise  | width |
| assert | del      | finally | import | nonlocal | return | yield |
| break  | elif     | for     | in     | not      | True   |       |
| class  | else     | from    | is     | or       |        |       |

A escolha dos nomes é, em teoria livre. Mas não nos podemos esquecer que se é verdade que os programas são escritos para serem executados e resolver problemas, não é menos verdade que eles são objecto de correcções, de alterações e de reutilizações. Por isso, é fundamental que o que fazem seja claro. A escolha de nomes apropriados pode (e deve) facilitar essa tarefa. Vejamos um exemplo simples.

```

1 >>> x = 4
2 >>> y = 3.14
3 >>> z = 2 * y * x

```

```

4 >>> z
5 25.12
6 >>> raio = 4
7 >>> pi = 3.14
8 >>> perimetro = 2 * pi * raio
9 >>> perimetro
10 25.12
11 >>>

```

O leitor concordará connosco, sobre qual é o modo de designar os objectos que torna o programa mais facilmente comprehensível. Quando os nomes são muito extensos é normal tornar a sua leitura mais simples. Por exemplo, usar `peso_total` em vez de `pesototal`<sup>1</sup>

## 4.2 Atribuição

Já sabemos que uma atribuição não é mais do que a **associação** de um nome a um objecto. Os nomes são criados no momento da primeira atribuição (do nome) passando a fazer parte do espaço de nomes<sup>2</sup>. Assim depois da sessão

```

1 >>> dna='GAATCC'
2 >>> x=5.4
3 >>>

```

teremos a situação retratada na figura 4.1.

Como também sabemos a partir do momento em que associamos um nome a um objecto este pode passar a ser referenciado pelo seu nome como ilustramos na seguinte sessão.

```

1 >>> x = 5.4
2 >>> y = 3
3 >>> z = x + y
4 >>> z
5 8.4
6 >>>

```

A associação de um nome a um objecto pode ser feita de modo **indirecto**. Na realidade a sintaxe da instrução de atribuição apenas exige que à direita

---

<sup>1</sup>Existem outras maneiras de fazer. Para o caso acima há quem prefira a chamada notação camel: `pesoTotal`, embora seja uma questão de estilo pessoal o que convém é manter a coerência na notação. Para uma discussão sobre as alternativas ver ....

<sup>2</sup>O Espaço de Nomes tecnicamente não é mais do que um dicionário, forma de objectos de que nos ocuparemos mais adiante.

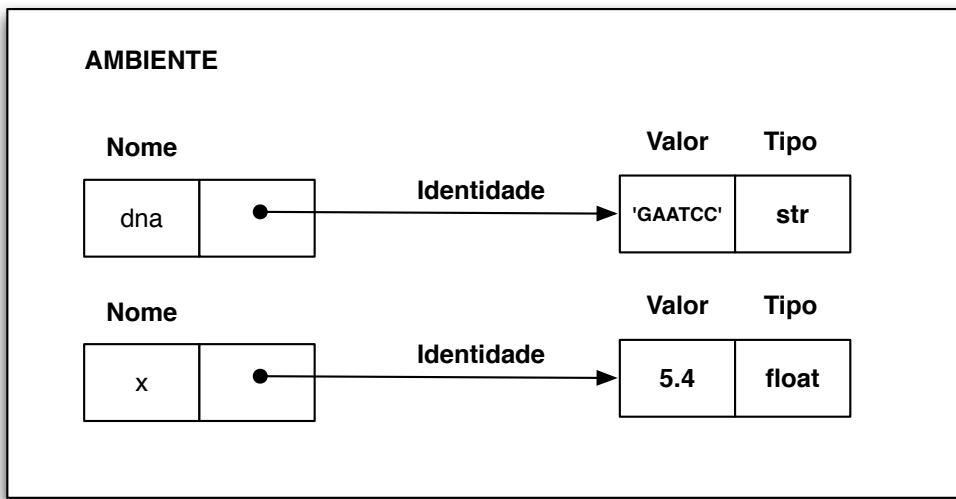


Figura 4.1: Ligação Nomes - Objectos

do sinal de igual esteja uma **expressão**. Na listagem acima, o objecto a que **z** fica associado foi encontrado depois da expressão **x + y** ter sido avaliada.

É evidente que não podemos referenciar um nome que não pertença ao espaço de nomes, o que só acontece depois da sua associação a um objecto, sob pena de gerarmos um erro.

```

1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
 2336.1.00)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
 information.
4 >>> a = 5
5 >>> a
6 5
7 >>> b
8 Traceback (most recent call last):
9 File "<stdin>", line 1, in <module>
10 NameError: name 'b' is not defined
11 >>> dir()
12 ['__builtins__', '__doc__', '__name__', '__package__', 'a']
13 >>>
```

Na sessão anterior **b** não foi associado a nenhum objecto pelo que não faz parte do espaço de nomes activo como se pode ver pelo resultado do comando

`dir()`. Diferentes nomes podem estar associados ao mesmo objecto<sup>3</sup>.

```

1 >>> a = 6
2 >>> b = a
3 >>> b
4 6
5 >>>

```

### Atribuição Implícita

Existem algumas atribuições **implícitas**. Por exemplo, quando importamos um módulo ou quando definimos uma função. Existem outras situações como as seguintes que a seu tempo se explicará.

```

1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
 2336.1.00)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
 information.
4 >>> dir()
5['__builtins__', '__doc__', '__name__', '__package__']
6>>> import math
7>>> dir()
8['__builtins__', '__doc__', '__name__', '__package__', 'math']
9>>> math
10<module 'math' from '/usr/local/pythonbrew/pythons/Python
 -3.2.3/lib/python3.2/lib-dynload/math.so'>
11>>> def duplo(x):
12... return 2 * x
13...
14>>> duplo
15<function duplo at 0x10eadda68>
16>>> dir()
17['__builtins__', '__doc__', '__name__', '__package__', 'duplo',
 'math']
18>>>

```

Uma pergunta que naturalmente se coloca é sobre o que acontece quando passamos a associar um nome já utilizado anteriormente a um objecto diferente, por exemplo fazendo `x = 10`. A resposta é simples e pode ser visualizada na figura 4.2.

---

<sup>3</sup>Também pode acontecer que um objecto não tenha nenhum nome associado. Não esquecer que o nome é apenas um atributo do objecto.

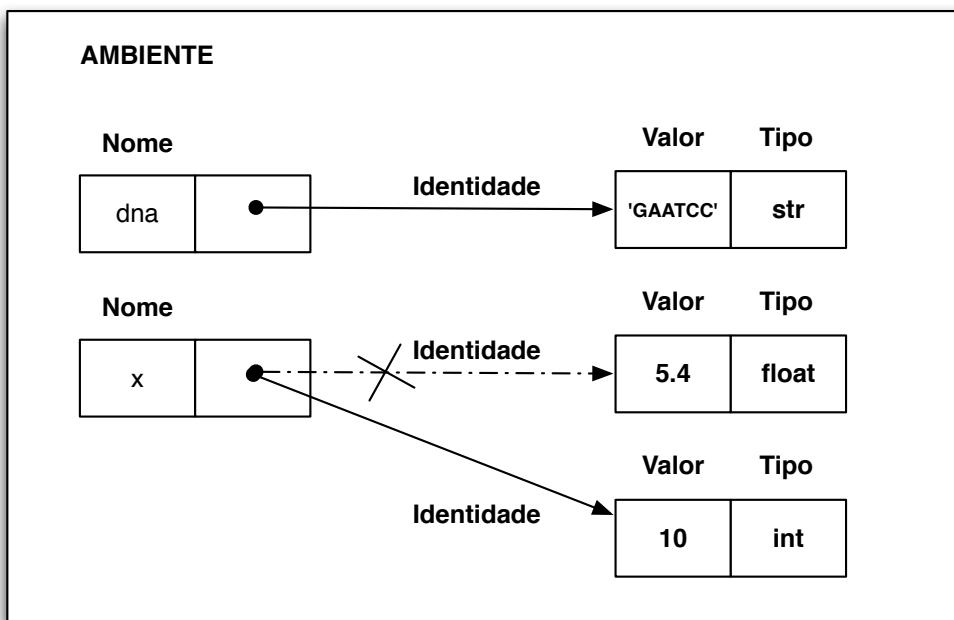


Figura 4.2: Tipagem Dinâmica

Aqui se manifesta o facto de Python ser uma linguagem **não tipada** e com **tipagem dinâmica**: o tipo não tem que ser declarado, sendo uma característica do objecto e não do nome que, num dado instante, lhe está associado: o nome **x** deixou de estar associado a um objecto do tipo **float** e ficou associado a um objecto do tipo **int**. Vejamos uma consequência subtil deste carácterística<sup>4</sup>.

Seja a sessão.

```

1 >>> x = 10
2 >>> y = x
3 >>> id(x) == id(y)
4 True
5 >>> y = y + 1
6 >>> id(x) == id(y)
7 False
8 >>> x
9 10
10 >>> y
11 11

```

<sup>4</sup>Em linguagens como C/C++ ou Java as coisas passam-se de modo diferente!

12 &gt;&gt;&gt;

A figura 4.3 ilustra o que aconteceu.

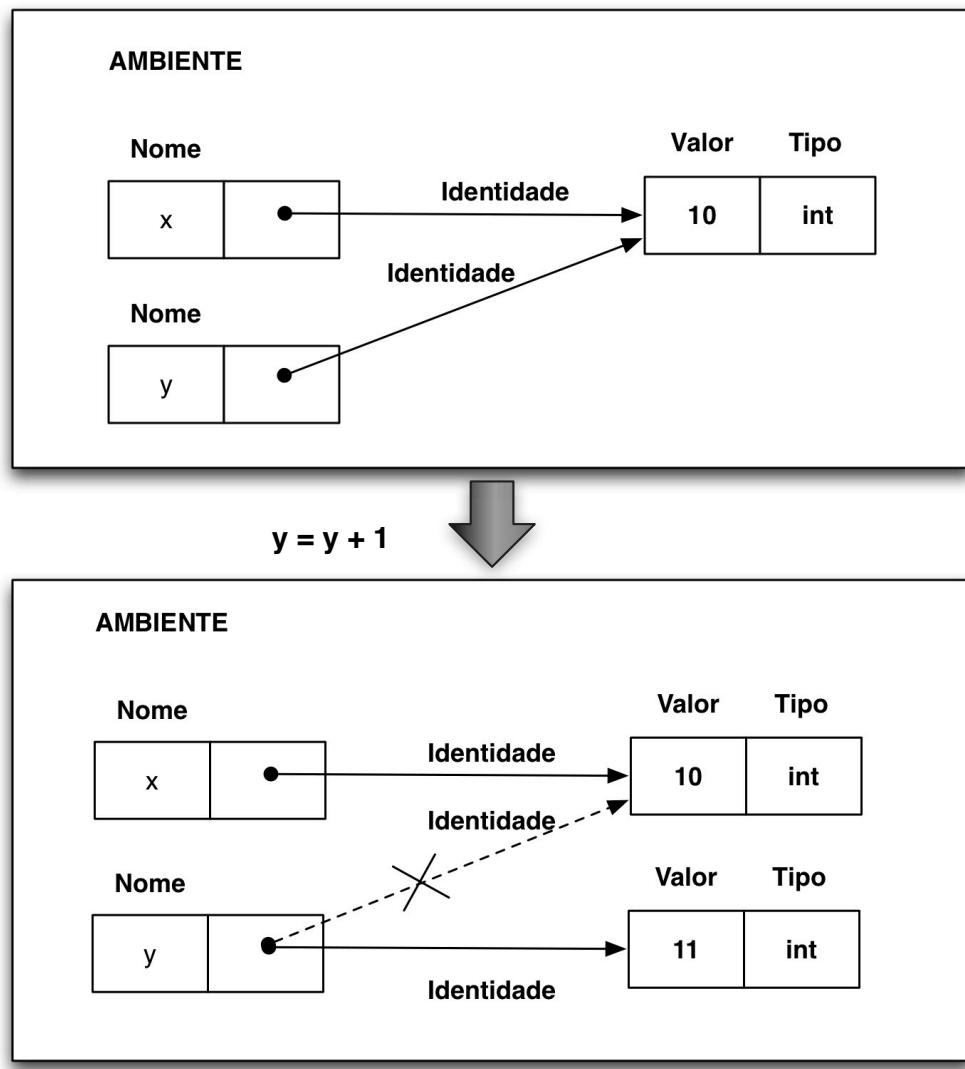


Figura 4.3: Mudança de identidade

### Outras formas de atribuição

Python permite outros modos de efectuar atribuições: em cadeia ou múltiplas. Vejamos o que significa. Comecemos pelas atribuições em cadeia.

```

1 >>> a = b = 3
2 >>> a
3 3
4 >>> b
5 3
6 >>> id(a) == id(b)
7 True
8 >>>

```

Aqui temos que o objecto 3 tem agora **dois** nomes associados. Isso é visível no facto de terem a mesma identidade. Passemos às atribuições múltiplas.

```

1 >>> x, y = 5, 7
2 >>> x
3 5
4 >>> y
5 7
6 >>> id(x)
7 4377546176
8 >>> id(y)
9 4377546240
10 >>>

```

Agora associamos de uma só vez dois objectos (5 e 7) a dois nomes (**x** e **y**). Com naturalidade as suas identidades são distintas. Esta característica permite fazer coisas interessantes:

```

1 >>> x,y=y,x
2 >>> x
3 7
4 >>> y
5 5
6 >>> id(x)
7 16790848
8 >>> id(y)
9 16790872
10 >>>

```

Com uma instrução trocamos os nomes associados aos objectos o que arrasta o seu valor, a identidade e, caso fossem de tipos diferentes, o tipo.

### 4.3 Leitura

Sabemos desde o capítulo 1 que existem diferentes maneiras de um programa comunicar com o ambiente com que interage. No caso de uma definição podemos introduzir dados através dos parâmetros formais ou da instrução `input`, e podemos comunicar resultados recorrendo à instrução `return` ou à instrução `print` (ver figura 4.4).

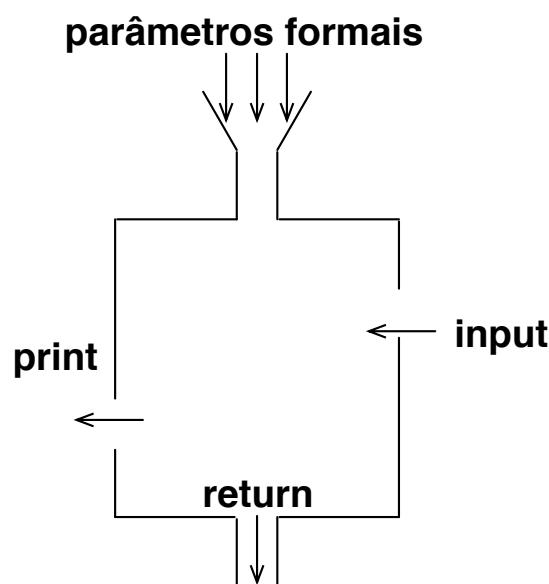


Figura 4.4: Entrada e Saída de dados

No caso da instrução de `input` existe um argumento opcional que, quando presente, é uma cadeia de caracteres que funciona como uma mensagem.

```

1 >>> idade = input('A sua idade por favor : ')
2 A sua idade por favor : 59
3 >>>

```

Precisamos ter cuidado com o uso da instrução de entrada pois o que ela devolve é **sempre** uma cadeia de caracteres. Sabendo isto é sem surpresa que observamos o que acontece na listagem seguinte.

```

1 >>> idade = input('A sua idade por favor : ')
2 A sua idade por favor : 59
3 >>> idade
4 '59'

```

```
5 >>> idade + 1
6 Traceback (most recent call last):
7 File "<stdin>", line 1, in <module>
8 TypeError: Can't convert 'int' object to str implicitly
9 >>>
```

Já sabemos que podemos usar o **construtor** dos diferentes tipos de dados para forçar um objecto a ser convertido, quando tal é possível, para outro tipo de objecto. No caso presente, recorriamo a `int`.

```
1 >>> idade = int(input('A sua idade por favor : '))
2 A sua idade por favor : 59
3 >>> idade
4 59
5 >>> idade + 1
6 60
7 >>>
```

Podemos generalizar esta abordagem usando não o construtor mas a função `eval`.

```
1 >>> idade = eval(input('A sua idade por favor : '))
2 A sua idade por favor : 59
3 >>> idade
4 59
5 >>> idade = eval(input('A sua idade por favor : '))
6 A sua idade por favor : 59.5
7 >>> idade
8 59.5
9 >>> idade + 1
10 60.5
11 >>>
```

A função `eval` tem como argumento uma cadeia de caracteres que é interpretada como sendo uma **expressão**. Daí ser possível algo como:

```
1 >>> entrada = eval(input('Introduza uma expressão >>> '))
2 Introduza uma expressão >>> 4 + 5
3 >>> entrada
4 9
5 >>> def duplo(x):
6 ... return 2 + x
7 ...
8 >>> def duplo(x):
```

```

9 ... return 2 * x
10 ...
11 >>> entrada = eval(input('Introduza uma expressão >>> '))
12 Introduza uma expressão >>> duplo(4)
13 >>> entrada
14 8

```

A chamada de uma função é sintaticamente uma expressão, explicando o que acontece na última situação.

## 4.4 Escrita

A forma mais simples de um programa comunicar com o exterior é através da função **print**. A chamada desta função é feita tendo por argumento zero ou mais expressões<sup>5</sup>.

A forma mais simples de expressões são as constantes, de qualquer tipo.

```

1 >>> print(True)
2 True
3 >>> print((1,2,3))
4 (1, 2, 3)
5 >>> print(1.23)
6 1.23
7 >>> print('Viva')
8 Viva
9 >>>

```

Podemos querer imprimir expressões mais complexas, como no exemplo seguinte.

```

1 >>> a = 2
2 >>> b = 3
3 >>> print(a, ' + ', b, ' = ', a + b)
4 2 + 3 = 5
5 >>>

```

Quando temos mais do que uma expressão a função **print** imprime os valores das expressões separadas por um caracter branco, a não ser que outro separador seja indicado explicitamente.

```

1 >>> print(a, '+', b, '=', a + b)

```

---

<sup>5</sup>Poderá parecer estranho usar a função sem qualquer argumento. No entanto, tal pode ser usado para mostrar uma linha em branco.

```

2 2 + 3 = 5
3 >>> print(a, ' + ', b, ' = ', a + b)
4 2 + 3 = 5
5 >>> print(a, ' + ', b, ' = ', a + b, sep='--')
6 2-- + --3-- = --5
7 >>>

```

O modo como a impressão termina também pode ser controlado:

```

1 >>> print(a)
2 2
3 >>> print(b)
4 3
5 >>> print(a,end=' ')
6 2>>> print(b)
7 3
8 >>> print(a,end=' ')
9 2 >>> print(a,end='**')
10 2**>>>

```

Por defeito é usado o carácter de mudança de linha `\n`. Temos assim uma sintaxe simples para a função `print`.

```

1 print([expressao,...], sep=' ', end='\n',file=sys.stdout)

```

Como se pode ver também podemos determinar onde vai ser escrito o resultado indicando qual o canal de saída que, por defeito, é o monitor (`sys.stdout`).

## Expressões de formatação

Na secção 3.4 já ilustrámos como podemos usar o operador sobrecarregado `%` para construir mensagens formatadas envolvendo vários objectos sem ter que recorrer à operação de concatenação de cadeias de caracteres.

```

1 >>> print('O primeiro nome é: %s, e o último é: %s' % ('Ernesto', 'Costa'))
2 O primeiro nome é: Ernesto, e o último é: Costa
3 >>>

```

A cadeia de caracteres à esquerda do operador `%` pode conter várias marcas de conversão que são precedidas de `%`<sup>6</sup>.

Vejamos alguns exemplos simples usados para formatar a saída de números.

---

<sup>6</sup>Admitimos que este uso múltiplo do sinal `%` é potencialmente factor de confusão.

```

1 >>> print('%d' % 12.34)
2 12
3 >>> print('%s' % 12.34)
4 12.34
5 >>> print('%e' % 12.34)
6 1.234000e+01
7 >>> print('%f' % 12.34)
8 12.340000
9 >>> print('%.0f' % 12.34)
10 12
11 >>> print('%.2f' % 12.34)
12 12.34
13 >>> print('%.5f' % 12.34)
14 12.34000
15 >>> print('%10.2f' % 12.34)
16 12.34
17 >>> print('%-6.2f -- %-6.2f' % (12.34, 12.34))
18 12.34 -- 12.34
19 >>> print('%06.2f -- %06.2f' % (12.34, 12.34))
20 012.34 -- 012.34
21 >>>

```

A sintaxe é simples: bandeiras<sup>7</sup>, o tamanho, seguido do número de casas decimais, seguido do código de conversão. Por exemplo, no caso de %-6.2f, indica tratar-se de um número em vírgula flutuante, com duas casas decimais, com tamanho mínimo de 6 posições e alinhado à esquerda. O leitor interessado em saber todas as possibilidades deve consultar o manual da linguagem.

### Método de formatação

Recentemente, a partir da versão 2.6, foi introduzido em Python um outro modo de formatar cadeias de caracteres que podemos usar com a função de `print`. Baseia-se no uso do método `format` que se aplica a um **modelo** de cadeia de caracteres que é instanciada por recurso a argumentos por **posição** ou por **nome**. A listagem seguinte ilustra a ideia.

```

1 >>> modelo_1 = '{0}, {1} e {2}'
2 >>> texto_1 = modelo_1.format('cama', 'mesa', 'roupa lavada')
3 >>> print(texto_1)
4 cama, mesa e roupa lavada

```

---

<sup>7</sup>do inglês *flags*.

```

5 >>> texto_2 = modelo_1.format('roupa lavada', 'mesa', 'cama')
6 >>> print(texto_2)
7 roupa lavada, mesa e cama
8 >>> modelo_2 = '{dorme}, {come} e {veste}'
9 >>> texto_21 = modelo_2.format(come='mesa',veste='roupa lavada'
10 ,dorme='cama')
11 >>> print(texto_21)
12 cama, mesa e roupa lavada
13 >>> 'Eu sou {0}!'.format('toto')
14 'Eu sou toto!'
15 >>> print('Eu sou {0}!'.format('toto'))
16 Eu sou toto!
17 >>>

```

Tratando-se de um método sobre uma cadeia de caracteres usamos a notação por ponto já referida na secção 3.4. Notar o uso de chavetas para referenciar os objectos. O método **format** devolve uma cadeia de caracteres, que sendo um objecto imutável que ou é enviado para o exterior ou é associado a um nome. Caso contrário perde-se.

Podemos usar marcas mais sofisticadas, como no caso do recurso a expressões de formatação, como se ilustra no exemplo seguinte.

```

1 >>> print('{0:6.2f}'.format(12.34))
2 12.34
3 >>> print('{0:<15.2f}{1:<15.2f}'.format(12.34, 12.34))
4 12.34 12.34
5 >>> print('{0:<15.2f}{1: >15.2f}'.format(12.34, 12.34))
6 12.34 12.34
7 >>>

```

Os dois modos de proceder à formatação das cadeias de caracteres podem actualmente ser usados. Cada um deles tem vantagens e inconvenientes, embora a versão por recurso ao método **format** seja considerada mais pitónica, havendo por isso a hipótese de, no futuro, a mais antiga ser descontinuada.

## Sumário

Neste capítulo retomámos a aprofundámos as noções de atribuição, de entrada e de saída de dados, enquadrando-as no conceito de instruções destrutivas.

## Testes os seus conhecimentos

Verifique se domina os conceitos listados e se sabe responder às questões colocadas.

- Qual o significado de estado? E de computação?
- Que palavras reservadas existem em Python ? Qual a sua função e importância?
- O que significa dizer que Python é uma linguagem não tipada e que o tipo associado a um objecto é determinado dinamicamente?
- Que consequências práticas existem da tipagem ser dinâmica?
- Como define o conceito de atribuição?
- Que formas de atribuição conhece?
- De que maneiras podem introduzir dados num programa? E de que maneiras pode retirar resultados?
- Como funciona a instrução `input`? Funciona com qualquer tipo de objecto?
- Qual a diferença entre formar a saída por recurso a uma expressão ou ao método `format`?

## Exercícios

**Exercício 4.1 F** Recorrendo ao interpretador, identifique quais dos seguintes nomes são válidos para nomes de objectos:

- abc
- 5peso
- \_valor
- Ernesto Costa
- ABC
- with
- peso\$

- minha\_altura
- class
- nome\_ALUNO
- a(b)
- \_\_\_\_<sup>1</sup>
- \_\_x\_\_
- import\_from
- area-rect

Justifique os casos em que os nomes não são válidos.

#### **Exercício 4.2 MF**

Escreva um programa que lhe permita imprimir os caracteres gregos  $\alpha, \beta, \gamma$ . **Sugestão:** obtenha os códigos **unicode** de cada carácter.

#### **Exercício 4.3 MF**

Experimente fazer o seguinte:

```

1 >>> x = 5
2 >>> y = 5
3 >>>

```

Inspeccione os objectos de nome  $x$  e  $y$ , isto é, determine a sua identidade, valor e tipo. Que conclusões pode tirar?

#### **Exercício 4.4 MF** Experimente fazer o seguinte:

```

1 >>> a = 10
2 >>> b = a
3 >>>

```

Inspeccione os objectos e tire conclusões.

#### **Exercício 4.5 MF**

Simule a seguinte sessão no interpretador:

```

1 >>> x = 5
2 >>> x = x + 1
3 >>>

```

Inspecione o objecto *x* após cada passo. Que conclusões pode tirar?

### Exercício 4.6 MF

Considere a seguinte sessão no interpretador:

```

1 >>> a = 10
2 >>> b = a
3 >>> a = 11

```

Como explica os resultados da inspecção?

**Exercício 4.7 M** desenhar diagramas do ambiente depois de um conjunto de atribuições

**Exercício 4.8 M** Explique o que acontece de modo claro, sintético e **rigoroso** quando executa o comando:

```

1 >>> cad = 'a' * 3

```

A sua explicação deve incluir a visualização do **espaço de nomes** e do **espaço de objectos** depois de executado o comando indicado.

**Exercício 4.9 F** Usando a instrução **print** e o método **format** diga como podia obter o efeito da listagem seguinte:

```

1 Bem vindo a IPRP
2 Bem vindo a IPRP
3 Bem vindo a IPRP e ao DEIUC

```

**Exercício 4.10 M** Desenvolva um programa que lhe permita imprimir a seguinte tabela.

|   | Número | Quadrado |
|---|--------|----------|
| 2 | 1      | 1        |
| 3 | 2      | 4        |
| 4 | 3      | 9        |
| 5 | 4      | 16       |
| 6 | 5      | 25       |

Caso pretenda que a tabela possa ter um número variável de linhas em que medida precisa, ou não, de alterar a sua solução? Se a resposta for afirmativa apresente o respectivo programa.

**Exercício 4.11 M** Desenvolva um programa que dado um número inteiro menor ou igual a dez imprime a tabela da respectiva tabuada. A listagem abaixo ilustra para o caso do número 7.

```

1 Tabuada do número 7
2 -----
3 7 x 1 = 7
4 7 x 2 = 14
5 7 x 3 = 21
6 7 x 4 = 28
7 7 x 5 = 35
8 7 x 6 = 42
9 7 x 7 = 49
10 7 x 8 = 56
11 7 x 9 = 63
12 7 x 10 = 70

```

**Exercício 4.12 M** Desenvolva um programa que dado um nome por extenso constrói o respectivo acrónimo. A listagem abaixo ilustra o pretendido.

```

1 >>> print(acronimo('Random Access Memory'))
2 RAM

```

**Exercício 4.13 F** Na descolagem de um avião a relação entre a aceleração,  $a$ , a velocidade,  $v$ , determina o comprimento mínimo da pista,  $c$ , para tudo correr bem, de acordo com a fórmula:

$$c = \frac{v^2}{2 \times a}$$

Escreva um programa que pede os dados ao utilizador e imprime uma mensagem com o resultado. Uma hipótese de interacção é dada na listagem seguinte:

```

1 Velocidade de descolagem (m/s): 50
2 Aceleração para descolagem (m/s?): 4.5
3 Para a velocidade 50.00 e aceleração 4.50 o comprimento mínimo
da pista é: 277.78.

```

**Exercício 4.14 F** A energia necessária para elevar a temperatura de uma dada massa de água de uma temperatura inicial até uma temperatura final é dada pela fórmula:

$$E = m \times (t_f - t_i) \times 4184$$

*E* é a energy (em Joules),  $t_i$  e  $t_f$  as temperaturas inicial e final (em graus Celcius) e  $m$  a massa (em quilogramas). Escreva um programa que pede os dados ao utilizador e imprime uma mensagem com o resultado. Uma hipótese de interacção é dada na listagem seguinte:

- 1 Temperatura inicial (Celcius): 10
- 2 Temperatura final (Celcius): 30
- 3 Quantidade de água (Quilogramas): 25
- 4 Para a massa de água 25.00, temperatura inicial 10.00 e temperatura final 30.00 a energia necessária é: 2092000.00 Joules.

**Exercício 4.15 F** A temperatura exterior depende de vários factores. Um das fórmulas de cálculo faz intervir a velocidade do vento:

$$t_v = 35.4 + 0.6215 \times t - 35.75 \times v^0.16 + 0.4275 \times t \times v^0.16$$

A temperatura é medida em graus Fahrenheit e a velocidade do vento em milhas por hora. Escreva um programa que pede os dados ao utilizador e imprime uma mensagem com o resultado. Uma hipótese de interacção é dada na listagem seguinte:

- 1 Velocidade do vento (milhas/hora): 5
- 2 Temperatura (Fashrenheit [-58, 41]): 10
- 3 Para a velocidade do vento 5.00 e temperatura exterior 10.00 a temperatura é sentida como: 1.24.

**Exercício 4.16** Desenvolva um programa que lhe permita imprimir os elementos de uma matriz antecedidos pela sua posição na matriz. A listagem abaixo exemplifica para a matriz  $[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]$ .

- 1 (0,0): 1 (0,1): 2 (0,2): 3
- 2 (1,0): 4 (1,1): 5 (1,2): 6
- 3 (2,0): 7 (2,1): 8 (2,2): 9
- 4 (3,0): 10 (3,1): 11 (3,2): 12

**Exercício 4.17 M** Pretende-se desenvolver um programa que pergunte ao utilizador a frequência de nascimentos, de mortes e de emigrantes em minutos. Depois, conhecida a população inicial deve calcular a nova população no final do ano. Assuma que o ano tem 365 dias. A listagem mostra uma interação possível.

```

1 Frequência de nascimentos (minutos): 20
2 Frequência de falecimentos (minutos): 15
3 Frequência de emigração (minutos): 10
4 Resumo dos dados:
5 -----
6 Frequência de nascimentos: 20
7 Frequência de mortes: 15
8 Frequência de emigrantes: 10
9 População Inicial: 10000000
10 Estimativa:
11 -----
12 A população ao fim de um ano: 9938680

```

Se quiser estimar o resultado ao fim de vários anos como modificaria a sua solução?

**Exercício 4.18 F** Considere a seguinte definição:

```

1 def add2me(x):
2 return x + x

```

Indique, **justificando**, quais os resultados esperados ao executar os comandos:

```

1 >>> add2me(23.4)
2 ???
3 >>> add2me('toto')
4 ???

```

**Exercício 4.19 M** Explique de modo claro, sintético e **rigoroso** o que aconteceu na sessão seguinte:

```

1 >>> def prod(x,y):
2 ... return x * y
3 ...
4 >>> a = 5

```

```
5 >>> print prod(a,3)
6 15
7 >>> a
8 5
9 >>> x
10 Traceback (most recent call last):
11 File "<string>", line 1, in <fragment>
12 NameError: name 'x' is not defined
13 >>>
```

# Instruções de Controlo

## Objectivos

- ✓ Tomar contacto com as instruções condicionais
- ✓ Tomar contacto com as instruções de repetição (ciclos)
- ✓ Exercitar estes conceitos por meio de exemplos simples

### 5.1 Introdução

Na nossa vida todos estamos habituados a tomar decisões: jantar fora e depois ir ao cinema, comprar um carro novo ou um carro em segunda mão, programar várias sessões de ginástica, são alguns exemplos simples do nosso dia a dia. As nossas escolhas têm por base a avaliação que fazemos das situações concretas e seus condicionalismos: se o jantar terminar cedo vamos seguramente ao cinema, se acabámos de receber uma herança compramos um carro novo, vamos uma vez por semana ao ginásio. Em programação, a tomada de decisão obriga a fazer testes booleanos e actuar em função do seu resultado, sendo uma condição fundamental para podermos resolver problemas com alguma complexidade. No capítulo 4 vimos a existência de instruções destrutivas, isto é, instruções que alteram a associação entre nomes e objectos ou instruções que alteram o valor dos objectos. Chegou a vez de estudarmos instruções não destrutivas que permitem determinar qual a próxima instrução a ser executada. São designadas por **instruções de controlo**. Existem três tipos:

Sequências,  
Condicionais e  
Ciclos

- sequências

- condicionais
- ciclos

### Indentação e Blocos

#### Indentação e Blocos

Antes de detalharmos a sua sintaxe e o seu funcionamento , interessa desde já referir que, em Python, o código está organizado em **blocos**, isto é, grupo de instruções que, por exemplo, podem ser executadas se uma dada condição for verdadeira ou executadas repetidas vezes. Podemos ter blocos dentro de blocos. A marca de início de um boco são os dois pontos (:). Para significar que um conjunto de instruções pertence ao mesmo bloco usa-se o mecanismo de **indentação** de código: as instruções estão alinhadas verticalmente graças ao uso de espaços ou tabulações<sup>1</sup>. A figura 5.1 ilustra essa situação e a listagem 5.1 mostra um exemplo concreto.

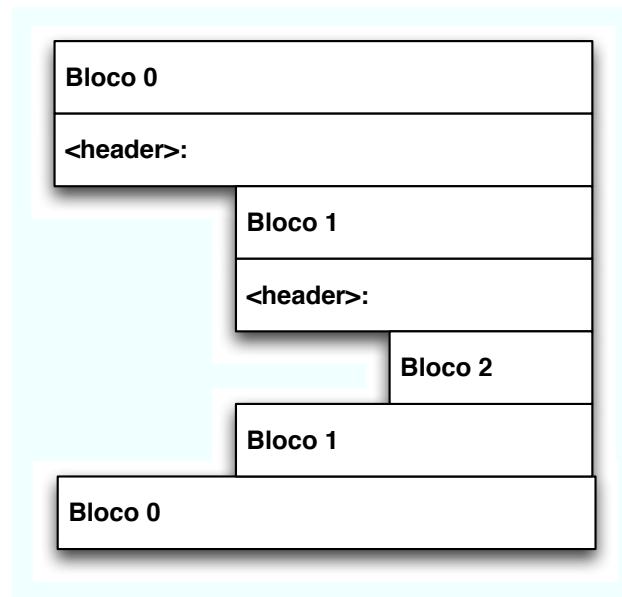


Figura 5.1: Os blocos de um programa

```

1 x = 1
2 if x == 1:
3 y = 2
4 if y == 2:
5 x = 2

```

<sup>1</sup>Não se podem misturar espaços e tabulações.

```

6 print('Bloco 2: x = %d, y = %d' % (x,y))
7 y = 1
8 print('Bloco 1: x = %d, y = %d' % (x,y))
9 x = 1
10 print('Bloco 0: x = %d, y = %d' % (x,y))
11
12 # Resultado da execução
13 Bloco 2: x= 2, y= 2
14 Bloco 1: x= 2, y= 1
15 Bloco 0: x= 1, y= 1

```

Listagem 5.1: Blocos

## Expressões Booleanas

O tipo de dados **boolean** foi introduzido anteriormente (ver 3.3). Ficámos a saber que só tem dois objectos, denotados por **True** e por **False**, e várias operadores relacionais e operadores lógicos. As expressões booleanas são construídas a partir dos objectos e dessas operações. Um exemplo simples do seu uso é dado pela listagem 5.2.

```

1 >>> x = 7
2 >>> y = 20
3 >>> z = w =10
4 >>> x < y
5 True
6 >>> z <= w
7 True
8 >>> y == z
9 False
10 >>> x != y
11 True
12 >>> 'A' < 'Z'
13 True
14 >>> 'a' < 'A'
15 False
16 >>> 3.4 < 4
17 True
18 >>> (x < y) and (w == 10)
19 True
20 >>> not ('a' < 'A')
21 True

```

```

22 >>> ('a' < 'A') or (x !=y)
23 True
24 >>> not ('a' < 'A') or (x !=y)
25 True
26 >>> not (('a' < 'A') or (x !=y))
27 False
28 >>> (x < y) or (x/0)
29 True
30 >>>

```

Listagem 5.2: Operadores relacionais

Observe-se a última situação na qual ocorre uma divisão por zero e que não é detectada. Isso acontece devido ao modo como se determina o valor lógico final da expressão. Num **and** a avaliação termina mal apareça uma situação falsa, enquanto num **or** a avaliação cessa mal apareça um resultado parcial verdadeiro.

Existem outros operadores que quando são aplicados aos objectos apresentam um resultado do tipo **boolean**, como o **in** que permite responder se um elemento pertence a uma sequência ou o **is** que permite saber se dois nomes estão associados ao mesmo objecto (ver listagem 5.3).

```

1 >>> 'a' in 'bcad'
2 True
3 >>> 'ola' in 'bolacha'
4 True
5 >>> not ('pe' in 'sope')
6 False
7 >>> z = w =10
8 >>> x = 5
9 >>>

```

Listagem 5.3: Outros operadores

Como acabámos de ver o resultado de uma expressão booleana é um de dois objectos: **True** ou **False**. No entanto, em função do contexto podemos usar outros objectos para significar **False**, como se indica na tabela 5.1<sup>2</sup>.

Por oposição, tudo o que não denotar falso denota a condição verdadeira. Dito de outro modo: em função do contexto qualquer objecto em Python pode ser interpretado como um valor de verdade. Embora pareça estranho e eventualmente confuso, esta situação pode ser bastante vantajosa como veremos através de diferentes exemplos.

---

<sup>2</sup>Listas e dicionários serão tratados mais à frente.

Tabela 5.1: Representações de Falso

| Objecto           | Descrição                  |
|-------------------|----------------------------|
| <code>None</code> | Nada                       |
| <code>0</code>    | Zero                       |
| <code>"</code>    | Cadeia de Caracteres vazia |
| <code>()</code>   | Tuplo vazio                |
| <code>[]</code>   | Lista vazia                |
| <code>{}</code>   | Dicionário vazio           |

## 5.2 Sequências

A forma mais simples de organizar a ordem de execução das instruções de um programa é através da sua sequenciação (ver figura 5.2). Executamos primeiro **A**, depois **B** e, finalmente, **C**.

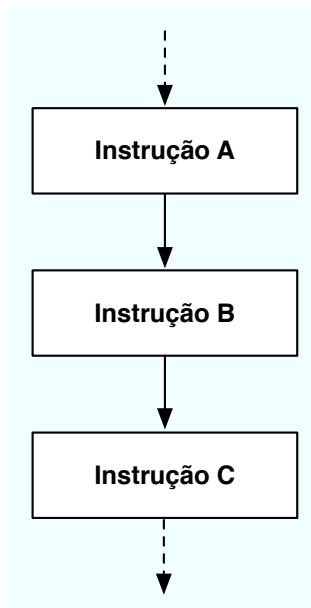


Figura 5.2: Sequência

Para impor essa ordem recorremos a **delimitadores**. Estes podem ser implícitos ou explícitos como se mostra na listagem 5.4.

<sup>1</sup> >>> **import** math

```

2 >>> a=5; b=7
3 >>> L=[1,2,
4 ... 3,4]
5 >>> L
6 [1, 2, 3, 4]
7 >>> fich=open('/tempo/\n
8 ... cod.txt')
9 >>> fich
10 <open file '/tempo/cod.txt', mode 'r' at 0x5b4a0>
11 >>>

```

Listagem 5.4: Sequência

Na primeira situação (linha 1) usamos a tecla *<enter>*. No segundo caso (linha 2) a separação é feita pelo ponto e vírgula. Na terceira situação trata-se de uma estrutura, neste caso uma lista, que tem uma marca de início (`[`) e de fim (`]`). Finalmente a última situação mostra o uso do *backslash* introduzido numa cadeia de caracteres.

Num programa qualquer também aparece a estrutura sequência, como se mostra na listagem seguinte.

```

1 def soma_produto(x,y):
2 soma = x+y
3 produto = x*y
4 print(soma,produto)

```

Aqui temos três instruções executadas em sequência, duas de atribuição (`soma` e `produto`) e uma de impressão. Notar o alinhamento das instruções indicando que fazem parte do mesmo bloco.

### 5.3 Condicionais

Devido à sua natureza, os programas escritos com apenas a estrutura de controlo sequência são muito pobres e resolvem apenas problemas muito simples. A introdução das condicionais, com a sua possibilidade de se fazerem escóllhas, vai permitir a escrita de programas mais interessantes. As condicionais dividem-se em três categorias:

- simples (uma via)
- normal (duas vias)
- geral (várias vias)

**Condicional Simples** A situação mais simples corresponde ao caso em que num teste com duas alternativas, numa delas executamos acções e na outra nada fazemos. A figura 5.3 ilustra a situação.

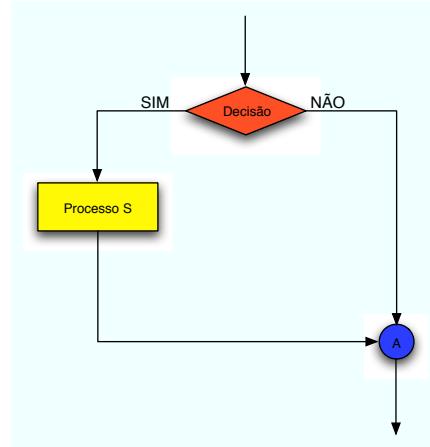


Figura 5.3: A condicional if-then

Em pseudo código temos:

```

1 if <condição>:
2 <corpo>

```

Se uma dada condição booleana for satisfeita executamos o corpo, caso contrário não fazemos nada.

Um exemplo trivial:

```

1 def exe1():
2 t= input('temperatura sff:')
3 if t > 38.5:
4 print('ATENÇÃO: febre!')

```

Outro exemplo:

```

1 def exe2():
2 nome=raw_input(u'Qual é o seu nome?')
3 if nome.endswith('Costa'):
4 print('Olá Senhor Costa')

```

Ainda outro:

```

1 def exe3(n):
2 if (n % 2) == 0:
3 print("O número %d é par" % n)

```

E mais outro para concluir.

```

1 def exe4():
2 seq='ATGAnnTAG'
3 if 'n' in seq:
4 conta= seq.count('n')
5 print('A sequência %s tem %d bases não definidas' % (seq,
6 conta))

```

Testando os programas.

```

1 ATENÇÃO: febre!
2 Olá Senhor Costa
3 O número 4 é par
4 A sequência ATGAnnTAG tem 2 bases não definidas

```

**Condicional Normal** Vejamos o caso mais frequente de termos duas alternativas mas ambas com acções a realizar. A figura 5.4 ilustra a ideia.

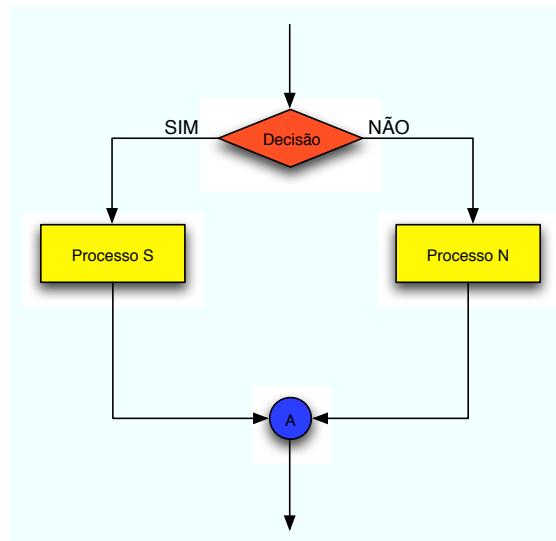


Figura 5.4: A condicional if-then-else

Em termos abstractos temos a seguinte sintaxe:

```

1 if <condição>:
2 <corpo1>
3 else:
4 <corpo2>

```

Se a condição booleana for verdadeira executamos o corpo1, caso contrário executamos o corpo2. Alguns exemplos simples aparecem na listagem ??.

```

1 # --- par
2 def exe5(n):
3 if (n % 2) == 0:
4 print("O número %d é par" % n)
5 else:
6 print("O número %d é ímpar" % n)
7 # -- descohecido
8 def exe6():
9 nome=raw_input(u'Qual é o seu nome?')
10 if nome.endswith('Costa'):
11 print ('Olá Senhor Costa')
12 else:
13 print('Olá desconhecido')
14
15 # -- primer
16 def exe7():
17 primer='AACTAACCACTTCGGAATCTAGGACGGGGAG\
18 CGTTTACATGACGCCGTGGACCAAAGATTAGGCAATCGTCA\
19 GTCGCTGCGCCAAGAACACGGAGAGTACCTCATGCGTGAT\
20 CTTTCATAGAGCTTGAGAACTGCTGACCTAGGGTTT'
21 comp_primer=len(primer)
22 percent_GC= float(primer.count('G') + primer.count('C'))/
23 comp_primer
24 if percent_GC > 0.50:
25 if comp_primer > 20:
26 programaPCR=1
27 else:
28 programa_PCR=2

```

**Condicional Geral** Vamos agora generalizar a condicional permitindo um número variável de vias de decisão (ver figura 5.5).

Genericamente a sintaxe é a seguinte:

```

1 if <condição1>:
2 <instruções1>
3 elif <condição2>:
4 <instruções2>
5 elif <condição3>:
6 <instruções3>
7 ...
8 else:

```

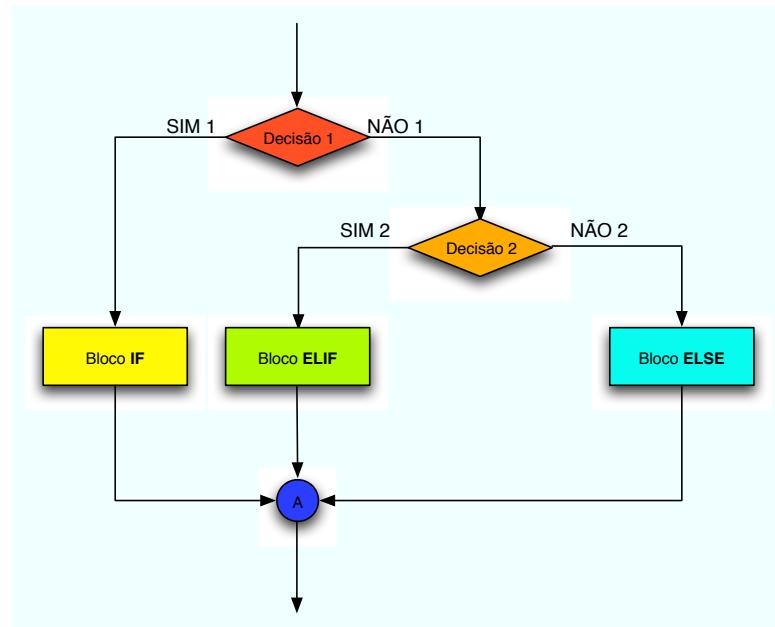


Figura 5.5: A condicional if-elif-else

## 9 &lt;instruções&gt;

A palavra reservada **elif** resulta da compressão de **else** e de **if**. A semântica é simples: as várias alternativas (i.e. as que envolvem testes) vão sendo percorridas por ordem descendente e a primeira condição que for verdadeira desencadeia a execução das instruções associadas. Caso nenhuma seja verdadeira serão activadas as instruções do ramo **else**.

```

1 def exe8():
2 seq='vATGCAnATG'
3 base=seq[0]
4 if base in 'ATGC':
5 print 'Nucleótido exacto'
6 elif base in 'dbhkmnrsuvwxy':
7 print 'Nucleótido ambíguo'
8 else:
9 print 'Não é um nucleótido'

```

Listagem 5.5: Condicional Geral: exemplo

### Exemplo clássico: raízes de um Polinómio

Consideremos o problema de calcular as raízes de uma equação do segundo grau:

$$a \times x^2 + b \times x + c = 0$$

Todos nos lembramos da fórmula resolvente:

$$r_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 \times a \times c}}{2 \times a}$$

onde  $(b^2 - 4 \times a \times c)$  é o discriminante. Comecemos por pensar na situação simples em que criamos um programa que usa directamente a fórmula resolvente (ver listagem 5.6).

```

1 import math
2
3 def main1():
4 """ Calculo das raízes reais de um polinómio."""
5 a,b,c = input("Os coeficientes sff (a,b,c):\t")
6 r1,r2=raizes1(a,b,c)
7 print ("As raízes do polinómio de coeficientes\
8 a=%d b=%d c= %d são r1=%3.2f r2=%3.2f" % (a,b,c,r1,r2))
9
10
11 def raizes1(a,b,c):
12 """ Calcula raízes reais."""
13 discriminante= pow(b,2) - 4 * a * c
14 raiz_discrim = math.sqrt(discriminante)
15 raiz1=float((-b + raiz_discrim)) / (2 * a)
16 raiz2=float((-b - raiz_discrim)) / (2 * a)
17 return raiz1,raiz2

```

Listagem 5.6: Raízes: solução trivial

O programa está decomposto em duas partes: uma, que pede os coeficientes, calcula as raízes e imprime o resultado, a segunda que calcula efectivamente a solução usando a fórmula. Claro que esta solução só funciona para o caso das raízes serem reais. Mas sabemos que podemos ter raízes múltiplas. Para ter esse facto em atenção basta alterar o programa principal introduzindo um teste (ver listagem 5.7).

```

1 def main2():
2 """ Calculo das raízes reais de um polinómio."""

```

```

3 a,b,c = input("Os coeficientes sff (a,b,c):\t")
4 r1,r2=raizes2(a,b,c)
5 if r1 == r2:
6 print("Raízes múltiplas!!")
7 print ("A raíze múltipla do polinómio de coeficientes\
8 a=%d b=%d c= %d é r=%3.2f " % (a,b,c,r1))
9 else:
10 print("As raízes do polinómio de coeficientes\
11 a=%d b=%d c= %d são r1=%3.2f r2=%3.2f" % (a,b,c,r1,r2))

```

Listagem 5.7: Raízes múltiplas

É melhor protegermos o programa para o caso de não existirem raízes reais. Isso faz-se uma vez mais com um teste para tentar saber o sinal do discriminante. Isso será aproveitado para uma versão melhorada (ver listagem 5.8).

```

1 def main3():
2 """ Calculo das raízes reais de um polinómio."""
3 a,b,c = input("Os coeficientes sff (a,b,c):\t")
4 r1,r2=raizes3(a,b,c)
5 if r1 == r2 == None:
6 print ("Não tem raízes reais!")
7 elif r1 == r2:
8 print("O polinómio de coeficientes\
9 a=%d b=%d c= %d tem uma raiz múltipla r=%3.2f" % (a,b,c,
10 r1))
11 else:
12 print ("As raízes do polinómio de coeficientes\
13 a=%d b=%d c= %d são r1=%3.2f r2=%3.2f" % (a,b,c,r1,r2))
14
15 def raizes3(a,b,c):
16 """ Calcula raízes reais."""
17 discriminante= pow(b,2) - 4 * a * c
18 if discriminante < 0:
19 return None,None
20 elif discriminante == 0:
21 raiz1 = raiz2 = float(-b)/ (2 * a)
22 return raiz1, raiz2
23 else:
24 raiz_discrim = math.sqrt(discriminante)
25 raiz1=float((-b + raiz_discrim)) / (2 * a)
26 raiz2=float((-b - raiz_discrim)) / (2 * a)

```

```
26 return raiz1,raiz2
```

Listagem 5.8: Testa raízes reais

Alterámos as duas definições e o programa ficou um pouco melhor. Mas admitamos que alguém insiste em que o programa deve também calcular as raízes mesmo quando são complexas? Bom, para tal basta socorrermo-nos do módulo **cmath** (ver listagem 5.9).

```
1 import cmath
2
3 def main4():
4 """ Calculo das raízes de um polinómio."""
5 a,b,c = input("Os coeficientes sff (a,b,c):\t")
6 r1,r2=raizes4(a,b,c)
7 if r1 == r2:
8 print("O polinómio de coeficientes\
9 a=%d b=%d c= %d tem uma raiz múltipla r=%3.2f" % (a,b,c,
10 r1))
11 elif isinstance(r1,float):
12 print("As raízes do polinómio de coeficientes\
13 a=%d b=%d c= %d são r1=%3.2f r2=%3.2f" % (a,b,c,r1,r2))
14 else:
15 print("As raízes do polinómio de coeficientes\
16 a=%d b=%d c= %d são r1=%s r2=%s" % (a,b,c,r1,r2))
17
18
19 def raizes4(a,b,c):
20 """ Calcula raízes."""
21 discriminante= pow(b,2) - 4 * a * c
22 if discriminante > 0:
23 raiz_discrim = math.sqrt(discriminante)
24 raiz1=float((-b + raiz_discrim)) / (2 * a)
25 raiz2=float((-b - raiz_discrim)) / (2 * a)
26 return raiz1,raiz2
27 elif discriminante == 0:
28 raiz1 = raiz2 = float(-b)/ (2 * a)
29 return raiz1, raiz2
30 else:
31 raiz_discrim = cmath.sqrt(discriminante)
32 raiz1=(-b + raiz_discrim) / (2 * a)
33 raiz2=(-b - raiz_discrim) / (2 * a)
```

```
34 return raiz1,raiz2
```

Listagem 5.9: Raízes: solução geral

Atente-se ao modo como separamos a impressão das raízes reais das raízes complexas.

## 5.4 Ciclos

Com frequência ficamos perante um problema que se traduz pela repetição de uma dada tarefa. Por exemplo, podemos querer transferir todos os meses uma certa quantia de dinheiro entre duas contas. Em termos informáticos podemos querer imprimir números entre dois limites conhecidos, ou desenhar um segmento de recta várias vezes, embora em localizações diferentes (ver listagem 5.10). Para este tipo de problemas as linguagens de programação disponibilizam estruturas de controlo repetitivas, também conhecidas por ciclos.

A listagem 5.10 mostra um exemplo gráfico simples em que se pretende repetir quatro vezes o mesmo par de comandos.

```
1 from turtle import *
2
3 def exec1(lado,angulo):
4 pd()
5 fd(lado)
6 rt(angulo)
7 fd(lado)
8 rt(angulo)
9 fd(lado)
10 rt(angulo)
11 fd(lado)
12 rt(angulo)
13 ht()
```

Listagem 5.10: Desenhar por repetição

Claro que podemos ter situações mais complexas, quando os comandos a repetir envolvem algumas computações, como no exemplo seguinte.

```
1 def exec2(lado,angulo):
2 pd()
3 fd(lado)
4 lado*= 1.5
5 rt(angulo)
```

```

6 angulo += 30
7 fd(lado)
8 lado*= 1.5
9 rt(angulo)
10 angulo += 30
11 fd(lado)
12 lado*= 1.5
13 rt(angulo)
14 angulo += 30
15 fd(lado)
16 lado*= 1.5
17 rt(angulo)
18 angulo += 30
19 ht()

```

As linguagens de programação incluem instruções de controlo de tipo repetitivo que permitem diminuir o tamanho do código, tornar os programas mais inteligíveis e ainda resolver situações em que o número de repetições não é fixo à partida. Como vamos ver, os ciclos repetitivos desdobram-se em duas grandes famílias: **for** e **while**.

### for

Em aplicações simples os ciclos **for** são executados um número fixo e pré-definido de vezes. Por exemplo, o exemplo da listagem 5.10 pode ser reescrito conforme mostramos na listagem 5.11. A adaptação para o segundo exemplo acima é trivial.

```

1 def exec2(lado,angulo):
2 pd()
3 for i in range(4):
4 fd(lado)
5 rt(angulo)
6 ht()

```

Listagem 5.11: De novo o quadrado

Esta construção pode ser generalizada de modo que o número de vezes que se repete a acção seja comunicada ao programa no acto de o chamar. Igualmente podemos usar valores diferentes do lado e do ângulo (ver listagem 5.12..

```

1 def exec2(lado,angulo,vezes):
2 pd()

```

```

3 for i in range(vezes):
4 fd(lado)
5 rt(angulo)
6 ht()

```

Listagem 5.12: Formas simples

Deixamos ao leitor o cuidado de visualizar as formas que se obtém para diferentes valores dos três parâmetros. A construção repetitiva **for** obedece assim ao seguinte padrão sintático:

```

1 for <nome> in <iterador>:
2 <instrução>

```

O iterador é qualquer objecto composto. Por exemplo, pode ser uma cadeia de caracteres ou um tuplo<sup>3</sup>. Antes de executar a <instrução> o <nome> assume o primeiro elemento do iterador. Depois de executada a <instrução> toma o seu segundo valor, e assim sucessivamente. A figura 5.6 ilustra o conceito.

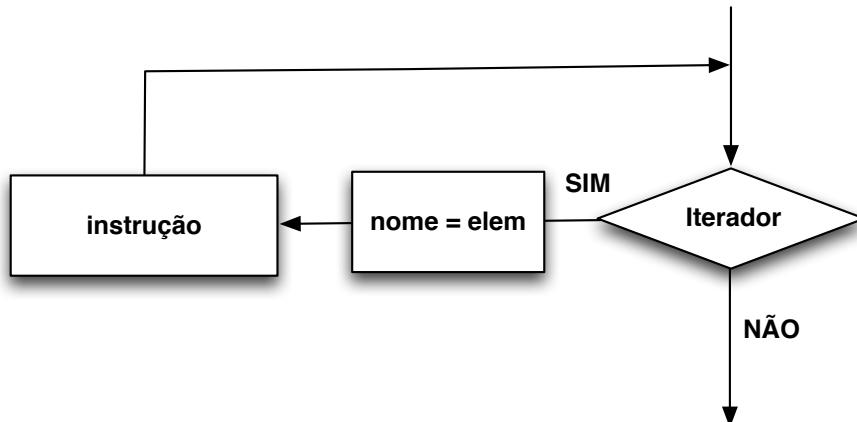


Figura 5.6: Ciclo for

Um exemplo simples do uso do ciclo **for** é ilustrado na listagem 5.13.

```

1 # No módulo
2 def exec4(seq):
3 for ch in seq:

```

<sup>3</sup>Com a descoberta de novos objectos nos capítulos seguintes veremos que há uma grande variedade que pode assumir o papel de iterador.

```

4 print(ch)
5
6 # no interpretador
7 >>> exec4('abc')
8 a
9 b
10 c
11 >>>

```

Listagem 5.13: Ciclo for

Este modo de percorrer o iterador, pelo seu **conteúdo**, não é o único. Podemos também usar um iterador formado pelos **índices** dos elementos de um objecto, como se pode ver na listagem abaixo.

```

1 def exec5(seq):
2 for i in range(len(seq)):
3 print(seq[i])

```

## while

Os ciclos **while** são utilizados quando garantidamente temos que repetir um conjunto de instruções um número variável e, à partida, desconhecido de vezes. A sua sintaxe é muito simples:

```

1 while <condição>:
2 <instruções>

```

Graficamente apresenta-se de modo semelhante ao ciclo **for**.

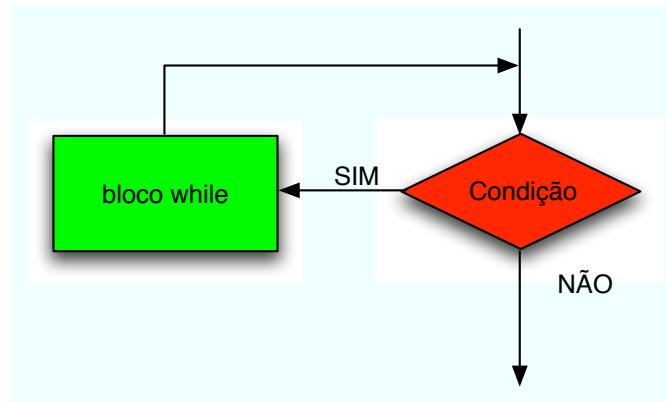


Figura 5.7: O ciclo while

Imaginemos uma situação muito simples em que o programa pede ao utilizador a introdução de um número positivo. Para garantir que o número é mesmo positivo podemos usar um ciclo **while** que é executado enquanto o utilizador não cumprir com o pretendido. A listagem 5.14 mostra um possível programa<sup>4</sup>.

```

1 def main():
2 numero=input("numero por favor:\t")
3 while numero <=0:
4 print ("\n0 numero tem que ser positivo!")
5 numero=input("numero por favor:\t")
6 print (" Numero e igual a %d" % numero)

```

Listagem 5.14: Introdução protegida de dados

**Exemplos Simples** Apresentamos de seguida alguns exemplos simples que recorrem aos ciclos indefinidos. O primeiro vai imprimindo uma cadeia de caracteres, primeiro completa, depois sem o primeiro carácter, de seguida sem os dois primeiros caracteres, e assim sucessivamente até esgotar os caracteres.

```

1 def simples():
2 palavra=raw_input("Entre a palavra sff:\t")
3 while palavra:
4 print(palavra, end=' ')
5 palavra=palavra[1:]
6 return 0

```

O segundo exemplo permite imprimir os números ímpares por ordem decrescente a partir de um certo valor fornecido.

```

1 def impares():
2 limite=input("Entre o limite superior sff:\t")
3 while limite:
4 if limite % 2 != 0:
5 print(limite, end=' ')
6 limite = limite -1
7 return 0

```

O terceiro exemplo, envolve cadeias de caracteres e permite contar o número de ocorrências de um padrão ocorre numa dada sequência. Este problema é inspirado na biologia. Por exemplo, pode servir para determinar

---

<sup>4</sup>Não queremos dizer com isto que esta é a única forma (ou mesmo a melhor) de resolver este problema concreto. Serve apenas de exemplo muito simples!

quantas vezes a sequência **TATA**<sup>5</sup> ocorre numa cadeia de ADN. Quer a cadeia quer o padrão são codificados como cadeias de caracteres.

```

1 def conta_modelo(cadeia, padrao):
2 conta = 0
3 sobreposicao = len(padrao) - 1
4 while len(cadeia) > sobreposicao:
5 if cadeia.startswith(padrao):
6 conta += 1
7 cadeia = cadeia[1:]
8 return conta

```

A ideia do programa é simples: ir avançando na cadeia uma posição de cada vez, testando se nessa posição se inicia o padrão. Avançamos um a um pois admitimos poderem existir padrões sobrepostos.

## 5.5 Intermezzo

A experiência acumulada na resolução de problemas por recurso ao computador fez emergir um conjunto de soluções tipo, e estão na origem do aparecimento dos chamados **padrões de desenho**<sup>6</sup>. Suponhamos que nos pedem para calcular a soma dos primeiros  $n$  números inteiros positivos, isto é:

[Padrões de Desenho](#)

$$\sum_{i=1}^n i$$

Uma maneira de resolver o problema é dispor dos números em sequência e ir efectuando as sucessivas adições: somamos 1 com 2, o resultado 3 somamos com 3, o resultado 6 somamos com 4, e assim sucessivamente. O processo termina quando adicionamos o último número,  $n$ , ao resultado parcial anterior. Este método obriga então a saber sempre qual o resultado parcial num dado instante e o valor do próximo número a somar a esse resultado parcial. Do ponto de vista informático recorremos a duas variáveis: uma que guarda o resultado total parcial, a outra que guarda o valor do próximo número a adicionar. A figura 5.8 ilustra o procedimento.

Posto isto o programa que resolve o nosso problema é trivial:

```

1 def padrao_1(n):
2 acum = 0

```

---

<sup>5</sup>Conhecida por *TATA box* na literatura inglesa.

<sup>6</sup>Do inglês *Design Patterns*.

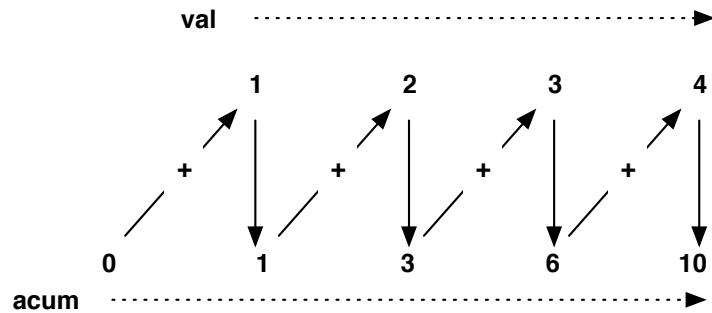


Figura 5.8: O Padrão Acumulador

```

3 for val in range(1,n+1):
4 acum = acum + val
5 return acum

```

A solução baseia-se na existência de um ciclo, sendo que em cada execução do ciclo há um valor que vai sendo acumulado numa variável. Essa variável de acumulação tem que ser inicializada, o que fazemos atribuindo-lhe o valor do elemento neutro da operação usada na acumulação. Neste exemplo a contagem do número de vezes que o ciclo se repete é implícita, embora noutro tipo de problemas essa contagem seja explícita e, nalguns casos, essa contagem seja o valor que queremos saber.

### Invariante de ciclo

Há um aspecto importante quando se usam ciclos nos nossos problemas: propriedades que são verdadeiras à entrada do ciclo e no final de cada execução do ciclo. São designadas por **invariante de ciclo**. Veremos num capítulo mais à frente como os invariantes de ciclo nos podem ajudar a mostrar a correcção do nosso programa e a desenvolver o próprio programa. No nosso caso o invariante é:

$$acum = \sum_{i=1}^k i \wedge val = k$$

Quando saímos do ciclo *val* tem o valor *n* pelo que *acum* tem guardado a soma pretendida.

Porque chamamos a este tipo de programa um padrão de desenho? Bom, imaginemos que o pretendido não era a soma mas o produto dos primeiros *n* inteiros positivos. Olhemos para a solução:

```

1 def padrao_2(n):
2 acum = 1
3 for val in range(1,n+1):
4 acum = acum * val
5 return acum

```

Quando as compararmos o que se alterou foi apenas a operação de acumulação e a inicialização do acumulador com o valor do elemento neutro para a operação (no caso multiplicação). Seja agora o problema ligeiramente diferente de guardar todos os prefixos de uma dada sequência. Eis o programa:

```

1 def padrao_3(seq):
2 acum = ()
3 for val in range(len(seq) + 1):
4 acum = acum + (seq[:val],)
5 return acum

```

Usámos um tuplo para acumular os resultados parciais. A operação agora é a de concatenação de tuplos. Uma vez mais se pode ver a semelhança com os outros dois programas. Deixamos ao leitor o cuidado de definir os invariantes de ciclo para estes dois últimos casos. Todos estes programas podem ser definidos em função do mesmo padrão:

```

1 def padrao(objecto):
2 acum = func_0()
3 for val in func_1(objecto):
4 acum = func_2(acum, val)
5 return acum

```

## 5.6 Qual o valor de $\pi$ ?

Existem várias categorias de números. Os números irracionais são números que não podem ser expressos como o quociente de dois números inteiros, sendo a sua parte decimal uma sequência infinita não periódica. Por isso, o melhor que podemos fazer é aproximar-los por meio de um número real, truncando a parte decimal. Mas os números irracionais ainda se podem dividir em algébricos e transcendentais. Os algébricos são raízes raízes de equações algébricas de coeficientes inteiros como, por exemplo  $\sqrt{2}$ . Os transcendentais são números como a base dos logaritmos naturais,  $e$ , ou o número  $\pi$ . O número  $\pi$  define-se como o quociente entre o perímetro e o diâmetro de uma circunferência. Ao longo dos anos muitas foram as soluções para se obter o

valor aproximado deste número. Vamos usar **Python** para implementar algumas das fórmulas e métodos<sup>7</sup>. Mas antes de discutir a solução informática importa dizer que desde há longos séculos se procurou encontrar um valor aproximado para  $\pi$ . A listagem abaixo mostra algumas das tentativas.

```

1 >>> # Babilónia
2 >>> 3 + 1/8
3 3.125
4 >>> # Antigo Egipto
5 >>> 256/81
6 3.1604938271604937
7 >>> # Grécia
8 >>> 10**0.5
9 3.1622776601683795
10 >>> # Hipparchus
11 >>> 377/120
12 3.1416666666666666
13 >>> # Tsu Chung-chih
14 >>> 355/113
15 3.1415929203539825
16 >>> # Fibonacci
17 >>> 864/275
18 3.14181818181816

```

Cientistas mais recentes procuram aproximações tão exactas como quiséssemos. Por exemplo, **Leibniz** propôs a fórmula, uma série infinita::

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$$

Por seu lado, **Wallis** propôs como aproximação:

$$\frac{\pi}{2} = 2 \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \dots$$

São estes dois métodos que vamos usar para construir duas soluções informáticas. Se as duas fórmulas são distintas do ponto de vista informático são muito semelhantes, uma vez que remetem para um mesmo **padrão** de programação, baseado no recurso a um ciclo e uma variável de acumulação. Comecemos pela fórmula de Leibniz. A única dificuldade está em definir o

---

<sup>7</sup>Não se esqueça: os computadores têm limitações na representação dos números reais, logo o resultado que obtiver nunca pode ter mais algarismos significativos do que aqueles que o sistema permite.

método que nos permite ir enumerando os diferentes termos da série. Cada termo é uma fração em que o denominador é sempre 1 e o denominador são os números ímpares. Para além disso, o sinal vai alternando. Clarificados estes pontos podemos escrever o programa.

```

1 def leibniz_pi(num_termos):
2 """
3 Calcula valor de pi segundo fórmula de Leibniz.
4 """
5 acum = 0.0
6 sinal = 1
7 for i in range(1,2*num_termos+1,2):
8 acum = acum + sinal * (1.0/i)
9 sinal *= -1
10 return 4 * acum

```

Listagem 5.15:  $\pi$  segundo Leibniz

Note-se como se faz a alternância do sinal. Claro que podemos imaginar alternativas. Por exemplo, sabendo que a forma compacta da série é dada por

$$\frac{\pi}{4} = \sum_{i=0}^n (-1)^i \times \frac{1}{2 \times i + 1}$$

escrevemos o programa:

```

1 def leibniz_pi(num_termos):
2 """ Calcula valor de pi segundo fórmula de Leibniz.
3 """
4 acum = 0.0
5 for i in range(num_termos):
6 acum = acum + ((-1)**i) * (1.0/(2 * i + 1))
7 return 4 * acum

```

Passemos agora ao caso da fórmula de Wallis. Este caso parece um pouco mais complexo, devido ao modo como se podem gerar os diferentes factores que depois vão ser usados na acumulação do resultado. Uma ideia possível é juntar **dois** factores de cada vez:

```

1 def wallis_1(num_fact):
2 """
3 Calcula o valor de pi usando a fórmula de Wallis.
4 """
5 acum = 1.0

```

```

6 for i in range(2, num_fact,2):
7 esquerda = i / float((i-1))
8 direita = i /float((i+1))
9 acum = acum * esquerda * direita
10 return 2 * acum

```

Uma vez mais, temos alternativas a este programa. Uma resulta de considerar o produto dos factores dois a dois:

```

1 def wallis_pi(num_fact):
2 """
3 Calcula o valor de pi usando a fórmula de Wallis.
4 """
5 acum = 1.0
6 for i in range(1, num_fact/2):
7 acum = acum * (float((2 * i) ** 2) / ((2 * i) ** 2 -
8 1))
9 return 2 * acum

```

Comparando esta versão com a última baseada na fórmula de Leibniz fica patente a semelhança dos dois **programas**.

### Método de Monte Carlo

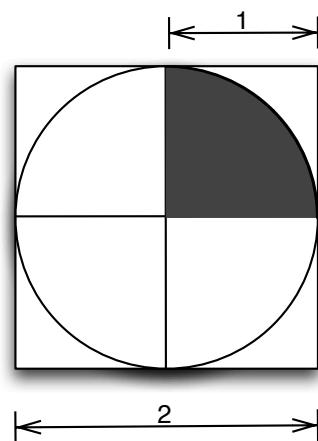
Admitamos que temos um quadrado de lado 2 unidades e dentro dele uma circunferência de raio 1. Divida-se o quadrado em quatro partes iguais. É óbvio que área de cada um dos quatro quadrados pequenos é de 1 (ver figura 5.9).

Por outro lado sabemos que a área da circunferência é igual a  $\pi \times r^2 = \pi^8$ . Logo, a área da parte da circunferência que cobre cada quadrado pequeno vale  $\frac{\pi}{4}$ . O método de Monte Carlo<sup>9</sup>, é um método estocástico inicialmente proposto por Stanislaw Ulam e Nicholas Metropolis, que pode ser usado para calcular o valor aproximado de  $\pi$ . Neste caso, consiste em simular o lançamento de dardos na direcção de um dos quadrados pequenos, no caso o assinalado, contar a proporção dos que caiem dentro do quarto de circunferência, e multiplicar esse valor por 4. Implementar uma solução informática obriga a resolver duas questões: primeiro, como simular o lançamento dos dados e, em segundo como determinar o número de dardos que cairam *dentro* da circunferência. A primeira questão, resolve-se recorrendo a uma distribuição uniforme, no intervalo  $(0, 1)$ , disponibilizada pelo módulo `random`. Com

---

<sup>8</sup>Pois o raio é 1.

<sup>9</sup>O nome do método foi uma homenagem ao Principado do Mónaco e à sua cidade Monte Carlo, que, como sabemos, tem uma longa tradição em jogos de fortuna.

Figura 5.9: Simulação de monte Carlo: o caso de  $\pi$ 

ela geramos as coordenadas  $(x, y)$  do dardo. A segunda questão, envolve o cálculo da distância euclidiana do ponto  $(x, y)$  à origem  $(0, 0)$ . Se essa distância for menor ou igual a 1 é porque o ponto  $(x, y)$  está no interior (ou sobre) a circunferência. Postas estas considerações, apresentamos o programa. Deixamos ao leitor o cuidado de correr o programa com diversos valores para o número de dardos e verificar a precisão do resultado.

```

1 import random
2
3 def monte_carlo_pi(num_dardos):
4 """
5 Calcula o valor de pi pelo método de Monte Carlo.
6 """
7 # define e inicializa acumulador
8 conta_dardos_in = 0.0
9 for i in range(num_dardos):
10 # gera posição dardo i
11 x= random.random()
12 y= random.random()
13 # calcula distância à origem
14 d = (x**2 + y**2)**0.5
15 if d <= 1:
16 conta_dardos_in = conta_dardos_in + 1
17 res_pi = 4 * (conta_dardos_in/num_dardos)
18 return res_pi

```

Podemos criar uma versão animada do método de Monte Carlo, como se ilustra na listagem 5.16.

```
1 import random
2 import turtle
3
4 def visualiza(pontos):
5 """
6 Valor de pi pelo método de Monte Carlo.
7 Versão gráfica.
8 """
9 # Prepara a visualização
10 turtle.setworldcoordinates(-2,-2,2,2)
11 janela = turtle.Turtle()
12 janela.hideturtle()
13 # Desenha os eixos
14 janela.up()
15 janela.goto(-1,0)
16 janela.down()
17 janela.goto(1,0)
18
19 janela.up()
20 janela.goto(0,1)
21 janela.down()
22 janela.goto(0,-1)
23 # Desenha circunferência
24 janela.up()
25 janela.goto(0,-1)
26 janela.down()
27 janela.circle(1, steps=360)
28 # Desenha quadrado
29 janela.up()
30 janela.goto(-1,-1)
31 janela.down()
32 for i in range(4):
33 janela.forward(2)
34 janela.left(90)
35 janela.up()
36 # Mostra dardos
37 for elem in pontos:
38 x,y = elem
```

```

39 d = (x**2 + y**2) ** 0.5
40 if d <= 1:
41 janela.color("blue")
42 else:
43 janela.color("red")
44 janela.goto(x,y)
45 janela.dot()
46
47
48 def monte_carlo_pi(num_dardos):
49 """
50 Calcula o valor de pi pelo método de monte Carlo.
51 """
52 # define e inicializa acumulador a armazenador
53 conta_dardos_in = 0.0
54 tuplos_dardos = tuple()
55 for i in range(num_dardos):
56 # gera posição dardo i
57 x= random.random()
58 y= random.random()
59 tuplos_dardos += ((x,y),)
60 # calcula distância à origem
61 d = (x**2 + y**2)**0.5
62 if d <= 1:
63 conta_dardos_in = conta_dardos_in + 1
64 res_pi = 4 * (conta_dardos_in/num_dardos)
65 print(res_pi)
66 return tuplos_dardos
67
68
69 if __name__ == '__main__':
70 dardos= monte_carlo_pi(1500)
71 visualiza(dardos)
72 turtle.exitonclick()

```

Listagem 5.16: Método de Monte Carlo animado

Como se pode ver, alterámos ligeiramente o programa que calcula o valor de  $\pi$ , por forma a guardar num tuplo as coordenadas dos diferentes pontos. O cálculo dos pontos e a sua visualização foi feita separadamente. Ao executar o programa com 1500 dardos obtemos o resultado final ilustrado na figura 5.10.

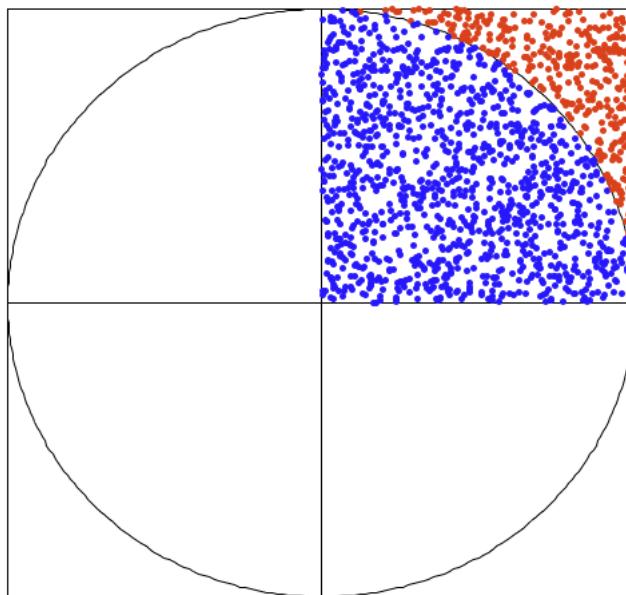


Figura 5.10: Monte Carlo animado

### Simulações

O método de Monte Carlo pode ser usado em diferentes situações. Suponhamos que queremos calcular a área sob um curva, entre dois pontos determinados<sup>10</sup>. Para ser mais concreto, admitamos que se trata da função  $e^{-x^2}$  cujo gráfico se apresenta na figura 5.11.

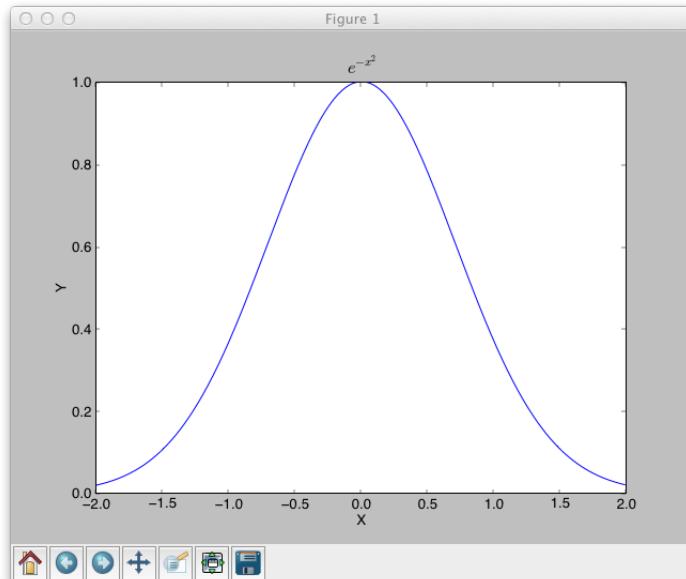
A ideia consiste em criar um rectângulo envolvendo a parte que nos interessa, lançar os dardos para esse rectângulo, calcular a percentagem dos que caem por debaixo da curva e usar esse valor para saber o valor da área. Vejamos o programa.

```

1 import random
2 import math
3
4 def area_f(f,a,b,min_, max_, num_dardos):
5 """ Calcula a área sob a curva f entre a e b. """
6 conta = 0
7 area = (max_ - min_) * (b - a)
8 for i in range(num_dardos):

```

<sup>10</sup>O leitor reconhecerá que se pretende calcular o integral da função entre dois pontos.

Figura 5.11: A função  $e^{-x^2}$ 

```

9 x = random.uniform(a,b)
10 y = random.uniform(min_,max_)
11 if y <= f(x):
12 conta += 1
13 percent = conta / num_dardos
14 return percent * area
15
16 def f(x):
17 return math.exp((-x**2)/2)
18
19 if __name__ == '__main__':
20 print(area_f(f,0,2,0,1,1000))

```

Listagem 5.17: Calcular uma área

## 5.7 Outras Instruções de controlo

Existem muitas situações em programação em que se justifica interromper um ciclo no meio da sua execução. Essa interrupção pode ser definitiva, i.e., o ciclo é abandonado ou então parcial levando apenas ao regresso ao início

do ciclo<sup>11</sup>.

### **break**

#### **break**

Suponhamos que queremos determinar o **maior** factor que divide um número. Um modo de o fazer consiste em testar de modo descendente os números candidatos. É claro que mal encontrarmos esse número não temos mais nada a fazer e queremos abandonar o teste. A instrução **break** permite-nos realizar esse objectivo de modo elegante como nos mostra a listagem 5.18.

```

1 def factor_max(y):
2 x= y /2
3 while x>1:
4 if y % x == 0:
5 print('O maior factor de %d é %d' % (y,x))
6 break
7 x = x-1

```

Listagem 5.18: Break: exemplo de uso

Por paradoxal que pareça existem programas que funcionam sem nunca parar. Os sistemas operativos são o exemplo paradigmático. Em programação de aplicações também existem situações em que nos vemos obrigar a ter um ciclo *potencialmente* infinito. Recorre-se para tal a um padrão de programação que envolve a trilogia **while-True-break**. No exemplo da listagem 5.19 o programa pede ao utilizador nome e idade até que o utilizador introduza o nome **stop** altura em que o programa deve terminar.

```

1 def entra_dados():
2 """ Exemplo de uso de break."""
3 while True:
4 nome=input("O seu nome:\t")
5 if nome =='stop':
6 break
7 idade=int(input("A sua idade:\t"))
8 print ("\nViva %s!\t %d é uma linda idade..." % (nome,
 idade))

```

Listagem 5.19: Ciclos *potencialmente* infinitos

---

<sup>11</sup>Este tipo de quebra do percurso normal da execução de um programa é semelhante às antigas instruções **goto** que tanta celeuma geraram nos anos 70. No entanto, estas instruções quebram um programa de modo *controlado*.

Num último exemplo, reproduzido na listagem 5.20, procuramos o maior número a partir de um certo limite que é um quadrado perfeito.

```

1 from math import sqrt
2
3 def quad_perfeito(n):
4 "O maior quadrado perfeito menor do que n"
5 for num in range(n,0,-1):
6 raiz=sqrt(num)
7 if raiz == int(raiz):
8 print ("Maior quadrado perfeito menor do que %d é %d" %
9 (n,num))
10 break

```

Listagem 5.20: Break: quadrados perfeitos

### continue

Há situações em que não queremos interromper o ciclo mas apenas retomá-lo a partir do início. Um exemplo simples é quando queremos filtrar certos casos em que nada acontece de outros em que é preciso fazer algo. O ciclo fica assim separado em duas partes. O separador é um teste condicional. Na listagem 5.21 mostramos o exemplo (um pouco académico) de queremos imprimir a sequência de números ímpares.

[continue](#)

```

1 def impares(x):
2 """ Exemplo de uso de continue."""
3 while x:
4 x=x-1
5 if x % 2 == 0:
6 continue
7 print("%d é ímpar." % x)
8 print ("\nFinito!")
9 return 0

```

Listagem 5.21: Continue: ímpares

O exemplo seguinte (listagem 5.22) mostra a protecção para a entrada de um código. Também exemplifica a possibilidade de combinar **break** e **continue**.

```

1 def codigo():
2 "Pedir um código com exactamente quatro caracteres"
3 while True:

```

```

4 cod = input('Código sff: ')
5 if len(cod) != 4:
6 print('O código tem que ter 4 caracteres')
7 continue
8 else:
9 print('Bem-vindo')
10 break

```

Listagem 5.22: Continue: entra código

Há um aspecto importante sobre a instrução **continue** que deve ser sublinhado pois é muitas vezes mal entendido. A instrução faz voltar para o início do ciclo **obrigando** ao teste da condição de saída antes de retomar a execução do ciclo, o que só acontece se o teste vir verdadeiro.

**else****else**

Os ciclos **while** e **for** podem ser enriquecidos por utilização de uma cláusula **else** a seguir ao ciclo. Associada a essa cláusula estão instruções que serão executadas caso o ciclo **não** tenha sido interrompido por meio de um **break**. Retomando o exemplo 5.18 podemos ver como se pode usar essa possibilidade para encontrar números primos, como ilustra a listagem 5.23.

```

1 def primo(y):
2 x= y /2
3 while x>1:
4 if y % x == 0:
5 print (y, "Tem factor ",x)
6 break
7 x = x-1
8 else:
9 print (y, " é um número primo")

```

Listagem 5.23: Else: números primos

Vejamos mais um exemplo de introdução controlada de um código. No exemplo da listagem 5.24 podemos ver uma combinação do uso de **break**, **continue** e **else**.

```

1 def password(lista_passw):
2 " Três chances para introduzir correctamente uma password"
3 conta=3
4 while conta:
5 codigo=input("Entre o seu código sff:")
6 if codigo in lista_passw:

```

```

7 print "Bem-vindo"
8 break
9 print ("Código errado.")
10 conta= conta - 1
11 continue
12 else:
13 print("Acabaram as suas tentativas!!!")
```

Listagem 5.24: Tudo junto

`pass`

O leitor consegue imaginar uma instrução que não faz nada? Pois existe e `pass` chama-se `pass`. Mas para que pode servir? Bom, para algumas coisas como por exemplo:

- no desenvolvimento de programas. Deste modo podemos testar partes do programa, deixando para mais tarde completar o que está em desenvolvimento. Consegue-se também deste modo isolar melhor eventuais erros no código.
- quando somos obrigados, por razões **sintáticas**, a colocar uma instrução numa zona do código.
- no tratamento de excepções, assunto que será tratado mais adiante.

No desenvolvimento de programas podemos ter situações por finalizar mas queremos que mesmo assim o resto do código possa ser testado. Um exemplo muito simples:

```

1 def verifica(nome):
2 if nome =='Ernesto Costa':
3 print('Bem-vindo')
4 elif nome == 'Bill Gates':
5 print ('Acesso Negado!')
6 else:
7 # depois decidir...
8 pass
```

Suponhamos agora uma situação em que o programa está à espera de uma interrupção via teclado. O programa abaixo ilustra a necessidade sintática da instrução `pass`.

```

1 while True:
2 pass
```

Outro exemplo de necessidade sintática:

```

1 def nada(x):
2 if cond1(x):
3 f1(x):
4 elif cond2(x):
5 pass
6 else:
7 fn(x)

```

Aqui pretendo que, no caso de uma dada situação for verdadeira, não fazer nada!

## 5.8 Excepções

Infelizmente quem programa sabe que das coisas mais comuns são erros durante a execução do programa. Isso corresponde em muitos casos ao aparecimento de situações não previstas. O tratamento de excepções permite reagir de modo controlado a situações anómalas do meu programa, procurando sempre que possível dar informações relevantes ao programador por forma a que possa melhorar o seu código. A situação da listagem 5.25 ilustra o caso simples em que isolamos a situação de uma divisão por zero.

```

1 def try_1(x):
2 try:
3 y=float(input("\nDenominador: "))
4 print(float(x) / y)
5 except ZeroDivisionError:
6 print("\nCuidado: o denominador não pode ser zero!")

```

Listagem 5.25: Excepções: divisão por zero

O código que se protege é colocado entre a palavra reservada **try** e a palavra reservada **except**. Quando o erro ocorre o código que se segue ao **except** é executado. O próximo exemplo mostra uma situação mais realista de protecção de dados de entrada.

```

1 def try_2():
2 """ Exemplo de uso de pass.
3 Espera interrupção pelo keyboard.
4 """
5 while True:
6 try:
7 x=int(input("Um número sff:\t"))

```

```

8 break
9 except ValueError:
10 pass
11 print "\nFinito!"
```

Regressando ao nosso exemplo das raízes de uma equação do segundo grau vejamos (listagem 5.26) como podemos proteger o programa que apenas prevê raízes reais.

```

1 def try_3():
2 """ Calculo das raizes reais de um poinomio.
3 """
4 try:
5 a,b,c = input("Os coeficientes sff (a,b,c):\t")
6 discriminante=pow(b,2)- 4 * a * c
7 raiz_discrim = math.sqrt(discriminante)
8 raiz1=float((-b + raiz_discrim)) / (2 * a)
9 raiz2=float((-b - raiz_discrim)) / (2 * a)
10 if raiz1 == raiz2:
11 print ("O polinomio de coeficientes\
12 a=%d b=%d c= %d tem raizes multiplas raiz1= raiz2=%3.2f
13 " % (a,b,c,raiz1))
14 else:
15 print("As raizes do polinomio de coeficientes\
16 a=%d b=%d c= %d sao raiz1=%3.2f raiz2=%3.2f" % (a,b,c,
17 raiz1,raiz2))
18 except ValueError:
19 print ("\n Nao tem raizes reais!")
```

Listagem 5.26: Try: raízes reais apenas

Um exemplo clássico do uso de exceções envolve a abertura de um ficheiro, como se ilustra a seguir.

```

1 def try_4():
2 while True:
3 try:
4 fich = input("Nome do ficheiro: ")
5 f_in = open(fich, 'r')
6 break
7 except IOError:
8 print("O ficheiro %s não existe. Tente de novo." % fich)
9 # resto do programa
```

Neste caso estamos a pedir o nome de um ficheiro até ele ser válido, altura em que se abandona o ciclo `while`.

Existem vários tipos de excepções., estando organizadas numa hierarquia. A figura 5.12 dá uma visão parcial dessa hierarquia. O leitor é convidado a consultar o manual de referência da linguagem.

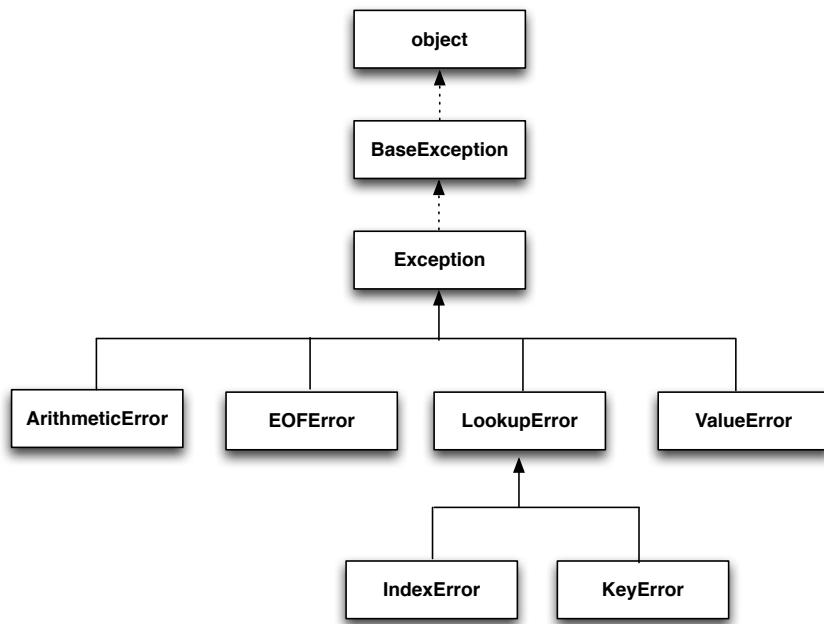


Figura 5.12: Hierarquia de excepções (visão parcial)

Nada impede no entanto o utilizador de definir, ele próprio, excepções. Embora tal exija conhecimentos que o leitor ainda não tem mostramos mesmo assim um exemplo simples na listagem 5.27.

```

1 # nova excepção
2 class ParseError(Exception):
3 pass
4
5 def testa(x):
6 try:
7 parse(x)
8 except ParseError:
9 pass

```

Listagem 5.27: Exepções: definição pelo utilizador

Associado com um **try** podem existir mais do que uma cláusula **except** embora só uma seja activada (a que corresponder à excepção levantada).

```

1 import sys
2
3 try:
4 f = open('myfile.txt')
5 s = f.readline()
6 i = int(s.strip())
7 except IOError, (errno, strerror):
8 print("I/O error(%s): %s" % (errno, strerror))
9 except ValueError:
10 print("Os dados não puderam ser convertidos para inteiros
11 .")
12 except:
13 print("Erro inesperado:", sys.exc_info()[0])
14 raise

```

Neste exemplo chamamos a atenção para o último **except** onde não existe nenhuma excepção definida associada e é levantada (**raise**) uma geral. Pode ainda aparecer uma cláusula **else**, sempre a seguir às excepções, e que será executada se nada de anormal se passar com o **try**. No exemplo abaixo estamos a pedir dados ao utilizador até ele não cometer nenhum erro.

```

1 def try_5():
2 while True:
3 try:
4 x=input('o numerador:')
5 y=input('o denominador:')
6 res= x/y
7 print('%d a dividir por %d = %d' % (x,y,res))
8 except:
9 print('Entrada inválida. Tente novamente.')
10 else:
11 break

```

Para concluir importa referir que pode também existir uma cláusula opcional denominada **finally** no fim da excepção e que será **sempre** executada independentemente do resultado das operações do **try - except**. Normalmente é usado para actividades de *limpeza*, como por exemplo fechar um ficheiro com segurança. Veja-se o exemplo da listagem 5.28

```

1 >>> def divide(x, y):
2 ...
3 try:

```

```

3 result = x / y
4 except ZeroDivisionError:
5 print ("divisão por zero!")
6 else:
7 print("o resultado é", result)
8 finally:
9 print("a executar a cláusula finally")
10 ...
11 >>> divide(2, 1)
12 o resultado é 2
13 a executar a cláusula finally
14 >>> divide(2, 0)
15 divisão por zero!
16 a executar a cláusula finally
17 >>> divide("2", "1")
18 a executar a cláusula finally
19 Traceback (most recent call last):
20 File "<stdin>", line 1, in ?
21 File "<stdin>", line 3, in divide
22 TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

Listagem 5.28: try: uso de finally

## Asserções

Acontece com alguma frequência querermos interromper um programa se uma dada condição não se verificar. Tal pode ser feito com um teste simples como ilustra a listagem 5.29.

```

1 if <condição>:
2 <aborta_programa>

```

Listagem 5.29: Teste para abortar programa

`assert`

É mais correcto e genérico usar o comando `assert`.

```

1 def exe_ass(n):
2 assert 7 < n < 77, 'Valores fora dos limites'
3 print(n)
4 # -- resultado da chamada com n = 100
5 AssertionTypeError: Valores fora dos limites

```

Como se pode ver, ao falhar a asserção é levantada uma excepção, que podemos usar como anteriormente. As asserções são igualmente importantes para os testes unitários de que falaremos mais adiante.

## Sumário

Neste capítulo introduzimos as duas estruturas de controlo fundamentais: condicionais e ciclos. Vimos as várias variantes, condicionais simples, completas ou gerais e os ciclos definidos e indefinidos. No caso dos ciclos exemplificámos ainda as instruções adicionais que podem ser usadas (**break**, **continue**). No final referimos de modo breve o uso da instrução **pass**, as excepções (**try**) e as asserções (**assert**). Ficam assim cobertas as instruções usadas em programação procedural.

## Testes os seus conhecimentos

Determine se conhece os conceitos indicados e se consegue responder às questões abaixo colocadas.

- O que entende por bloco e como se relaciona com a indentação do código.
- De que maneira pode representar os valores booleanos **True** e **False**.
- Que tipos de instruções condicionais existem e como se distinguem.
- Qual a diferença fundamental entre um ciclo **for** e um ciclo **while**.
- De que modo pode percorrer um ciclo.
- Que formas tem de interromper/quebrar um ciclo.
- Em que princípios se baseia o Método de Monte Carlo.
- Que diferenças existem entre **excepções** e **asserções**.

## Exercícios

### Exercício 5.1 F

Escreva um programa que lhe permita apresentar uma tabela de conversão entre milhas e quilómetros. Uma milha é igual a 1.609 quilómetros. A tabela deve conter todas as equivalências entre dois valores de referência. A tabela deve ter um aspecto semelhante ao da listagem seguinte, que ilustra o caso entre os números 10 e 20.

|    | Milhas | Quilómetros |
|----|--------|-------------|
| 1  |        |             |
| 2  |        | -----       |
| 3  | 10.00  | 16.09       |
| 4  | 11.00  | 17.70       |
| 5  | 12.00  | 19.31       |
| 6  | 13.00  | 20.92       |
| 7  | 14.00  | 22.53       |
| 8  | 15.00  | 24.13       |
| 9  | 16.00  | 25.74       |
| 10 | 17.00  | 27.35       |
| 11 | 18.00  | 28.96       |
| 12 | 19.00  | 30.57       |
| 13 | 20.00  | 32.18       |

**Exercício 5.2 F** Escreva um programa que apresenta por ordem crescente três números inteiros positivos dados como entrada. Em que medida a sua solução minimiza o número de comparações necessárias?**Nota:** Não pode usar nenhuma função/método de ordenamento pré-definido de Python .

**Exercício 5.3 F** Para realizar a viagem entre o Porto e Coimbra (120 km de distância) existem várias estradas como alternativa. A tabela 5.2 ilustra as diferentes alternativas em termos de trajecto considerando o custo de combustível por km e o custo das portagens. Escreva um programa que dada a designação da estrada retorne o custo total da viagem para essa alternativa.

| Estradas | Custo combustível / Km | Custo Portagens |
|----------|------------------------|-----------------|
| A1       | 0.15                   | 6.52            |
| A20      | 0.12                   | 15.2            |
| A21      | 0.10                   | 5.75            |

Tabela 5.2: O que escolher? Preços em euros.

**Exercício 5.4 F** O vencimento bruto de um trabalhador está sujeito a descontos: 25% para o IRS, 5% para a Segurança social e 10% para a Caixa Nacional de Aposentações. O vencimento líquido é o que resulta da subtração destes descontos ao vencimento bruto. Desenvolva um programa que

dado o vencimento bruto devolve o correspondente vencimento líquido.

**Exercício 5.5 F** A avaliação nesta cadeira resulta de 5 provas: 4 testes e um exame. Cada teste vale 7.5% da nota final, enquanto que o exame vale 70%. Isto significa que a nota é dada pela expressão:

$$\text{nota} = 0.075 * (t_1 + t_2 + t_3 + t_4) + 0.7 * e$$

Escreva um programa que dadas as 5 notas parciais calcula a nota final e, como resultado devolve a cadeia de caracteres "Aprovado", se a média for maior ou igual a 14, "Reprovado", se a média for inferior a 7, e "Oral", se a média for maior ou igual a 7 e inferior a 14. Admita que as notas são números reais entre 0 e 20.

**Exercício 5.6 F** Considere o seguinte pedaço de código Python .

```
1 i = 20
2 while (i >= 0):
3 print "i=",i
4 i = i - 2
```

Listagem 5.30: Ciclo while

Escreva um pedaço de código equivalente em que o ciclo **while** é substituído por um ciclo **for**.

**Exercício 5.7 M**

```
1 >>> for i in range(3):
2 ... for j in range(1,3):
3 ... print float(i)/j
4 ...
5 ?
6 >>>
```

Diga, **justificando**, o que vai aparecer no lugar do **ponto de interrogação** quando o código é executado no interpretador.

**Exercício 5.8 M** Duas palavras de igual comprimento dizem-se **amigas** se o número de posições em que os respectivos caracteres **difere** for inferior a 10%. Escreva um programa que dadas duas palavras indica se são ou não amigas.

**Exercício 5.9 M** Escreva um programa que calcula o divisor mais pequeno

de um número inteiro maior do que 1. Use esse programa como auxiliar na determinação de um dado inteiro maior do que um é um número primo.

**Exercício 5.10** **M** Suponha um dado em que cada uma das faces está numerada com os inteiros de 1 a 6. Desenvolva um programa que simula o lançamento repetido do dado um certo número de vezes, e calcula a **percentagem** de vezes em que saiu um número par.

### Exercício 5.11

Considere a figura 5.13. Suponha que o quadrado externo tem uma dimensão 2 por 2 e os internos 1 por 1.

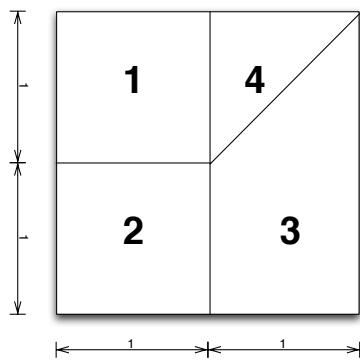


Figura 5.13: Calcular probabilidades

Use o Método de Monte Carlo para calcular a probabilidade de ao atirar um dardo ele cair numa região de número ímpar.

### Exercício 5.12 **F**

O factorial de um número é dado por:

$$\text{fact}(n) = n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

Implemente um programa que lhe permita calcular o factorial de um inteiro positivo.

### Exercício 5.13 **M**

O valor do seno de um ângulo que pode ser computado a partir da fórmula:

$$\text{seno}(x) = \sum_{i=0}^{\infty} \frac{(-1)^i \times x^{(2 \times i + 1)}}{(2 \times i + 1)!}$$

Implemente o respectivo programa usando o número de parcelas como parâmetro. Altere o programa por forma a poder usar a precisão como critério de paragem do seu programa.

### Exercício 5.14 Módulo matplotlib F

O número harmónico  $H_n$  define-se pela fórmula:

$$H_n = \sum_{k=1}^n \frac{1}{k}$$

Escreva um programa que permita calcular o valor de  $H_n$ , usando  $n$  como parâmetro. Use esse programa para poder encontrar os sucessivos valores dos números harmónicos até um dado limite. Recorra ao módulo **matplotlib** para visualizar o resultado. Que comentários se lhe oferecem fazer face ao gráfico?

### Exercício 5.15 F

Os números harmónicos podem ser calculados de modo aproximado pela fórmula:

$$H_n \approx \ln(n) + \gamma$$

com  $\ln(n)$  o logaritmo natural (base e) e  $\gamma = 0.5772156649$  a constante de Euler. Escreva um programa que lhe permita calcular o valor de  $H_n$  de modo aproximado para sucessivos valores de  $n$ . Visualize o resultado e compare com o obtido no exercício 5.8. Que pode dizer sobre a qualidade da aproximação?

### Exercício 5.16 F

O logarithm natural é definido pela fórmula:

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Escreva um programa que lhe permita calcular o valor aproximado de  $e$  com uma dada precisão.

### Exercício 5.17 M

Um número diz-se perfeito se for igual à soma dos seus divisores próprios. Por exemplo, 6 e 28 são perfeitos. Escreva um programa que determine quais os números perfeitos que existem num dado intervalo.

### Exercício 5.18 M

Considere os padrões de números seguintes.

```

1 # Padrão A
2 1
3 1 2
4 1 2 3
5 1 2 3 4
6 1 2 3 4 5
7
8 #Padrão B
9 1 2 3 4 5
10 1 2 3 4
11 1 2 3
12 1 2
13 1
14
15 # Padrão C
16 1
17 2 1
18 3 2 1
19 4 3 2 1
20 5 4 3 2 1

```

Escreva três programas, que lhe permitam imprimir cada um dos padrões. Em que medida a sua solução depende da dimensão do número máximo  $n$ ?

### Exercício 5.19 Módulo turtle M

Suponha que quer desenhar uma grelha como a da figura 5.14, em que a dimensão da grelha e o tamanho de cada célula são parâmetros do problema. Use o módulo **turtle** para o fazer.



Figura 5.14: Desenho de uma grelha

**Exercício 5.20** Módulo `turtle` M

Um passeio aleatório é um conceito que permite modelizar vários processos que ocorrem na natureza. Admita que tem um agente que se movimenta de modo aleatório num mundo 2D. Suponha que a cada momento o agente decide deslocar-se ou para norte, ou para este, ou para sul ou para oeste, sendo que essa decisão é aleatória. Usando o módulo `turtle` simule um passeio aleatório do nosso agente, admitindo que o seu mundo 2D tem a forma de uma grelha como a que obteve no exercício 5.8. O mundo é suposto ser **finito**. A figura 5.15 ilustra o que se pretende ( o ponto marca o início do passeio e a seta o final).

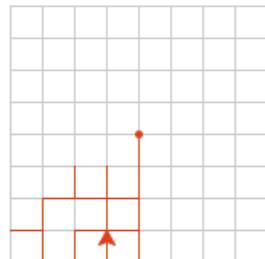


Figura 5.15: Passeio aleatório

**Exercício 5.21** DA sequência de Fibonacci define-se indutivamente do modo seguinte: os seus dois primeiros números são iguais a um e a partir daí cada elemento da sequência é igual à soma dos dois elementos imediatamente anteriores. Eis os primeiros números da sequência:

- 1 1, 1, 2, 3, 5, 8, 13, 21,....

Escreva um programa que dado um número **verifica** se ele pertence ou não à sequência de Fibonacci.



# Objectos Estruturados

## Objectivos

- ✓ introduzir os objectos estruturados lista e dicionário
- ✓ mostrar por meio de exemplos simples a sua utilidade

### 6.1 Introdução

Umas das práticas mais comuns em programação consiste em associar um nome a um objecto para mais tarde podermos referenciar o objecto por esse nome. A instrução que nos permite fazer isso, já o sabemos, é a instrução de atribuição. Mas o que acontece se quisermos associar vários objectos a um único nome, sendo que entre esses vários objectos existe uma dada relação? Por exemplo, uma lista ordenada com os nomes e classificação de uma prova desportiva ou uma pequena base de dados com os nomes, moradas, idade, data de aniversário e telefone dos nossos colegas de turma? As listas e os dicionários são dois dos objectos estruturados mais importantes em programação. Existem em **Python** e o sistema fornece um conjunto de operações elementares sobre objectos desse tipo. Com o seu uso passamos a ter capacidades de armazenamento (memória) acrescidas e, por isso, podemos aspirar a resolver de modo simples problemas mais delicados. É disso que trataremos neste capítulo.

### 6.2 Listas

Suponhamos que queremos calcular a nota de uma prova em que as perguntas são de escolha múltipla. Uma questão que se nos coloca é como representar a

solução correcta e a resposta do aluno. Com aquilo que já sabemos podemos pensar em usar cadeias de caracteres. Daí decorre a solução do programa 6.1 sendo que a nota é dada pela proporção de respostas certas.

```

1 def nota(exame):
2 '''Calcula a nota de um exame num teste de escolha
3 múltipla.'''
4 solucao = 'ABBEADDB'
5 conta = 0
6 for i in range(len(solucao)):
7 if exame[i] == solucao[i]:
8 conta = conta + 1
9 return float(conta)/len(solucao)

```

Listagem 6.1: Nota

Parece uma solução aceitável. Mas o que acontece se:

- as perguntas têm pesos diferentes
- usamos números e não letras nas respostas
- misturamos letras e números nas respostas

Não são questões sem solução, mas não parece claro que esta opção seja a melhor. Imaginemos outro problema. Agora pretendemos observar ao longo de um conjunto de dias o número de baleias que aparecem num certo local. Com base nas nossas observações queremos poder tirar conclusões, como seja o número médio de baleias que aparecem por dia. Temos por isso que guardar os dados recolhidos. Podemos fazê-lo num ficheiro externo, mas nem sempre a informação que necessitamos guardar precisa ser mantida permanentemente. É neste contexto que aparecem as **listas**. As listas são objectos (em **Python** tudo são objectos!), logo têm identidade valor e tipo. Os elementos das listas são separados por vírgulas. A marca sintáctica das listas são os parênteses rectos (ver exemplos na listagem 6.2).

```

1 >>> baleias = [5,4,7,3,2,3,2,6,4,2,1,7,1,3]
2 >>> baleias
3 [5, 4, 7, 3, 2, 3, 2, 6, 4, 2, 1, 7, 1, 3]
4 >>> id(baleias)
5 11797688
6 >>> type(baleias)
7 <type 'list'>
8 >>>

```

Listagem 6.2: Baleias

A tabela 6.1 mostra os literais usados na construção de diferentes listas.

| Literal                              | Interpretação                 |
|--------------------------------------|-------------------------------|
| [733, -15, 0]                        | Lista de números              |
| ['praxe', 'sporting', 'abracadabra'] | Lista de cadeia de caracteres |
| []                                   | A lista ... vazia!            |
| [1, [2, 3], 4]                       | Lista de listas               |
| [1, 'a', 'b', 3.0 + 4j]              | Lista heterogénea             |
| [x * 2 for x in range(1, 4)]         | Listas por compreensão        |

Tabela 6.1: Literais para Listas

As listas são colecções ordenadas de objectos, de comprimento variável, [Definição](#) acedidas por posição, heterogéneas e mutáveis. Como no caso das cadeias de caracteres existe uma ordem nas listas, sendo por isso também **sequências**. O elementos no interior das listas podem ser de qualquer tipo, incluindo listas, o que justifica a característica de serem heterogéneas. Uma segunda característica que as diferencia das cadeias de caracteres e dos tuplos é o facto de serem objectos **mutáveis**: podemos alterar o seu valor sem alterar a sua identidade. Este aspecto tem consequências muito relevantes como iremos ver ao longo deste texto.

São várias as operações associadas às listas algumas das quais partilhadas com as sequências , como se pode ver na tabela 6.2.

| Nome         | Operador       | Significado |
|--------------|----------------|-------------|
| Indexação    | [< n >]        | Acede       |
| Concatenação | $L_1 + L_2$    | Junta       |
| Repetição    | $L * n, n * L$ | Replica     |
| Pertença     | $in, not in$   | Testa       |
| Comprimento  | $len$          | Quantifica  |
| Fatiamento   | [::]           | Parte       |

Tabela 6.2: Operações sobre Listas

A listagem 6.2 mostra alguns exemplos de uso destas operações.

```

1 >>> [1,2,3][1]
2 2
3 >>> [1,2,3] + [4,5,6]
4 [1, 2, 3, 4, 5, 6]

```

```

5 >>> ['Ai!'] * 4
6 ['Ai!', 'Ai!', 'Ai!', 'Ai!']
7 >>> len([1,2,3])
8 3
9 >>> 2 in [1,2,3]
10 True
11 >>> lista = [1,2,3,4,5,6]
12 >>> lista[1:4]
13 [2,3,4]
14 >>> for i in [1,2,3]:
15 ... print i
16 ...
17 1
18 2
19 3
20 >>> L=[0,1,2,3,4,5,6,7,8,9]
21 >>> L[5]
22 5
23 >>> L[3:7]
24 [3, 4, 5, 6]
25 >>> L[::-2]
26 [0, 2, 4, 6, 8]
27 >>> L[::-1]
28 [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
29 >>>

```

Mas como já referimos também existem diferenças importantes entre as listas e as cadeias de caracteres. As listas são **heterogéneas**:

```

1 >>> lista = [4.34, 'gato', ['eu', 2, 'ele'], 3 + 4j]
2 >>>

```

Esta lista tem como elementos, números (vírgula flutuante e complexos) nas posições 0 e 3, cadeias de caracteres na posição 1 e uma lista de três elementos na posição 2. Este último elemento tem ele próprio três elementos, duas cadeias de caracteres e um número inteiro.

As listas são **mutáveis**:

```

1 >>> lista = [0,1,2,3,4,5,6,7,8,9]
2 >>> id(lista)
3 4502228072
4 >>> lista[5] = 'muta'

```

```
5 >>> lista
6 [0, 1, 2, 3, 4, 'muta', 6, 7, 8, 9]
7 >>> id(lista)
8 4502228072
9 >>> tuplo = (0,1,2,3,4,5,6,7,8,9)
10 >>> id(tuplo)
11 4500285240
12 >>> tuplo[5] = 'não muta'
13 Traceback (most recent call last):
14 File "<string>", line 1, in <fragment>
15 builtins.TypeError: 'tuple' object does not support item
16 assignment
16 >>> tuplo = (0,1,2,3,4) + ('não muta',) + (6,7,8,9)
17 >>> tuplo
18 (0, 1, 2, 3, 4, 'não muta', 6, 7, 8, 9)
19 >>> id(tuplo)
20 4497755816
21 >>> cadeia = 'abcdefghijklm'
22 >>> id(cadeia)
23 4497751096
24 >>> cadeia[5] = '5'
25 Traceback (most recent call last):
26 File "<string>", line 1, in <fragment>
27 builtins.TypeError: 'str' object does not support item
28 assignment
28 >>> cadeia = cadeia[:5] + '5' + cadeia[6:]
29 >>> cadeia
30 'abcde5hgi'
31 >>> id(cadeia)
32 4497765408
```

Posso alterar o valor da lista *lista* sem alterar a sua identidade, o mesmo não sendo verdadeiro no caso do tuplo *tuplo* e no caso da cadeia de caracteres *cadeia*.

## Construtor

Não se estranhará que o nome do construtor das listas seja **list**. O seu uso é semelhante ao dos construtores de outros tipos de objectos.

```
1 >>> lista = list()
2 >>> lista
```

```

3 []
4 >>> lista = list('123')
5 >>> lista
6 ['1', '2', '3']
7 >>> lista = list((1,2,3))
8 >>> lista
9 [1, 2, 3]
10 >>> lista = list(range(5))
11 >>> lista
12 [0, 1, 2, 3, 4]
13 >>> lista = list([1,2,3])
14 >>> lista
15 [1, 2, 3]
16 >>>

```

A primeira situação mostra que o construtor usado sem argumento devolve uma lista especial, a lista vazia, enquanto nas outras situações procura converter o seu argumento numa lista. O argumento deve ser um objecto iterável.

### Mutabilidade e partilha

Para se compreender melhor as implicações da propriedade de mutabilidade importa ter a noção de que uma lista é guardada em memória como uma **tabela de referências** para os objectos que compõem a lista. Por exemplo, se criarmos uma lista que tem por elementos os números 1,2 e 3, através da instrução `lista = [1,2,3]`, teremos a situação ilustrada na figura 6.1.

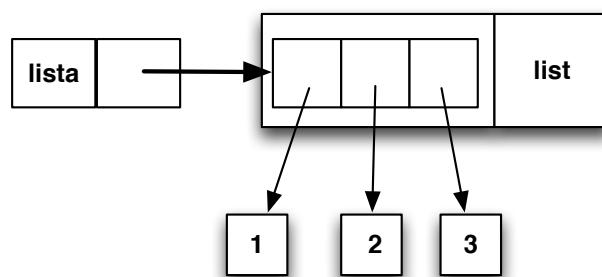


Figura 6.1: A representação de uma lista simples

Se agora alterar o segundo elemento de 2 para 4 a nova situação é retratada na figura 6.2.

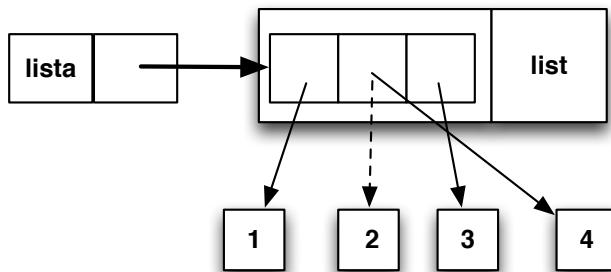


Figura 6.2: A lista alterada

Como se vê pela imagem a referência de toda a lista, isto é a sua identidade, mantem-se! É óbvio que listas mais complexas têm representações ... mais complexas. A figura 6.3 mostra de modo simplificado<sup>1</sup> o caso da lista **nova\_lista** = [1, [2,3,4], [5,6]].

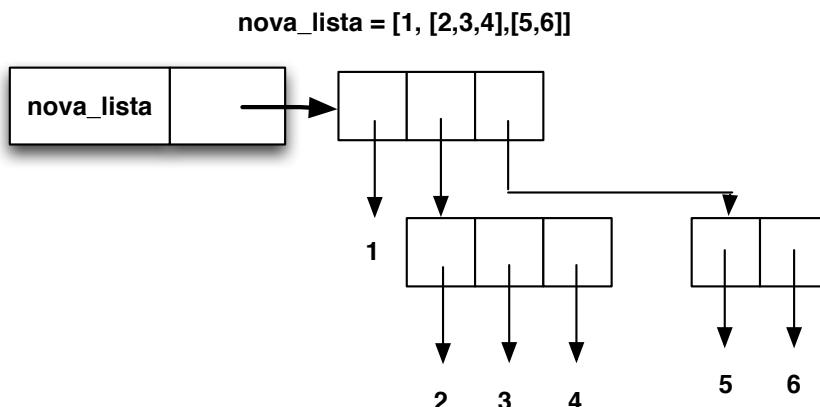


Figura 6.3: Uma lista mais complexa

Suponhamos que temos agora dois nomes distintos associados ao mesmo objecto obtido do modo que a listagem 6.3 exemplifica. Esta situação é conhecida na literatura inglesa por *aliasing*.

```

1 >>> vogais = ['A', 'E', 'I', 'O', 'U']
2 >>> id(vogais)

```

---

<sup>1</sup>Sempre que julgarmos conveniente e não afectar o entendimento, usaremos uma notação gráfica simplificada

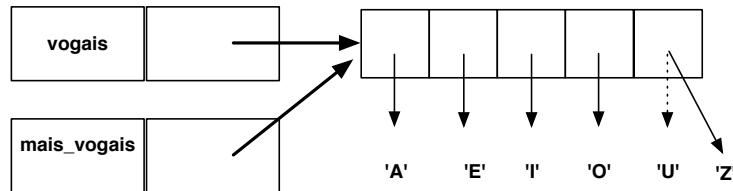
```

3 6913560
4 >>> mais_vogais = vogais
5 >>> id(mais_vogais)
6 6913560
7 >>> vogais[4] = 'Z'
8 >>> vogais
9 ['A', 'E', 'I', 'O', 'Z']
10 >>> mais_vogais
11 ['A', 'E', 'I', 'O', 'Z']
12 >>>

```

Listagem 6.3: Mutabilidade e Referências

A figura 6.4 ilustra a situação em termos da memória.

Figura 6.4: *Aliasing*

Como se pode observar os nomes são **referências** para o mesmo objecto. Assim, naturalmente, ao usar um deles para mudar o valor do objecto o acesso pelo outro nome também encontra o objecto alterado. Este acontecimento pode ter efeitos não desejados. Se não queremos que este efeito lateral aconteça, uma solução é usar um **cópia**. A listagem 6.4 mostra como o podemos fazer.

```

1 >>> vogais = ['A', 'E', 'I', 'O', 'U']
2 >>> id(vogais)
3 11791776
4 >>> copia = vogais[:] # <-- cópia
5 >>> copia
6 ['A', 'E', 'I', 'O', 'U']
7 >>> id(copia)
8 11797968
9 >>> copia[4] = 'Z'
10 >>> copia
11 ['A', 'E', 'I', 'O', 'Z']

```

```

12 >>> vogais
13 ['A', 'E', 'I', 'O', 'U']
14 >>>

```

Listagem 6.4: Mutabilidade e Referências (II)

Esta situação pode ser visualizada (ver figura 6.5).

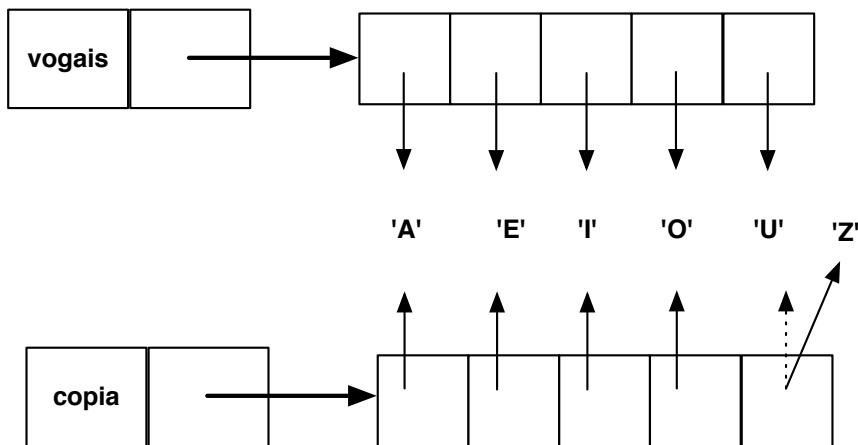


Figura 6.5: Alterar sem efeitos não desejados

Mas cuidado! Esta solução não é perfeita pois apenas as referências de primeiro nível são alteradas. Assim, se o objecto tiver elementos que são listas e alterarmos elementos destas listas o problema acontece de novo. Vejamos um exemplo.

```

1 >>> vogais = ['A', 'E', ['I', 'O'], 'U', 'Z']
2 >>> copia = vogais[:]
3 >>> id(vogais)
4 4421502088
5 >>> id(copia)
6 4421503240
7 >>> copia[2][1] = 'AI!'
8 >>> copia
9 ['A', 'E', ['I', 'AI!'], 'U', 'Z']
10 >>> vogais
11 ['A', 'E', ['I', 'AI!'], 'U', 'Z']
12 >>>

```

Uma vez mais a visualização da situação em memória ajuda a compreender a raiz do problema, como se ilustra na figura 6.6.

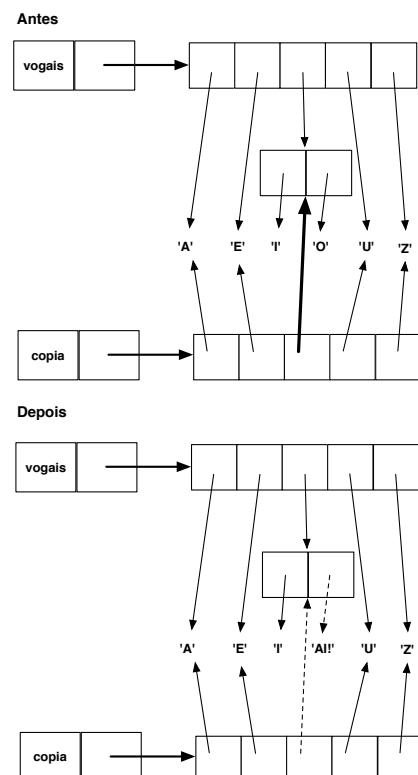


Figura 6.6: Tudo se complica...

### Cópia profunda

Para termos a certeza que não temos nenhuma surpresa devemos fazer uma **cópia profunda**<sup>2</sup> utilizando o método `deepcopy` do módulo `copy`.

```

1 >>> import copy
2 >>> vogais = ['A', 'E', ['I', 'O'], 'U', 'Z']
3 >>> copia = copy.deepcopy(vogais)
4 >>> id(vogais)
5 4421484272
6 >>> id(copia)
7 4421446976
8 >>> copia[2][1] = 'AI!'
9 >>> copia
10 ['A', 'E', ['I', 'AI!'], 'U', 'Z']
11 >>> vogais
12 ['A', 'E', ['I', 'O'], 'U', 'Z']
13 >>>

```

O método `deepcopy` separa completamente as duas estruturas a todos os níveis. A figura 6.7 ilustra isso mesmo.

### Mutabilidade e passagem de parâmetros

Todas estas situações derivam do facto de dois ou mais nomes partilharem (sub-)estruturas através das identidades (referências) dessas (sub-)estruturas. Quando o utilizador define uma função e mais tarde usa (chama) essa função, acontece um processo idêntico de associação de nomes a estruturas através da partilha da referência para essas estruturas. De novo um exemplo.

```

1 >>> def estraga(lst):
2 ... lst[2] = 'Ai!'
3 ... return lst
4 ...
5 >>> lista = [1,2,3]
6 >>> id(lista)
7 4421501800
8 >>> id(lst)
9 Traceback (most recent call last):
10 File "<stdin>", line 1, in <module>
11 NameError: name 'lst' is not defined
12 >>> estraga(lista)

```

---

<sup>2</sup>do inglês *deepcopy*.

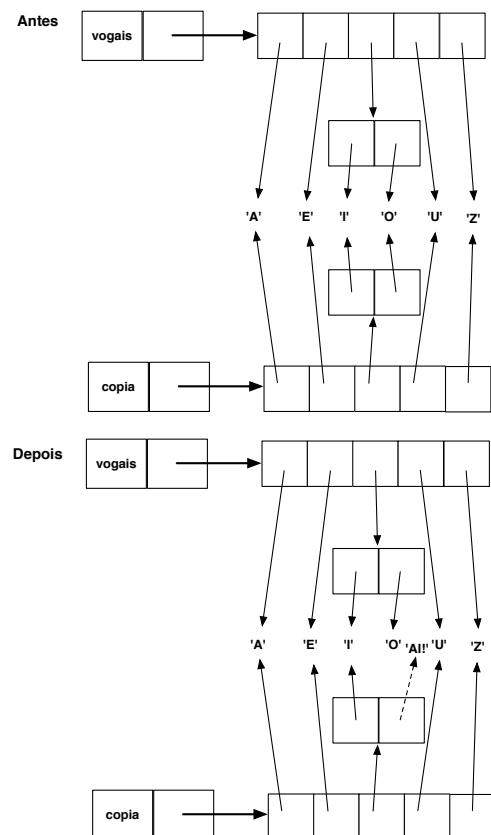


Figura 6.7: Sem problema...

```

13 [1, 2, 'Ai!']
14 >>> lista
15 [1, 2, 'Ai!']
16 >>> lst
17 Traceback (most recent call last):
18 File "<stdin>", line 1, in <module>
19 NameError: name 'lst' is not defined
20 >>>

```

Começamos por definir uma função a associar um objecto ao nome `lista`. Neste momento, no ambiente corrente, apenas são conhecidos os nomes **estraga** e **lista**. É por isso que quando tentamos saber a identidade de `lst` dá erro. De seguida, quando executamos a função **estraga** é criado um novo ambiente, ligado ao primeiro, no qual o nome `lst` tem a mesma identidade que **estraga**. A alteração que é feita durante a execução de **estraga** através do nome do parâmetro formal `lst`, repercute-se em `lista`. Esta alteração é permanente, isto é, mantém-se mesmo depois do programa terminar e o nome `lst` deixar novamente de ser conhecido. A figura 6.8 mostra a situação no momento em que é feita a chamada da função. Passamos a ter dois ambientes, ligados hierarquicamente, e em cada um deles os nomes que são conhecidos. Fica claro que toda a alteração a `lst` é feita no objecto partilhado. Por outro lado, quando a execução termina e o ambiente 2 desaparece, passando o ambiente 1 a ser o ambiente corrente, as alterações vão manter-se.

## Métodos

Para além das operações referidas e comuns às sequências, existem outras, próprias das listas, que mostramos na tabela 6.3.

As listagens que se apresentam de seguida ilustram a sua utilização. Refira-se desde já que alguns destes métodos modificam *in situ* os objectos. Deve-se também ter presente que alguns destes métodos devolvem um valor enquanto que outros não<sup>3</sup>.

```

1 >>> L=['eu','sou','eu', 'e','tu', 'es','tu','isto','digo','eu'
]
2 >>> L.count('eu')
3 3
4 >>> L.index('eu')
5 0
6 >>> L.append('tu')

```

---

<sup>3</sup>Na realidade devolvem o objecto `None`, e esse facto é fonte de muitas surpresas desagradáveis.

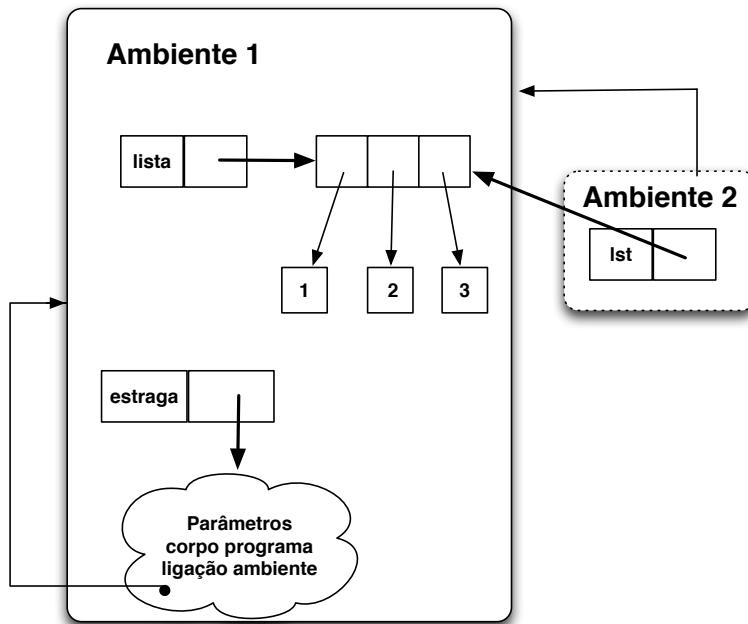


Figura 6.8: Ambientes, objectos e nomes

| Método                                         | Operação                                                 |
|------------------------------------------------|----------------------------------------------------------|
| <b>Não Modificam</b>                           |                                                          |
| <code>list.index(obj, i=0, j=len(list))</code> | Menor índice da ocorrência do objecto                    |
| <code>list.count(obj)</code>                   | Conta o número de vezes <i>obj</i> ocorre em <i>list</i> |
| <b>Modificam</b>                               |                                                          |
| <code>list.append(obj)</code>                  | Adiciona o objecto no fim da lista                       |
| <code>list.extend(seq)</code>                  | junta <i>seq</i> à <i>list</i>                           |
| <code>list.insert(index,obj)</code>            | Insere o objecto na posição dada pelo índice             |
| <code>list.remove(obj)</code>                  | Retira o objecto da lista                                |
| <code>list.pop(index)</code>                   | Retira o objecto da lista na posição dada pelo índice    |
| <code>list.reverse()</code>                    | inverte a lista <i>in situ</i>                           |
| <code>list.sort(cmp, key, reverse)</code>      | Ordena <i>in situ</i> a lista <i>list</i>                |

Tabela 6.3: Métodos Pré-Definidos para Listas

```

7 >>> L
8 ['eu', 'sou', 'eu', 'e', 'tu', 'es', 'tu', 'isto', 'digo', 'eu'
9 , 'tu']
10 >>> x=L.append(['ou', 'ele'])
11 ['eu', 'sou', 'eu', 'e', 'tu', 'es', 'tu', 'isto', 'digo', 'eu'
12 , 'tu', ['ou', 'ele']]
13 >>> x
14 None
15 >>>

```

O método `append` acrescentam um elemento no final da lista e devolve o objecto `None`. É por isso que a variável *x* fica associada a esse objecto!

```

1 >>> L=[1,2,3,4]
2 >>> L.insert(2,'a')
3 >>> L
4 [1, 2, 'a', 3, 4]
5 >>> L.remove('a')
6 >>> L
7 [1, 2, 3, 4]
8 >>> L.pop(2)
9 3
10 >>> L
11 [1, 2, 4]
12 >>> A=['abc','aaaa','z','b','ccc']
13 >>> A.sort(key=len)
14 >>> A
15 ['z', 'b', 'abc', 'ccc', 'aaaa']
16 >>> L.sort(reverse=True)
17 >>> L
18 [4, 2, 1]
19 >>>

```

Neste exemplo verifique-se que o método `pop` devolve um valor: o valor que é eliminado da lista.

```

1 >>> A='Estava mesmo a ver que ia dar asneira!'
2 >>> L=A.split(' ')
3 >>> L
4 ['Estava', 'mesmo', 'a', 'ver', 'que', 'ia', 'dar', 'asneira!']

```

```

5 >>> X=' '.join(L)
6 >>> X
7 'Estava mesmo a ver que ia dar asneira!'
8 >>> Y='abc'
9 >>> Z=list(Y)
10 >>> Z
11 ['a', 'b', 'c']
12 >>> W=str(Z)
13 >>> W
14 "['a', 'b', 'c']"
15 >>>

```

## Listas por compreensão

Suponhamos que queremos escrever um programa que nos permite construir uma lista com  $n$  números inteiros, escolhidos aleatoriamente no intervalo  $[1, 100]$ . Uma solução óbvia seria.

```

1 import random
2
3 def gera_lista(n):
4 lista = []
5 for i in range(n):
6 lista.append(random.randint(1,100))
7 return lista

```

Para problemas que seguem este padrão Python disponibiliza uma construção, denominada **lista por compreensão**, que nos permite escrever programas mais compactos e legíveis. Vejamos a solução para este caso:

```

1 def gera_lista_b(n):
2 return [random.randint(1,100) for i in range(n)]

```

A forma mais simples de uma lista por compreensão é:

`[ <expressão> for <item> in <iterável> ]`

É tão trivial que pode não fazer nada:

```

1 >>> [i for i in [1,2,3]]
2 [1, 2, 3]
3 >>>

```

Ou então coisas muito simples, como calcular o quadrado dos elementos numa lista e devolver a lista dos resultados.

```

1 >>> [i ** 2 for i in [1,2,3]]
2 [1, 4, 9]
3 >>>

```

Mas também pode ter associado um **filtro**, como formar uma lista com os elementos pares que aparecem noutra lista.

```

1 >>> [i for i in [1,2,3,4,5,6] if i % 2 == 0]
2 [2, 4, 6]
3 >>>

```

Como as listas por comprehensão apenas necessitam de um iterável (por exemplo, cadeia de caracteres, tuplo, lista), e como o seu resultado é uma lista, logo um objecto iterável, podemos ter listas por comprehensão **imbricadas**.

```

1 >>> [[i**2 for i in elem] for elem in [[1,2,3],[4,5,6]]]
2 [[1, 4, 9], [16, 25, 36]]
3 >>> [i**2 for elem in [[1,2,3],[4,5,6]] for i in elem]
4 [1, 4, 9, 16, 25, 36]
5 >>>

```

O exemplo acima mostra a importância do modo como se usa esta construção. A segunda forma permite resolver de modo elegante o problema de tornar plana uma lista de listas:

```

1 def aplana_lc(lista):
2 """Transforma uma lista de lista numa lista simples."""
3 res = [val for elem in lista for val in elem]
4 return res

```

Mais alguns exemplos de utilização:

```

1 >>> [i * j for i in [1,2,3] for j in ['a','b','c']]
2 ['a', 'b', 'c', 'aa', 'bb', 'cc', 'aaa', 'bbb', 'ccc']
3 >>> [i for elem in [[1,-2,3],[-4,5,-6]] for i in elem if i >
4 0]
5 [1, 3, 5]
6 >>>

```

Podemos usar as listas por comprehensão para determinar a transposta de uma matriz<sup>4</sup> de um modo muito elegante.

---

<sup>4</sup>A transposta de uma matriz é o que se obtém quando o elemento na posição (i,j) vai para a posição (j,i).

<sup>5</sup>Existe um modo de implementar a transposta que usa outras construções da linguagem. Será apresentado mais à frente.

```

1 def transposta_c(matriz):
2 """ Transposta de uma matriz."""
3 return [[matriz[j][i] for j in range(len(matriz))] for i in
4 range(len(matriz[0]))]

```

### Enumerate

Quando usada num ciclo **for**, uma lista pode ser percorrida seja pelo seu conteúdo seja pela posição dos seus elementos. A opção depende do problema. Existem no entanto situações em que nos interessa ter acesso ao elemento e à sua posição. quando isso acontece podemos usar a função pré-definida **enumerate**. Vejamos o que acontece quando o fazemos.

```

1 >>> lista = ['a', 'b', 'c']
2 >>> for i,v in enumerate(lista):
3 ... print(i,v)
4 ...
5 0 a
6 1 b
7 2 c
8 >>>
9
10 Se usarmos como segundo argumento um inteiro positivo a
 enumeração começa nesse valor. Por defeito essa valor
 é 0.

```

## Baleias

Regressemos ao problema das baleias e vejamos como podíamos resolver o nosso problema de análise de dados. Relembremos que temos guardado numa lista o número de baleias vistas numa certa zona ordenada por dias. Um primeiro problema consiste em obter uma caracterização dos dados através de medidas de centralidade. A média, isto é, o número médio de baleias avistadas, dado pela fórmula:

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

é um valor comum para a centralidade.

A solução informática socorre-se de um padrão de desenho comum para ciclos, o padrão **acumulador**, como se pode ver na listagem 6.5.

```

1 def media(lista):
2 """Calcula a média dos valores contidos na lista."""
3 soma = 0
4 for num in lista:
5 soma = soma + num
6
7 med = soma / len(lista)
8 return med

```

Listagem 6.5: Média

Esta solução não é única como se mostra na listagem 6.6<sup>6</sup>.

```

1 def media(lista):
2 """Calcula a média dos valores contidos na lista."""
3 return sum(lista) / len(lista)

```

Listagem 6.6: Média 2

Outro valor de centralidade é a mediana, que podemos obter por meio do programa simples que se lista em 6.7.

```

1 def mediana(lista):
2 """
3 Calcula a mediana: metade dos valores são inferiores,
4 metade é superior.
5 """
6 lista_aux = lista[:]
7 lista_aux.sort()
8 meio = len(lista_aux) / 2
9 if len(lista) % 2 == 0:
10 res = (lista_aux[meio-1] + lista_aux[meio]) / 2.0
11 else:
12 res = lista_aux[meio]
13 return res

```

Listagem 6.7: Mediana

Notar que temos o cuidado trabalhar sobre um cópia da lista e detemos de prever o caso de o número de elementos ser par ou ímpar.

Uma caracterização da nossa amostra não pode ficar completa se não medirmos também a dispersão. Por exemplo através da diferença entre o valor máximo e o valor mínimo:

---

<sup>6</sup>É possível encontrar ainda outras soluções. Uma delas faz uso da função `reduce` do módulo `functools`. Esta função aceita como argumento uma função que aplica ao segundo argumento. Trata-se de uma construção funcional da linguagem Python

```

1 def amplitude(lista):
2 """Diferença entre valores máximo e mínimo numa lista."""
3 return max(lista) - min(lista)

```

Mas a medida mais comum usada é o desvio padrão:

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{(n - 1)}}$$

que pode ser calculado por recurso ao código 6.8.

```

1 def desvio_padrao(lista):
2 """Calcula o desvio padrão."""
3 a_media = media(lista)
4 soma = 0
5 for elem in lista:
6 soma = soma + (elem - a_media) ** 2
7 desvio = math.sqrt(float(soma) / (len(lista) - 1))
8 return desvio

```

Listagem 6.8: Desvio Padrão

Façamos um pequeno desvio dos cálculos para exemplificar como os dados podiam ser **visualizados**. Para isso vamos usar o módulo `matplotlib` que nos fornece as funcionalidades necessárias.

```

1 import matplotlib.pyplot as plt
2
3 def mostra(lista):
4 """Gráfico simples de uma lista de valores."""
5 plt.xlabel('Dias')
6 plt.ylabel('Quantidade')
7 plt.title('Baleias')
8 plt.plot(lista)
9 plt.show()

```

Listagem 6.9: Visualização

Ao correr o programa obtemos a imagem da figura 6.9 (notar que o que se vê depende do valor concreto da lista).

### 6.3 Dicionários

Por muito que nos custe, há quem defende que a ciência do século XXI é a Biologia e não a Ciência dos Computadores. Descobertas recentes como a

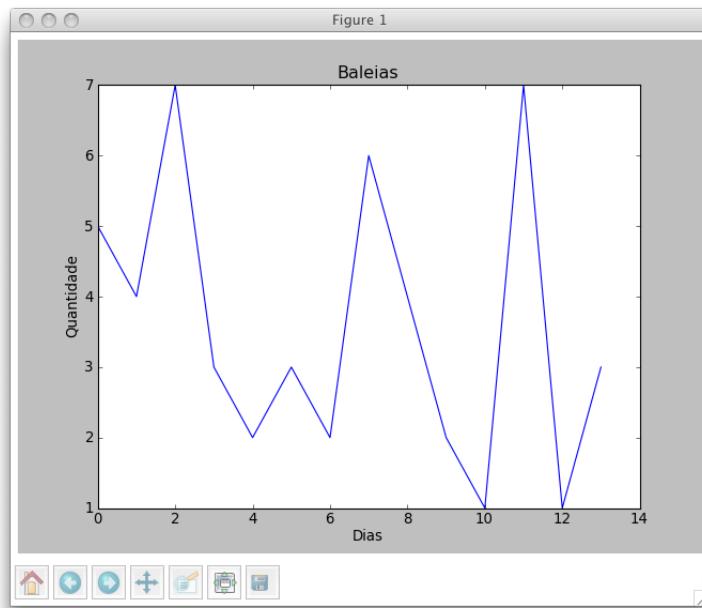


Figura 6.9: As Baleias dia a dia

sequênciação do genoma humano vieram dar esperança aos que acreditam que a descoberta da cura para doenças como o cancro, Alzheimer, Parkinson ou a Sida, está para breve. Sabemos que um dos processos mais relevantes para os seres vivos é a expressão dos genes contidos nos cromossomos que forma essa grande molécula que é designada por ADN. Em termos simples, o genoma humano pode ser identificado a uma longa sequência de quatro letras, que fazem parte do chamado alfabeto da vida: A,T,C e G. Cada letra designa uma base. São subsequências destas letras que formam os genes. A passagem dos genes às proteínas é um processo complexo mas que pode ser decomposto em duas fases: numa, designada por transcrição, há a transformação da molécula de ADN numa molécula de ARN. Esta última vê a base Timina ser substituída por outra chamada Uracil (letra U). Na segunda fase dá-se a tradução do ARN nas proteínas. Estas obtém-se graças ao código genético (ver figura 6.10): a cada três bases<sup>7</sup> corresponde um aminoácido .

São as sequências de aminoácidos codificados nos genes que originam as proteínas<sup>8</sup>. A figura 6.11 ilustra este processo.

Mas o que é que tudo isto pode ter que ver com a programação, sinto o

<sup>7</sup>Cada triplete é designado por codão

<sup>8</sup>Todo este processo é bem mais complexo, mas o leitor compreenderá que descrevê-lo não é o nosso objectivo neste texto sobre programação!

|                                         |   | Second letter of codon |     |              |     |     |      |              |      |
|-----------------------------------------|---|------------------------|-----|--------------|-----|-----|------|--------------|------|
|                                         |   | U                      |     | C            |     | A   |      | G            |      |
| First<br>letter of<br>codon<br>(5' end) | U | UUU                    | Phe | UCU          | Ser | UAU | Tyr  | UGU          | Cys  |
|                                         | U | UUC                    | Phe | UCC          | Ser | UAC | Tyr  | UGC          | Cys  |
|                                         | U | UU <b>A</b>            | Leu | U <b>C</b> A | Ser | UAA | Stop | UGA          | Stop |
|                                         | U | UUG                    | Leu | UCG          | Ser | UAG | Stop | UGG          | Trp  |
|                                         | C | CUU                    | Leu | CCU          | Pro | CAU | His  | CGU          | Arg  |
|                                         | C | CUC                    | Leu | CCC          | Pro | CAC | His  | CGC          | Arg  |
|                                         | C | CUA                    | Leu | CCA          | Pro | CAA | Gln  | CGA          | Arg  |
|                                         | C | CUG                    | Leu | CCG          | Pro | CAG | Gln  | CGG          | Arg  |
|                                         | A | AUU                    | Ile | ACU          | Thr | AAU | Asn  | AGU          | Ser  |
|                                         | A | AUC                    | Ile | ACC          | Thr | AAC | Asn  | AGC          | Ser  |
|                                         | A | AU <b>A</b>            | Ile | ACA          | Thr | AAA | Lys  | AGA          | Arg  |
|                                         | A | <b>AUG</b>             | Met | ACG          | Thr | AAG | Lys  | AGG          | Arg  |
|                                         | G | GUU                    | Val | GCU          | Ala | GAU | Asp  | GGU          | Gly  |
|                                         | G | GUC                    | Val | GCC          | Ala | GAC | Asp  | GGC          | Gly  |
|                                         | G | GU <b>A</b>            | Val | G <b>C</b> A | Ala | GAA | Glu  | G <b>G</b> A | Gly  |
|                                         | G | <b>GUG</b>             | Val | GC <b>G</b>  | Ala | GAG | Glu  | GGG          | Gly  |

Figura 6.10: Código Genético

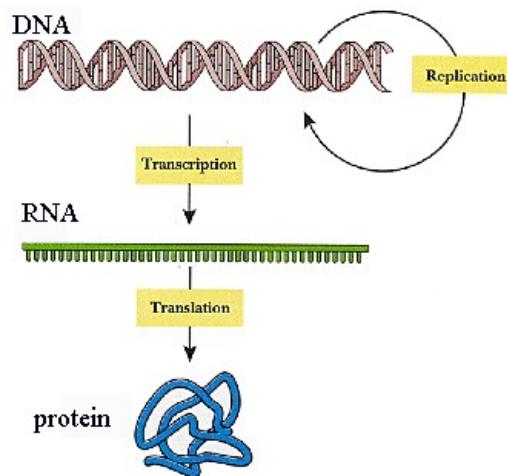


Figura 6.11: Expressão Genética

leitor a pensar? Bom, existem ficheiros com a sequência total ou parcial de ADN a partir do qual podemos extrair as proteínas correspondentes. Vamos resolver esse problema com a ajuda de um programa construído usando a idaia da programação descendente. Um programa muito simples, lê a cadeia de ADN, transcreve-a, ou seja substitui a Timina pela Uracil e depois traduz com base no código genético. Se representarmos o genoma por uma cadeia de caracteres uma primeira aproximação à solução será:

```

1 def exp_genes(adn):
2 """ A partir da sequência de ADN determina as proteínas.
3 """
4 arn = transcreve(adn)
5 amino = traduz(arn)
6 return amino

```

O passo seguinte é resolver os dois sub-problemas. Comecemos pela transcrição.

```

1 def transcreve(adn):
2 """ substitui T por U no adn."""
3 adn = adn.upper()
4 arn = adn.replace('T', 'U')
5 return arn

```

A tradução é um pouco mais complexa. Temos que identificar os codões e a partir do código genético gerar os aminoácidos correspondentes. Admitimos que primeiro obtemos os codões. Usamos uma lista para tal. A partir

dessa lista fabricamos a lista das proteínas

```

1 def traduz(arn):
2 """A partir do arn devolve a lista de proteínas."""
3 l_cod = codoes(arn)
4 l_amino = amino(l_cod)
5 return l_amino

1 def codoes(arn):
2 """
3 Devolve a lista de codões a partir de uma sequência.
4 A sequência é percorrida em grupos de três
5 enquanto é possível.
6 """
7 cod = []
8 for i in range(0, len(arn) - 2, 3):
9 cod.append(arn[i:i+3])
10 return cod

```

Já só falta obter os aminoácidos. Mas aqui coloca-se a questão de saber como representar o **código genético**. Do que conhecemos até aqui os objectos podem ser essencialmente do tipo numérico, cadeias de caracteres, tuplos ou listas. Os números e as cadeiras de caracteres estão naturalmente fora de questão. Ficam as listas. Uma hipótese simples, seria ter uma lista de listas em que cada uma delas seria um par de cadeias de caracteres. O primeiro elemento seria o representante do codão, e o segundo elemento o correspondente aminoácido.

```
1 código = [['UUA', 'Leu'], ['UCA', 'Ser'], \ldots]
```

Listagem 6.10: código genético

Mas esta representação não nos agrada. Por um lado, não é computacionalmente eficiente para o nosso problema. Em segundo lugar, não captura a ideia de mapeamento ou de correspondência entre dois objectos. É para resolver questões desta natureza que surgiram os **dicionários**. Um dicionário é uma colecções não ordenada, de pares de objectos, de comprimento variável, heterogénea, mutável, em que o acesso se faz por chave e não por posição. Um exemplo simples de dicionário:

```
1 bases = {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G':
 'Guanina'}
```

Neste exemplo encontramos as marcas sintácticas do dicionário que são os parênteses. Como nas listas, as vírgulas separam os elementos. Cada

## Definição

elemento tem duas componentes, sendo que o primeiro é a chave e o segundo o valor. A tabela 6.4 mostra outros exemplos.

| Literal                                                                       | Interpretação              |
|-------------------------------------------------------------------------------|----------------------------|
| <code>{'porto' : 'azul', 'sporting' : 'verde', 'benfica' : 'vermelho'}</code> | Dicionário simples         |
| <code>{}</code>                                                               | O dicionário ... vazio!    |
| <code>{'bolo_rei' : {'farinha' : 2, 'ovos' : 6, 'passas' : 0.5}}</code>       | Dicionário com Dicionários |
| <code>{1 : 'a', 'b' : 3.0 + 4j}</code>                                        | Dicionário heterogéneo     |
| <code>dicio = dict(zip(['praxe', 'lagartos'], [0, 5]))</code>                 | Outro modo de construir    |

Tabela 6.4: Literais para dicionários

## Construtor

O construtor do tipo dicionário chama-se `dict`. Pode ser usado sem argumentos, criando neste caso um dicionário vazio, ou com argumentos. A listagem seguinte ilustra diferentes formas de usar o construtor.

```

1 >>> d_1 = dict()
2 >>> d_1
3 {}
4 >>> d_2 = dict.fromkeys([1,2,3])
5 >>> d_2
6 {1: None, 2: None, 3: None}
7 >>> d_3 = dict(nome = 'ernesto', idade=60)
8 >>> d_3
9 {'idade': 60, 'nome': 'ernesto'}
10 >>> d_4 = dict(zip([1,2,3], ['a','b','c']))
11 >>> d_4
12 {1: 'a', 2: 'b', 3: 'c'}
13 >>> d_5 = dict.fromkeys([1,2,3],0)
14 >>> d_5
15 {1: 0, 2: 0, 3: 0}
16 >>>

```



### **zip**

Na listagem acima, na linha 10, introduzimos a função **zip**. A função devolve um iterável a partir de um ou mais iteráveis (e.g., listas,tuplos). Vejamos, com exemplos simples, como funciona.

```

1 >>> lista_1 = [1,2,3]
2 >>> lista_2 = [4,5,6]
3 >>> lista_3 = [7,8,9]
4 >>> junta = zip(lista_1,lista_2)
5 >>> junta
6 <zip object at 0x102cf41b8>
7 >>> next(junta)
8 (1, 4)
9 >>> next(junta)
10 (2, 5)
11 >>> next(junta)
12 (3, 6)
13 >>> next(junta)
14 Traceback (most recent call last):
15 File "<string>", line 1, in <fragment>
16 builtins.StopIteration:
17 >>> for elem in zip(lista_1,lista_3):
18 ... print(elem)
19
20 (1, 7)
21 (2, 8)
22 (3, 9)
23 >>> list(zip(lista_1,lista_2,lista_3))
24 [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
25 >>> list(zip(lista_1))
26 [(1,), (2,), (3,)]
```

Como se pode ver na linha 5 e 6, **zip** fabrica um objecto que é um iterável. Para conhecer os elementos em concreto temos várias alternativas. Podemos usar a função **next**, que nos vai fornecendo os elementos até não haver mais nenhum altura que é levantada uma excepção (linha 13); podemos usar o iterável num ciclo **for** (linha 17); podemos ainda envolver a chamada com a função **list**, como se mostra nas linhas 23 a 25. O utilizador pode, ele próprio definir iteráveis, como se ilustrará mais à frente.

Como sempre os dicionários, sendo objectos, têm identidade, valor e tipo.

```

1 >>> d={0:'zero',1:'um',2:'dois',3:'tres',4:'quatro',5:'cinco'}
2 >>> d
3 {0: 'zero', 1: 'um', 2: 'dois', 3: 'tres', 4: 'quatro', 5: 'cinco'}
4 >>> id(d)
5 50632576
6 >>> type(d)
7 <type 'dict'>
8 >>>
```

Vejamos agora as operações mais comuns.

| Nome               | Operador          | Significado |
|--------------------|-------------------|-------------|
| <i>Indexação</i>   | [< chave >]       | Acede       |
| <i>Pertença</i>    | <i>in, not in</i> | Testa       |
| <i>Comprimento</i> | <i>len</i>        | Quantifica  |

Tabela 6.5: Operações sobre Dicionários



### Listas e dicionários

Comparando com as listas, notar que o acesso nos dicionários se faz por chave, que a pertença tem uma operação própria. Não existe a operação de fatiamento. Se pensarmos bem ela não tem sentido, pois os dicionários não têm ordem. Num dicionário os valores podem ser de qualquer tipo. O mesmo já não acontece com as chaves que têm que ser de tipo imutável.

### Exemplo de utilização

```

1 >>> d
2 {'A': 'Adenina', 'T': 'Timina'}
3 >>> d['A']
4 'Adenina'
5 >>> d['C']
6 Traceback (most recent call last):
7 File "<stdin>", line 1, in <module>
8 KeyError: 'C'
9 >>>
```

Tentar obter um valor através de uma chave inexistente dá erro. A heterogeneidade existe nas chaves e nos valores.

```

1 >>> dicio={'A':'Adenina',2:'dois',4.6: 'quatro ponto seis':
2 5+4.5j: 'complexo'}
3 >>> dicio
4 {'A': 'Adenina', 'dois': 2, 'bases': ['a', 't', 'c', 'g']}
5 >>>

```

Sendo mutável, podemos não apenas consultar como também modificar o valor sem alterar a identidade.

```

1 >>> dicio
2 {'A': 'Adenina', 3: 'THREE', 2: 'TWO', 'bases': ['A', 'U', 'C',
3 , 'G']}
4 >>> dicio.setdefault('T','Timina')
5 'Timina'
6 >>> dicio
7 {'A': 'Adenina', 3: 'THREE', 2: 'TWO', 'bases': ['A', 'U', 'C',
8 , 'G'], 'T': 'Timina'}
9 >>> dicio.setdefault(2, 'dois')
10 'TWO'
11 >>> dicio
12 {'A': 'Adenina', 'bases': ['A', 'U', 'C', 'G'], 2: 'TWO', 3: 'THREE',
13 , 'T': 'Timina'}
14 >>> dicio_1 = {'A':'Adenina'}
15 >>> dicio_2 = {'C':'Citosina','G': 'Guanina','T':'Timina'}
16 >>> dicio_1.update(dicio_2)
17 >>> dicio_1
18 {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'
19 }
20 >>> dicio_2
21 {'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'}
22 >>> del dicio_1['C']
23 >>> dicio_1
24 {'A': 'Adenina', 'T': 'Timina', 'G': 'Guanina'}
25 >>> del dicio_2
26 >>> dicio_2
27 Traceback (most recent call last):
28 File "<string>", line 1, in <fragment>
29 NameError: name 'dicio_2' is not defined
30 >>> dicio_1.clear()
31 >>> dicio_1
32 {}
33 >>> dicio_2 = {'C':'Citosina','G': 'Guanina','T':'Timina'}

```

```

30 >>> dicio_2.pop('T')
31 'Timina'
32 >>> dicio_2
33 {'C': 'Citosina', 'G': 'Guanina'}
34 >>> dicio_2.pop('A')
35 Traceback (most recent call last):
36 File "<string>", line 1, in <fragment>
37 KeyError: 'A'
38 >>> dicio_2.pop('A', 'Oops!')
39 'Oops!'
40 >>>
```

Alguns métodos característicos dos dicionários.

```

1 >>> dicio_2
2 {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'
3 '}
4 >>> dicio_1 = dicio_2.copy()
5 >>> dicio_1
6 {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'
7 '}
8 >>> dicio_1.items()
9 [('A', 'Adenina'), ('C', 'Citosina'), ('T', 'Timina'), ('G', 'Guanina')]
10 >>> dicio_1.keys()
11 ['A', 'C', 'T', 'G']
12 >>> dicio_1.values()
13 ['Adenina', 'Citosina', 'Timina', 'Guanina']
14 >>> 'T' in dicio_1
15 True
16 >>> 'X' in dicio_1
17 False
18 >>> dicio_1.get('T')
19 'Timina'
20 >>>
```

Podemos também ter operações de repetição sobre objectos do tipo dicionário. O ciclo pode ser controlado pelas chaves, pelos valores ou pelos elementos. Nunca pela ordem!

```

1 >>> d
```

```

2 { 'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'
 }
3 >>> for c in d.keys():
4 ... print(c)
5 ...
6 A
7 C
8 T
9 G
10 >>> for v in d.values():
11 ... print(v)
12 ...
13 Adenina
14 Citosina
15 Timina
16 Guanina
17 >>> for c,v in d.items():
18 ... print('d[',c,'] = ',v)
19 ...
20 d[A] = Adenina
21 d[C] = Citosina
22 d[T] = Timina
23 d[G] = Guanina
24 >>>

```

### Métodos

Em resumo, temos um conjunto de métodos, uns de consulta outros de modificação, como a tabela 6.6 ilustra.

### Exemplo de Contagem de Bases

Já aqui referimos que o ADN pode ser descrito por uma cadeia de letras de um alfabeto com apenas quatro elementos: A, T, C e G. Existem bases de dados que descrevem partes do ADN. Quando existe ambiguidade (por exemplo, é uma Adenina ou uma Timina?) usam-se outras letras. A figura 6.12 ilustra a situação e mostra os códigos. Por exemplo, usamos a letra N quando a ambiguidade é total.

Um problema importante é o de saber quantas bases existem de cada tipo. Uma solução, admitindo que a sequência está codificada através de uma cadeia de caracteres seria:

```

1 >>> seq = "TKKAMRCRAATARKWC"

```

|   | A | T | C | G |
|---|---|---|---|---|
| A | ■ |   |   |   |
| B |   | ■ | ■ | ■ |
| C |   |   | ■ |   |
| D | ■ | ■ |   | ■ |
| G |   |   |   | ■ |
| H | ■ | ■ | ■ |   |
| K |   | ■ |   | ■ |
| M | ■ |   | ■ |   |
| N | ■ | ■ | ■ | ■ |
| R | ■ |   |   | ■ |
| S |   |   | ■ | ■ |
| T |   | ■ |   |   |
| V | ■ |   | ■ | ■ |
| Y |   | ■ | ■ |   |
| W | ■ | ■ |   |   |

Figura 6.12: Códificação das Bases

| Método                                   | Operação                                                      |
|------------------------------------------|---------------------------------------------------------------|
| <b>Não Modificam</b>                     |                                                               |
| <i>dict.copy()</i>                       | Devolve uma cópia do <i>dict</i>                              |
| <i>key in dict</i>                       | Verifica a existência da chave <i>key</i> em <i>dict</i>      |
| <i>dict.items()</i>                      | Devolve um iterável de pares (chave,valor) de <i>dict</i>     |
| <i>dict.keys()</i>                       | Devolve um iterável chaves de <i>dict</i>                     |
| <i>dict.values()</i>                     | Devolve um iterável de valores de <i>dict</i>                 |
| <i>dict.get(key,default=None)</i>        | Devolve o valor caso exista senão devolve <i>default</i>      |
| <b>Modificam</b>                         |                                                               |
| <i>dict.clear()</i>                      | Retira todos os elementos de <i>dict</i>                      |
| <i>dict.pop(key,default=None)</i>        | Retira e devolve o elemento de <i>key</i>                     |
| <i>dict.update(dict2)</i>                | Adiciona os pares (chave,valor) de <i>dict2</i> a <i>dict</i> |
| <i>dict.setdefault(key,default=None)</i> | Como <i>get</i> mas actualiza o par com <i>key:default</i>    |
| <i>del dict[key]</i>                     | Retira o item associado a <i>key</i>                          |

Tabela 6.6: Métodos Pré-Definidos para Dicionários

```

2 >>> A = seq.count("A")
3 >>> B = seq.count("B")
4 >>> C = seq.count("C")
5 >>> D = seq.count("D")
6 >>> G = seq.count("G")
7 >>> H = seq.count("H")
8 >>> K = seq.count("K")
9 >>> M = seq.count("M")
10 >>> N = seq.count("N")
11 >>> R = seq.count("R")
12 >>> S = seq.count("S")
13 >>> T = seq.count("T")
14 >>> V = seq.count("V")
15 >>> W = seq.count("W")
16 >>> Y = seq.count("Y")
17 >>> print("A =", A, "B =", B, "C =", C, "D =", D, "G =", G, "
18 H =", H, "K =", K, "M =", M, "N =", N, "R =", R, "S =", S,
19 "T =", T, "V =", V, "W =", W, "Y =", Y)
A = 4 B = 0 C = 2 D = 0 G = 0 H = 0 K = 3 M = 1 N = 0 R = 3 S
= 0
T = 2 V = 0 W = 1 Y = 0
>>>

```

Mas admitamos que é uma solução muito ... feia! Como já sabemos que os dicionários estabelecem correspondências podemos usar um. A chave é o código da base, o valor o número de ocorrências.

```

1 >>> seq = "TKKAMRCRAATARKWC"
2 >>> counts = {}
3 >>> counts["A"] = seq.count("A")
4 >>> counts["B"] = seq.count("B")
5 >>> counts["C"] = seq.count("C")
6 >>> counts["D"] = seq.count("D")
7 >>> counts["G"] = seq.count("G")
8 >>> counts["H"] = seq.count("H")
9 >>> counts["K"] = seq.count("K")
10 >>> counts["M"] = seq.count("M")
11 >>> counts["N"] = seq.count("N")
12 >>> counts["R"] = seq.count("R")
13 >>> counts["S"] = seq.count("S")
14 >>> counts["T"] = seq.count("T")
15 >>> counts["V"] = seq.count("V")
16 >>> counts["W"] = seq.count("W")
17 >>> counts["Y"] = seq.count("Y")
18 >>> print(counts)
19 {'A': 4, 'C': 2, 'B': 0, 'D': 0, 'G': 0, 'H': 0, 'K': 3, 'M':
 1, 'N': 0, 'S': 0, 'R': 3, 'T': 2, 'W': 1, 'V': 0, 'Y': 0}
20 >>>

```

Alterámos a representação mas não melhorámos a estética. Simplificando:

```

1 >>> seq = "TKKAMRCRAATARKWC"
2 >>> counts = {}
3 >>> for letter in "ABCDGHKMNRSTVWY":
4 ... counts[letter] = seq.count(letter)
5 ...
6 >>> print(counts)
7 {'A': 4, 'C': 2, 'B': 0, 'D': 0, 'G': 0, 'H': 0, 'K': 3, 'M':
 1, 'N': 0, 'S': 0, 'R': 3, 'T': 2, 'W': 1, 'V': 0, 'Y': 0}
8 >>>

```

Mas ainda podemos fazer melhor eliminando todos os casos em que o valor é 0.

```

1 >>> seq = "TKKAMRCRAATARKWC"
2 >>> counts = {}
3 >>> for base in seq:

```

```

4 ... if base not in counts:
5 ... n = 0
6 ... else:
7 ... n = counts[base]
8 ... counts[base] = n + 1
9 ...
10 >>> print(counts)
11 {'A': 4, 'C': 2, 'K': 3, 'M': 1, 'R': 3, 'T': 2, 'W': 1}
12 >>>

```

Mas podemos chegar a uma excelente solução recorrendo ao método dos dicionários `get`.

```

1 >>> seq = "TKKAMRCRAATARKWC"
2 >>> counts = {}
3 >>> for base in seq:
4 ... counts[base] = counts.get(base, 0) + 1
5 ...
6 >>> print(counts)
7 {'A': 4, 'C': 2, 'K': 3, 'M': 1, 'R': 3, 'T': 2, 'W': 1}
8 >>>

```

## Expressão Genética

Estamos agora em condições de encerrar a questão da expressão genética. Para tal representamos o código genético através de um ... dicionário.

```

1 def amino(l_codoes):
2 """Converte uma lista de codões na sequência de
3 aminoácidos"""
4 amino={}
5 'UUU': 'F', 'UUC': 'F', 'UUA': 'L', 'UUG': 'L',
6 'UCU': 'S', 'UCC': 'S', 'UCA': 'S', 'UCG': 'S',
7 'UAU': 'Y', 'UAC': 'Y', 'UAA': '*', 'UAG': '*',
8 'UGU': 'C', 'UGC': 'C', 'UGA': '*', 'UGG': 'W',
9 'CUU': 'L', 'CUC': 'L', 'CUA': 'L', 'CUG': 'L',
10 'CCU': 'P', 'CCC': 'P', 'CCA': 'P', 'CCG': 'P',
11 'CAU': 'H', 'CAC': 'H', 'CAA': 'Q', 'CAG': 'Q',
12 'CGU': 'R', 'CGC': 'R', 'CGA': 'R', 'CGG': 'R',
13 'AUU': 'I', 'AUC': 'I', 'AUA': 'I', 'AUG': 'M',
14 'ACU': 'T', 'ACC': 'T', 'ACA': 'T', 'ACG': 'T',
15 'AAU': 'N', 'AAC': 'N', 'AAA': 'K', 'AAG': 'K',

```

```

15 'AGU': 'S', 'AGC': 'S', 'AGA': 'R', 'AGG': 'R',
16 'GUU': 'V', 'GUC': 'V', 'GUA': 'V', 'GUG': 'V',
17 'GCU': 'A', 'GCC': 'A', 'GCA': 'A', 'GCG': 'A',
18 'GAU': 'D', 'GAC': 'D', 'GAA': 'E', 'GAG': 'E',
19 'GGU': 'G', 'GGC': 'G', 'GGA': 'G', 'GGG': 'G'
20 }
21 return ''.join([amino[codao] for codao in l_codoes])

```

## Segurança

Hoje cada vez mais a questão da segurança dos equipamentos é crucial. Um modo de os proteger é usar uma combinação de nome de utilizador e código de acesso. Esta correspondência pode ser representada por um dicionário.

```

1 def palavra_chave(nome_utilizador, segredo):
2 """
3 Verifica a palavra chave de um utilizador.
4 """
5 codigos = {'ernesto':'toto','patricia':'hello31','ana':''
6 'gato56'}
7 if nome_utilizador not in codigos :
8 return 'Não o conheço!'
9
10 codigo_correcto = codigos[nome_utilizador]
11 if segredo == codigo_correcto:
12 return 'Bem vindo!'
13 else:
14 return 'Enganou-se'

```

## Baleias

Regressemos ao nosso exemplo das baleias para saber, face aos resultados observados qual é o valor mais frequente, isto é, qual é o valor da **moda**. A solução passa por representar os dados da frequência por meio de um dicionário, sendo que a chave é a quantidade observada e o valor é o número de vezes que essa quantidade ocorreu.

```

1 def frequencia(valores):
2 """ Cria um dicionário com a frequência da ocorrência
3 dos valores. """
4 freq = dict()

```

```

4 for item in valores:
5 freq[item] = freq.get(item,0) + 1
6 return freq

```

Agora o cálculo da moda está facilitado.

```

1 def moda(valores):
2 """ Calcula a moda de um conjunto de valores. """
3 dicio_freq = frequencia(valores)
4 # -- porque pode haver mais do que uma moda...
5 lista_valores = list(dicio_freq.values())
6 maxima_freq = max(lista_valores)
7 # -- contrói resultado
8 lista_modas = list()
9 for chave in dicio_freq:
10 if dicio_freq[chave] == maxima_freq:
11 lista_modas.append(chave)
12 return lista_modas

```

## 6.4 Mais exemplos

### Construir dicionários

Existem muitos modos de **construir dicionários**. Podemos, por exemplo, começar com um dicionário vazio e depois ir acrescentando pares (chave : valor), ou podemos indicar explicitamente os pares na inicialização. Neste último caso as sintaxes possíveis são variadas, como foi ilustrado anteriormente. O que queremos ilustrar aqui é outra possibilidade: temos uma lista que contém **alternadamente** as chaves e os valores correspondentes e pretendemos, a partir dessa lista, construir o correspondente dicionário. Estamos a admitir que não existem chaves repetidas na lista.

A solução trivial será:

```

1 def dicio_lista(lista_chaves_valores):
2 """Constrói um dicionário a partir de uma lista com chaves
3 e valores
4 alternados."""
5 dicio = {}
6 for i in range(len(lista_chaves_valores)/2):
7 dicio[lista_chaves_valores[2*i]] =
8 lista_chaves_valores[2*i + 1]
9 return dicio

```

Mas podemos fazer melhor usando o método `zip` e o construtor para dicionários `dict`:

```

1 def dicio_lista_b(lista_chaves_valores):
2 """Constrói um dicionário a partir de uma lista com chaves
3 e valores
4 alternados."""
5 return dict(zip(lista_chaves_valores[::2],
6 lista_chaves_valores[1::2]))

```

Nas duas soluções tivemos que lidar com a separação dos elementos em chaves e valores. As chaves estão nas posições pares e os valores nas posições ímpares da lista. Claramente preferimos a segunda!

## Equívocos

Existem dois métodos que se aplicam a dicionários que, por serem semelhantes, levam a alguns erros. São `setdefault` e `get`. Para explicitar as diferenças vamos considerar um exemplo concreto. Admitamos que estamos a fazer o índice de um livro, isto é queremos associar a cada palavra a indicação das páginas do livro onde essa palavra ocorre. Vamos usar um dicionário para guardar esta associação. Suponhamos que queremos implementar o método que acrescenta uma palavra (e a respectiva página) ao índice. Uma solução trivial seria:

```

1 def add_palavra_triv(indice,palavra,pagina):
2 if palavra in indice:
3 indice[palavra].append(pagina)
4 else:
5 indice[palavra] = [pagina]

```

Note-se que não é preciso fazer `if palavra in indice.keys()`. Atente-se ainda no modo distinto como temos que tratar o caso de a palavra estar, ou não, no dicionário. O leitor mais conhedor de Python pode saber que é possível fazer tudo sem precisar do teste, e achar que usar o `setdefault` ou o `get` é o mesmo. Propõe por isso duas soluções alternativas:

```

1 def add_palavra_get(indice,palavra, pagina):
2 indice.get(palavra,[]).append(pagina)
3
4 def add_palavra_set(indice,palavra, pagina):
5 indice.setdefault(palavra,[]).append(pagina)

```

Mas, para sua surpresa, se executar o código seguinte o resultado não é bem o que estava à espera.

```

1 >>> dic = { 'eu':[1,5,7], 'tu': [2,4,7]}
2 >>> add_palavra_get(dic, 'eu', 10)
3 >>> add_palavra_get(dic, 'ele', 20)
4 >>> print(dic)
5 { 'eu': [1, 5, 7, 10], 'tu': [2, 4, 7]}
6 >>> add_palavra_set(dic,'tu', 8)
7 >>> add_palavra_set(dic,'ele', 33)
8 >>> print(dic)
9 { 'eu': [1, 5, 7, 10], 'tu': [2, 4, 7, 8], 'ele': [33]}

```

Fica claro que no caso em que a chave **não** está no dicionário os dois métodos são diferentes: enquanto **setdefault** acrescenta o novo par, o mesmo não acontece com o método **get**.

## Árvores Genealógicas

As árvores genealógicas têm informações sobre famílias. Admitamos um caso simples em que, num **dicionário** colocamos pares (progenitor,lista\_descendentes). Por exemplo:

```

1 dic = {'ernesto':['carlos', 'jorge', 'ana'], 'carlos':['ricardo',
 ', 'joana'], 'joana': [], 'jorge':['carla', 'francisca'], 'ana':[]}

```

Listagem 6.11: Genealogia I

São muitas as questões que podemos colocar. A mais simples será a de saber quais os descendentes directos de uma dada pessoa:

```

1 def filhos(dicio,progenitor):
2 """ lista dos filhos."""
3 res= dicio.get(progenitor,None)
4 return res

```

Listagem 6.12: Genealogia II

Com base neste modelo ultra-simplificado vamos tentar resolver algumas questões.

## Sumário

Neste capítulo tratámos de dois tipos especiais de objectos que são colecções: as listas e os dicionários. Vimos as suas características, os métodos que se lhes aplicam. Discutimos ainda as consequências de listas e dicionários serem objectos mutáveis. Foram dados alguns exemplos de aplicação.

## Teste os seus conhecimentos

- O que distingue as listas dos dicionários.
- A operação de fatiamento das listas distingue-se da mesma operação para cadeias de caracteres.
- Que tipo de objectos pode ser usado como chave de um dicionário. Quais as razões para haver restrições.
- Porque não existe operação de fatiamento em dicionários.
- Que implicações existem pelo facto de os objectos destes tipos serem mutáveis.
- O que entende por cópia profunda de uma estrutura. Em que situações é útil.
- Existem métodos que modificam e que não modificam os objectos. O que os distingue.
- O que são listas por compreensão.
- O que permite o uso de `enumerate` num ciclo.
- O que permite a função `zip`
- Porque é que em certas situações temos que envolver a chamada da função `zip` com a função `list`.

## Exercícios

### Exercício 6.1 F

Dada uma lista com as idades dos alunos de uma turma, desenvolva um programa para cada um dos seguintes problemas, usando apenas as operações acima referidas.

1. Mostre o número de idades;
2. Exiba todos as idades da lista;
3. Exiba todas as idades na ordem inversa à da lista fornecida;
4. Exiba todas as idades excepto a primeira e a última da lista;

5. Mostre a idade menor e a maior;
6. Calcule e mostre a soma dos valores na lista;
7. Calcule e mostre o número de elementos de valor abaixo de um outro, dado como referência.
8. Verifique se existe algum aluno com 17 anos;

**Exercício 6.2 F** Desenvolva um programa que dada uma lista de números devolva a **soma** dos seus números pares e a **soma** dos seus números ímpares. A listagem 6.14 ilustra o que se pretende.

```

1 >>> lst = [1,4,7,9,3,2,8,5,6]
2 >>> # ---> Aqui o seu programa de nome pares_impares.
3 >>> pares_impares(lst)
4 (20, 25)
5 >>>

```

Listagem 6.13: Pares e Ímpares

**Exercício 6.3** Desenvolva um programa que dadas duas listas devolve uma terceira formada pelos elementos das primeiras dispostos de modo alternado. Começa com a primeira lista.. A listagem 6.14 ilustra o que se pretende.

```

1 >>> l1 = [1,2,3]
2 >>> l2 = ['a','b','c']
3 >>> # ---> Aqui o seu programa de nome alterna.
4 >>> alterna(l1, l2)
5 [1,'a',2,'b',3,'c']
6 >>>

```

Listagem 6.14: alterna

**Exercício 6.4** Desenvolva um programa que, dados um elemento numérico e uma lista de números, determina **quantos** elementos da lista são **menores** do que o número. A listagem 6.15 ilustra o que se pretende.

```

1 >>> # ---> Aqui o seu programa de nome conta_menores.
2 >>> conta_menores(5,[2,8,6,5,3,2])
3 3
4 >>>

```

Listagem 6.15: contar menores

**Exercício 6.5 F** Módulo random

Suponha que tem dois dados numerados de 1 a 6. Vai lançá-los sucessivas vezes e guardar os resultados (a soma). Escreva um programa que mostre os resultados dos sucessivos lançamentos e determine a percentagem de vezes em que saiu uma soma par.

**Exercício 6.6 M**

Desenvolva um programa que receba uma lista de números e calcule a soma cumulativa, i.e., o programa deve retornar uma nova lista em que o elemento de ordem **i** é a soma dos primeiros **i+1** elementos da lista original. Exemplo: para [1,2,3] deve devolver [1,3,6]

**Exercício 6.7 M**

Uma imagem a preto e branco pode ser guardada como uma lista de listas. Cada elemento representa uma linha da imagem. O preto é representado por 1 e o branco por zero. Por exemplo, `[[0,1,0],[1,1,1],[0,1,0]]` representa uma cruz. Escreva um programa que, dada uma imagem produz o seu **negativo**, isto é uma nova imagem em que o branco passa a preto e o preto a branco.

**Exercício 6.8 MD**

Uma imagem a preto e branco pode ser guardada como uma lista de listas. Cada elemento representa uma linha da imagem. O preto é representado por 1 e o branco por zero. Por exemplo, `[[0,1,0],[1,1,1],[0,1,0]]` representa uma cruz. Escreva um programa que, dada uma imagem a roda 90° no sentido dos ponteiros do relógio.

**Exercício 6.9** Suponha que está perdido no meio de uma cidade e não tem GPS para se orientar. Pergunta a um transeunte como pode chegar ao seu destino. Como a cidade é geométrica a resposta é fácil. Recebemos uma sequência de indicações do tipo **vira à esquerda (E)**, **depois avança (A)**, **depois roda à direita (D)**, **depois avança (A)**, **depois avança de novo (A)**, **depois recua (R)**, .... Usando o módulo **turtle** desenvolva um programa, que quando executado, **simule** com a tartaruga os seus movimentos quando esta executa os comandos recebidos. A imagem ?? mostra o que acontece quando manda correr o programa geral **main\_tarta()**. Por conveniência de visualização marcámos o inicio e o fim do percurso com pontos, verde e vermelho, respectivamente. O seu programa chama-se, no código abaixo, **navega**.

```

1 def main_tarta():
2 tartaruga = cTurtle.Turtle()
3 tartaruga.setheading(0)
4 comandos = 'ADAEAAERDAAEARA'
5 navega(comandos, tartaruga)
6 tartaruga.exitOnClick()

```

**Exercício 6.10** Desenvolva um programa que dado um texto, isto é uma cadeia de caracteres, construa um **dicionário**, com as **posições** em que ocorrem as vogais. Note que os caracteres podem ser maiúsculos ou minúsculos. A listagem 6.16 ilustra o pretendido.

```

1 >>> print posicoes('agora e que vao ser elas, Ai, Ai!')
2 Evaluating mt3.py
3 {'a': [0, 4, 13, 22], 'A': [26, 30], 'e': [6, 10, 17, 20], 'i':
 : [27, 31], 'o': [2, 14], 'u': [9]}
4 >>>

```

Listagem 6.16: Índices das ocorrências

### Exercício 6.11 F

Crie o seguinte dicionário de linguagens de programação e respectivos autores:

```
autor = {"php": "Rasmus Lerdorf", "perl": "Larry Wall", "tcl": "John Ousterhout",
"awk": "Brian Kernighan", "java": "James Gosling", "parrot": "Simon Cozens",
"python": "Guido van Rossum", "xpto": "zxcv"}.
```

- Acrescente um elemento ao dicionário
- Altere o autor do python para "Guido van Rossum".
- Remova o elemento com chave "xpto".
- Quantos elementos tem o dicionário?
- Existe uma entrada para "c++"?

### Exercício 6.12 F

Suponha que tem uma pequena loja de vender fruta, e que construiu uma pequena base de dados para fazer a gestão do stock da fruta. Para cada tipo de fruta tem a indicação da quantidade, em quilos, que tem para venda. Admita que inicialmente tem esses elementos em duas listas. Uma com o nome das frutas e outra com as quantidades. A partir desses dados crie o respectivo dicionário.

**Exercício 6.13 M**

Suponha que quer tornar a sua gestão da loja de fruta mais eficiente. Para isso, para cada tipo de fruta associa a informação da quantidade de fruta que **você** comprou, do preço de compra por quilo, da quantidade que tem em stock e do preço de venda por quilo. Como guardaria esta informação? Escreva programas para cada uma destas questões:

- a) Qual o lucro já obtido?
- b) Qual a fruta mais cara?

**Exercicio 6.14 M**

Vamos querer implementar um conversor de datas. Para tal vamos supor que temos guardado num dicionário a relação entre números e dias da semana, `dias_semana={1:'Domingo', 2:'Segunda-Feira', 3:'Terça-feira', ..., 7:'Sábado'}`, e outro para os meses do ano `meses_ano = {1: 'Janeiro', 2:'Fevereiro', ..., 12: 'Dezembro'}`. O formato DS/DM/M/A é um dos que é possível utilizar para representar uma data. Neste formato *DS* corresponde ao valor inteiro do dia da semana (0 a 7), *DM* corresponde valor inteiro do mês e *A* corresponde ao ano. Faça uma função que receba os dois dicionários criados anteriormente e uma cadeia de caracteres com a data no formato DS/DM/M/A, e apresente essa data por extenso.

Exemplo:

Para a data: "4/5/2006"

Quarta-feira, 5 de Junho de 2006

**Exercício 6.15 M**

Escreva uma função que recebe um dicionário, em que cada elemento é formado pela chave, o número do BI de uma pessoa, e o valor contém informação sobre o sexo, idade, altura e peso, e devolve um novo dicionário com os rácios de metabolismo basal dessas pessoas. Tenha em conta que o rácio de metabolismo basal é dado por:  $66 + (6.3 * \text{peso}) + (12.9 * \text{altura})$

- $(6.8 * \text{idade})$  no caso de ser homem, e  $655 + (4.3 * \text{peso}) + (4.7 * \text{altura})$
- $(4.7 * \text{idade})$  no caso de ser mulher.

### Exercício 6.16 M

Escreva uma função que receba um dicionário, em que cada elemento associa o número do Bilhete de Identidade de uma pessoa (chave), com informação sobre a sua altura e peso, e devolva o mesmo dicionário onde foi acrescentado o índice de massa corporal de cada pessoa. O índice de massa corporal de uma pessoa é calculado dividindo o seu peso pelo quadrado da sua altura.

### Exercício 6.17 D

Faça um programa que inverta um dicionário, i.e., que coloque os valores como chaves e as chaves como valores. Deverá ter em atenção que chaves diferentes podem ter o mesmo valor.

Exemplo:

Input:{'joao':10,'pedro':18, 'tiago':13,'luis':18}  
Output:{18: ['luis', 'pedro'], 10: ['joao'], 13: ['tiago']}.

### Exercício 6.18 F

Dada umas árvore genealógica, organizada como um dicionário, escreva um programa que determine se duas pessoas são **irmãos/irmãs**.

### Exercício 6.19 M

Dada umas árvore genealógica, organizada como um dicionário, escreva um programa que determine os **netos** de uma pessoa, caso existam.

### Exercício 6.20 M

Dada umas árvore genealógica, organizada como um dicionário, escreva um programa que determine o **progenitor** de uma pessoa.

### Exercício 6.21 M

Dada umas árvore genealógica, organizada como um dicionário, escreva um programa que determine o **avô/avó** de uma pessoa, caso exista.

# Ficheiros

## Objectivos

- ✓ Introduzir o conceito de ficheiro
- ✓ Exercitar os conceitos de leitura, escrita e navegação
- ✓ Introduzir a instrução `with`
- ✓ Introduzir os módulos `csv` e `urllib.request`
- ✓ Introduzir o tipo `bytes`

### 7.1 Generalidades

Vamos supor que você é meteorologista e que andou a guardar informação relativa à temperatura e à pluviosidade em diversos locais, ao longo dos meses. Por exemplo, tem essa informação para várias cidades de Portugal, e agora chegou a altura de fazer um estudo comparativo. Esta é uma situação em que a informação teve que ser guardada de forma permanente, para mais tarde ser acedida, trabalhada e, eventualmente, os resultados da sua análise serem também eles guardados. É para isso que são usados os ficheiros. Chamamos **ficheiros** aos locais onde guardamos de forma permanente [informação<sup>1</sup>](#). Existem dois grandes tipos de ficheiros: de texto e binários. Na ausência de informação em contrário o que vamos descrever aplica-se aos

---

<sup>1</sup>Como é evidente os programas que escrevemos são eles próprios armazenados em ficheiros.

ficheiros de texto. Um ficheiro de texto não é mais do que uma (eventualmente muito longa) cadeia de caracteres.

## Abertura de um ficheiro

Posto isto vamos começar por ver o que são os ficheiros e como se manipulam. A primeira coisa a referir é a operação de abertura do ficheiro através de uma função pré-definida e cuja sintaxe é `open(nome, modo)`. Essa operação devolve um objecto de tipo *ficheiro*, instância da classe `_io.TextIOWrapper`. Como todos os objectos tem identidade, valor e tipo (ver listagem 7.1).

```

1 >>> meu_ficheiro = open('toto.txt', 'r')
2 >>> meu_ficheiro
3 <_io.TextIOWrapper name='toto.txt' mode='r' encoding='UTF-8'>
4 >>> id(meu_ficheiro)
5 4479733416
6 >>> type(meu_ficheiro)
7 <class '_io.TextIOWrapper'>
8 >>>
```

Listagem 7.1: Abertura de um ficheiro

Neste exemplo, abrimos um ficheiro de texto de nome **toto.txt**, no modo de leitura (**r**). O objecto devolvido por `open` foi associado ao nome **meu\_ficheiro** através do qual podemos aceder às várias operações sobre o ficheiro disponibilizadas pela classe (por exemplo, a operação de leitura `read`), como se mostra na figura 7.1.

No exemplo acima apenas tivemos que indicar o nome do ficheiro porque este estava na zona de trabalho corrente. Podemos ter que indicar o caminho absoluto para o ficheiro, caso ele esteja noutro local. Nessa situação, o modo de indicar o caminho depende da plataforma que estivermos a usar. Por exemplo, se for em ambiente **Windows** podemos usar dois modos alternativos:

```

1 meu_ficheiro = open(r"c:\\caminho\\para\\ficheiro\\toto.txt", 'r')
2 meu_ficheiro = open("c:\\\\caminho\\\\para\\\\ficheiro\\\\toto.txt", 'r')
```

Notar que se prefixarmos a cadeia de caracteres que indica o caminho com **r** podemos usar a notação habitual, caso contrário temos que usar duas barras para trás.

Em ambiente **Mac OS X** ou **Linux** será:

```
1 meu_ficheiro = open("/caminho/para/ficheiro/toto.txt", 'r')
```

## Construtor

`open` é o construtor do tipo. Existem vários modos de abrir um ficheiro

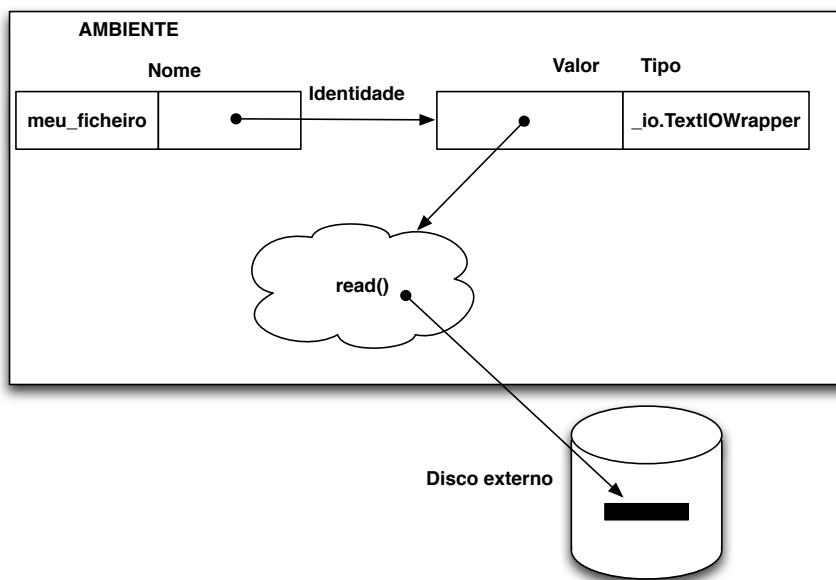


Figura 7.1: Representação em memória

dependendo da operação pretendida e da forma como deve ser interpretado o conteúdo. Para o caso de ficheiros de texto a tabela ?? mostra as alternativas..

| Modo | Interpretação             |
|------|---------------------------|
| r    | Modo de leitura           |
| w    | Modo de escrita           |
| a    | Modo de acrescentar       |
| r+   | Modo de leitura e escrita |

Tabela 7.1: Modos de abertura de ficheiros de texto

Para o caso de ficheiros binários devemos acrescentar a letra **b** ao modo.

É boa prática de programação fechar todo o ficheiro que já não está a ser usado. Para tal usa-se a operação de **close** aplicada ao nome associado ao ficheiro:

<sup>1</sup> `meu_ficheiro.close()`

Deste modo, além de libertarmos espaço, evitamos eventuais corrupções da informação guardada no caso de acontecer alguma situação anómala.

## 7.2 Leitura

Passemos agora ao problema da leitura de dados. Comecemos pelos operações de entrada possíveis como ilustra a tabela ??.

| Operador    | Interpretação                                            |
|-------------|----------------------------------------------------------|
| read()      | lê todo o ficheiro de uma só vez                         |
| read(N)     | lê N bytes                                               |
| readline()  | lê a próxima linha do ficheiro (incluindo \n)            |
| readlines() | lê e guarda como sequência de linhas (cada linha com \n) |

Tabela 7.2: Operações de leitura com ficheiros

Como se pode ver na tabela podemos ler toda a informação de uma só vez, ler um determinado número de caracteres (codificados em bytes), ler por linhas ou ainda ler e guardar como uma sequência de linhas, no caso uma lista. Tratando-se de métodos, a sintaxe a utilizar deverá ser:

*<nome\_ficheiro>. <operação>.*

Vejamos agora o que passa em concreto. Quando abrimos um ficheiro para ler ou consultar dados a situação é a apresentada na figura 7.2.

É como se existisse uma pequena janela que nos mostra o início do ficheiro. A listagem abaixo mostra o uso das operações de leitura<sup>2</sup>.

```

1 >>> meu_ficheiro = open('toto.txt', 'r')
2 >>> todo_ficheiro = meu_ficheiro.read()
3 >>> todo_ficheiro
4 'Um ficheiro pequeno,\ncom caracteres estranhos.\n\npara
 testar a leitura \\n de\\nficheiros.\nne outras coisas \\t
 mais!'
5 >>> print(todo_ficheiro)
6 Um ficheiro pequeno,
7 com caracteres estranhos.
8

```

---

<sup>2</sup>Mais à frente explicaremos porque fechamos e abrimos o ficheiro antes de cada operação.

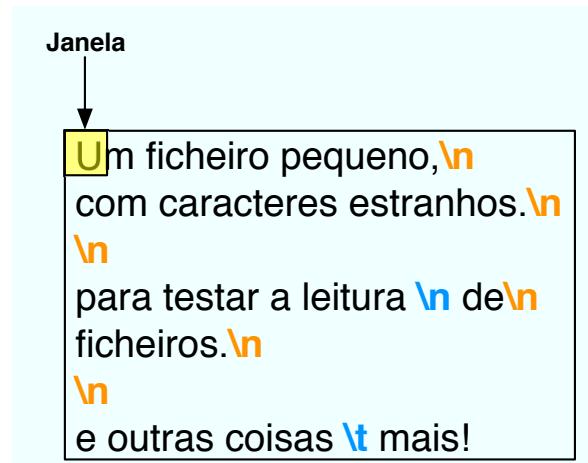


Figura 7.2: Um ficheiro depois de aberto

```
9 para testar a leitura \n de
10 ficheiros.
11 e outras coisas \t mais!
12 >>> meu_ficheiro.close()
13 >>> meu_ficheiro = open('toto.txt','r')
14 >>> uma_linha = meu_ficheiro.readline()
15 >>> uma_linha
16 'Um ficheiro pequeno,\n'
17 >>> print(uma_linha)
18 Um ficheiro pequeno,
19
20 >>> meu_ficheiro.close()
21 >>> meu_ficheiro = open('toto.txt','r')
22 >>> lista_linhas = meu_ficheiro.readlines()
23 >>> lista_linhas
24 ['Um ficheiro pequeno,\n', 'com caracteres estranhos.\n', '\n',
 , 'para testar a leitura \\n de\\n', 'ficheiros.\n', 'e
 outras coisas \\\t mais!']
25 >>> meu_ficheiro.close()
26 >>> meu_ficheiro = open('toto.txt','r')
27 >>> alguns_caracteres = meu_ficheiro.read(10)
28 >>> alguns_caracteres
29 'Um ficheir'
30 >>> alguns_caracteres = meu_ficheiro.read(15)
```

```

31 >>> alguns_caracteres
32 'o pequeno,\ncom '
33 >>>

```

Depois de abrir o ficheiro para leitura (linha 1), lemos todo o seu conteúdo e associamos o resultado (uma cadeia de caracteres) ao nome `meu_ficheiro` (linha 2). Se consultarmos agora pelo nome obtemos todos os caracteres do ficheiro, incluindo as marcas de escape e as mudanças de linha (linhas 3 e 4). Se mandarmos imprimir o conteúdo da variável (linha 5) os sinais são correctamente interpretados e o ficheiro aparece na forma desejada (linhas 6 a 10). O restante da listagem ilustra as outras operações possíveis.

Estas operações são frequentes pelo que normalmente se escrevem pequenos programas que podem depois ser reutilizados. Um exemplo simples para a leitura completa dos dados é dado na listagem 7.3.

```

1 def ler_tudo():
2 nome_f = input("Nome absoluto do ficheiro: ")
3 fich_ent = open(nome_f, 'r')
4 dados = fich_ent.read()
5 fich_ent.close()
6 return dados
7
8 if __name__ == '__main__':
9 print(ler_tudo())

```

Listagem 7.2: Leitura completa de um ficheiro

Analisemos com detalhe a definição `ler_tudo`. Começamos por pedir ao utilizador o nome do ficheiro. De seguida abrimos o ficheiro em modo de leitura e associamos o objecto correspondente ao nome `fich_ent`. Passamos à leitura completa do ficheiro que se traduz pela criação de um objecto do tipo cadeia de caracteres associada à variável `dados`. Porque não queremos fazer mais nada, fechamos o ficheiro e devolvemos o resultado.

Consideremos a sessão 7.3.

```

1 >>> import leitura
2 >>> dir(fleitura)
3 ['__builtins__', '__doc__', '__file__', '__name__', 'ler_tudo']
4 >>> leitura.ler_tudo()
5
6 Nome absoluto do ficheiro: /Users/ernestojfcosta/python/toto.
 txt

```

```

7 Um ficheiro pequeno,
8 com caracteres estranhos.

9
10 para testar a leitura \n de
11 ficheiros.

12
13 e outras coisas \t mais!
14 >>>

```

Listagem 7.3: Leitura de ficheiros

Que comentários podemos fazer? Em primeiro lugar temos uma linha em branco antes do pedido do nome do ficheiro seguida de alguns espaços em branco. Tal deve-se aos caracteres de controlo no interior da cadeia de caracteres. O nome do ficheiro é absoluto, ou seja, indicamos o caminho para o ficheiro. De seguida o ficheiro é impresso aparecendo exactamente na forma como o escrevemos. As marcas de fim de linha no entanto não aparecem!

Vejamos agora o resto das operações de entrada para o que escrevemos o programa da listagem 7.4.

```

1 #--*- coding: mac-roman -*-#
2 # Teste de ficheiros - Leitura
3
4 def leitura():
5 nome_f= input("\n Nome absoluto do ficheiro:\t")
6 fich_ent=open(nome_f, 'r')
7 bytes=fich_ent.read(8)
8 print("Bytes: ", bytes)
9 linha= fich_ent.readline()
10 print("Linha: ", linha)
11 linhas=fich_ent.readlines()
12 print("Linhas: ", linhas)
13 fich_ent.close()
14 return "Fim"
15
16 if __name__ == '__main__':
17 leitura()

```

Listagem 7.4: Operações de entrada

A sessão no interpretador foi a seguinte:

```

1 >>> import leitura

```

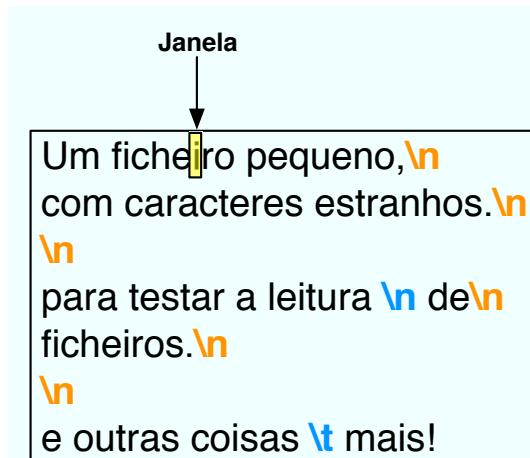


Figura 7.3: Situação depois de lidos os primeiros 8 bytes

```

2 >>> dir(leitura)
3 ['__builtins__', '__doc__', '__file__', '__name__', 'leitura']
4 >>> leitura.leitura()
5
6 Nome absoluto do ficheiro: /Users/ernestojfcosta/python/
 fichent.txt
7 Bytes: Um fiche
8 Linha: iro pequeno,
9
10 Linhas: ['com caracteres estranhos.\n', '\n', 'para testar a
 leitura \\n de\\n', 'ficheiros.\n', '\n', 'e outras coisas
 \\t mais!']
11 'Fim'
12 >>>

```

Olhando para o resultado notamos a importância do conceito de **janela** acima referido. Começamos por mandar ler 8 bytes <sup>3</sup>. Quando pedimos a leitura de uma linha ela é feita **a partir** do local onde a janela ficou depois de serem lidos os 8 bytes (ver figura 7.3). Igualmente quando mandamos ler as linhas do ficheiro são apenas as que restam que são lidas. Mas se quisermos que todas as leituras tenham como referência o início como proceder? Mais à frente trataremos dessa questão.

<sup>3</sup>Cada byte corresponde a um carácter.

## 7.3 Escrita

A escrita num ficheiro é a operação inversa da leitura. Pressupõe que um ficheiro foi aberto para escrita graças ao comando `open(nome, 'w')`. Como se pode ver na tabela ?? existem duas operações fundamentais. Podemos escrever algo simples com `write()` ou algo mais complicado graças a `writelines()`. A sintaxe geral do comando de escrita é:

```
<ficheiro>.write(<cadeia_de_caracteres>)
```

Notar que o que se escreve tem que ser um objecto do tipo cadeia de caracteres. Se o não for terá que ser convertido! A listagem ?? ilustra o processo, com um exemplo simples admitindo que o objecto ficheiro pode ser acedido pelo nome `fout`.

```
1 >>> x = 15
2 >>> fout.write(str(x))
```

| Operador                     | Interpretação                                |
|------------------------------|----------------------------------------------|
| <code>write(str)</code>      | escreve a cadeira de caracteres num ficheiro |
| <code>writelines(seq)</code> | escreve <i>seq</i> como sequência de linhas  |
| <code>close()</code>         | operação de fecho do ficheiro                |

Tabela 7.3: Operações de escrita com ficheiros

Vejamos um exemplo simples como o indicado na listagem 7.5.

```
1 >>> f=open('/Users/ernestojfcosta/python/escreve.txt', 'w')
2 >>> f.write('Teste de escrita\na\tb\c\nParece o abecedário!\nFim.')
3 >>> f.close()
4 >>> g=open('/Users/ernestojfcosta/python/escreve.txt', 'r')
5 >>> print(g.read())
6 Teste de escrita
7 a b\c
8 Parece o abecedário!
9 Fim.
10 >>>
```

Listagem 7.5: Escrita num ficheiro

Como se pode verificar os caracteres de controlo que incluímos na cadeia de caracteres são preservados. Devemos ter em atenção que quando se abre um ficheiro para escrita (modo 'w') caso o ficheiro já exista ele é destruído. Para que tal não acontece temos que abrir o ficheiro em modo de acrescentar (modo 'a'). Um exemplo.

```

1 >>> f=open('/Users/ernestojfcosta/python/escreve.txt','a')
2 >>> f.write('Vamos colocar mais \n qualquer coisa \nno
 ficheiro')
3 >>> f.close()
4 >>> f=open('/Users/ernestojfcosta/python/escreve.txt','r')
5 >>> print(f.read())
6 Teste de escrita
7 a b\c
8 Parece o abecedário!
9 Fim.Vamos colocar mais
10 qualquer coisa
11 no ficheiro
12 >>> f.close()
13 >>> f=open('/Users/ernestojfcosta/python/escreve.txt','r')
14 >>> f.read()
15 'Teste de escrita\na\tb\\c\nParece o abecedário!\nFim.Vamos
 colocar mais \n qualquer coisa \nno ficheiro'
16 >>>

```

Este exemplo mostra duas coisas. Por um lado, o que se acrescenta é colocado no final do ficheiro. Em segundo lugar, o que vemos ao ler o ficheiro é diferente caso utilizemos o comando **print** ou não. Porque será esta diferença?.

## 7.4 Navegar

Tratemos agora do problema de querer ler ou escrever em determinadas partes do ficheiro. Isso obriga a ter comandos que controlem o posicionamento da janela sobre o ficheiro. Para navegar num ficheiro existem dois comandos. O comando **tell**, que nos indica a posição da janela relativamente ao início do ficheiro. O comando **seek**, que nos permite reposicionar a janela. O funcionamento deste último comando depende do tipo de ficheiro (binário ou de texto).

Vamos começar com um ficheiro de texto, mais concretamente o ficheiro

| Nome | Operador              | Interpretação                           |
|------|-----------------------|-----------------------------------------|
| seek | fich.seek(pos,como=0) | movimenta para uma nova posição         |
| tell | fich.tell()           | qual a posição relativamente ao início? |

Tabela 7.4: Operadores de navegação

*ficheiro2013.txt*<sup>4</sup> de conteúdo:

```

1 um ficheiro pequeno,
2 com caracteres estranhos.
3
4 para testar a leitura \n de
5 ficheiros.
6 e outras coisas \t mais!

```

A listagem 7.6, ilustra como podemos navegar pelo ficheiro.

```

1 >>> f_ent = open('/data/ficheiro2013.txt', 'r')
2 >>> bytes = f_ent.read(8)
3 >>> bytes
4 'um fiche'
5 >>> f_ent.seek(6)
6 6
7 >>> mais_bytes = f_ent.read(10)
8 >>> mais_bytes
9 'heiro pequ'
10 >>> f_ent.tell()
11 16
12 >>> f_ent.seek(0,2)
13 111
14 >>> novos_bytes = f_ent.read(5)
15 >>> novos_bytes
16 ''
17 >>> f_ent.tell()
18 111
19 >>> f_ent.seek(0)
20 0
21 >>> ler_tudo = f_ent.read()
22 >>> ler_tudo

```

---

<sup>4</sup>Use um editor de texto simples para criar o ficheiro. Em alternativa para is ao sítio do livro, em <http://ipr.net> e baixar o ficheiro.

```

23 'um ficheiro pequeno,\ncom caracteres estranhos.\n\npara
 testar a leitura \\\n de\ficheiros.\ne outras coisas \\\t
 mais!'

```

Listagem 7.6: 'Navegar num ficheiro'

Começámos por abrir o ficheiro de texto em modo de leitura e mandamos ler os primeiros 8 bytes (linhas 1 a 4). De seguida reposicionamos a janela na posição 6 e lemos os dez bytes seguintes (linhas 5 a 9). Indagamos depois em que posição estamos (linhas 10 e 11). Colocamo-nos depois no final do ficheiro, ficando a saber quantos caracteres tem (linhas 12 e 13). Se tentarmos ler para além do final o que recebemos é uma cadeia de caracteres vazia (linhas 14 a 16). Verificamos que continuamos no fim do ficheiro (linhas 17 e 18). Voltamos ao início do ficheiro e lemos tudo (linhas 19 a 23).

Podemos fazer movimentações relativas ao ponto em que nos encontramos, se conjugarmos os dois comandos (**seek** e **tell**), como se ilustra na listagem.

```

1 >>> f_ent.seek(0)
2 0
3 >>> car = f_ent.read(15)
4 >>> car
5 'um ficheiro peq'
6 >>> f_ent.tell()
7 15
8 >>> f_ent.seek(f_ent.tell() + 5)
9 20
10 >>> mais_car = f_ent.read(10)
11 >>> mais_car
12 '\ncom carac'
13 >>> f_ent.tell()
14 30

```

No caso dos ficheiros binários estes movimentos relativos são mais simples de fazer. Na realidade, nestes casos o segundo argumento de **seek** pode assumir os valores 0, quando a referência é o início do ficheiro, 1, quando a referência é a posição corrente da janela, ou ainda 2, quando nos movimentamos relativamente ao fim do ficheiro. O valor do primeiro argumento pode ser um inteiro positivo ou negativo para nos movimentarmos para a direita ou para a esquerda, respectivamente do ponto de referência. É óbvio que se o ponto de referência for o início do ficheiro não nos podemos movimentar para a esquerda e, de modo semelhante, se a referência for o fim do ficheiro não podemos deslocarmos para a direita.

```
1 >>> f = open('/data/temp_fich', 'wb+')
2 >>> f.write(b'0123456789abcdef')
3 16
4 >>> f.seek(5,0)
5
6 >>> f.read(2)
7 b'56'
8 >>> f.seek(-2,1)
9 5
10 >>> f.read(4)
11 b'5678'
12 >>> f.tell()
13 9
14 >>> f.seek(-5,2)
15 11
16 >>> f.read(2)
17 b'bc'
18 f.close()
```

## 7.5 Intermezzo

### A instrução `with`

Por defeito, quando estamos a escrever para num ficheiro, os dados vão primeiro para uma memória tampão e só posteriormente são escritos em disco. A operação de fecho de um ficheiro começa por esvaziar a memória tampão e só depois fecha efectivamente o ficheiro. Podemos forçar o esvaziamento recorrendo ao método `flush`. O facto de se usar uma memória tampão pode colocar problemas caso haja um fim inesperado da execução do programa devido a um erro. Podemos evitar o uso deste tipo de memória com um parâmetro adicional no método `open` mas tal tem custos ao nível do desempenho. Deve pois ser pesado o uso, ou não, da memória tampão e como o método `flush` nos pode ajudar nesta questão.

Podemos no entanto garantir que o ficheiro é encerrado de forma conveniente mesmo quando ocorre uma interrupção anormal do programa. A solução baseia-se no conceito de **gestores de contexto** e no uso da instrução `with`. Gestores de contexto são objectos que têm associados duas operações, uma de entrada e outra de saída, operações essas que são executadas quando se entra ou se sai de um contexto, respectivamente. A instrução `with` define

Gestores de contexto

um contexto, tendo como sintaxe:

```
1 with expressão as var:
2 bloco
```

A parte **as** *var* é opcional. *expressão* tem que ser, ou gerar, um gestor de contexto, *var* é o nome que vai ficar associado ao objecto. É como se se tivesse feito uma atribuição do nome à expressão. Acontece que o objecto devolvido pela instrução **open** é um gestor de contexto. Podemos então fazer algo como:

```
1 with open(ficheiro,'r') as meu_fich:
2 bloco
```

Demos modo temos a garantia de que, mesmo que ocorra um erro durante a execução do bloco, o ficheiro será fechado.

### Formatação por linhas

Embora um ficheiro seja uma longa cadeia de caracteres ele está organizado por linhas. Quando manipulamos um ficheiro temos que ter isso em atenção. A listagem abaixo ilustra uma consequência dessa organização.

```
1 >>> f_ent = open('/data/ficheiro2013.txt','r')
2 >>> for linha in f_ent:
3 ... print(linha)
4 ...
5 um ficheiro pequeno,
6 com caracteres estranhos.
7
8
9
10
11 para testar a leitura \n de
12
13 ficheiros.
14
15 e outras coisas \t mais!
16 >>> f_ent.seek(0)
17 0
18 >>> for linha in f_ent:
19 ... print(linha[:-1])
20 ...
```

```
21 um ficheiro pequeno,
22 com caracteres estranhos.
23
24 para testar a leitura \n de
25 ficheiros.
26 e outras coisas \t mais
27 >>>
```

Nestes dois exemplos procuramos imprimir o conteúdo do ficheiro, mantendo a estrutura por linhas. Como as linhas terminam por um indicador de fim de linha que a função `print` interpreta, vemos que no primeiro caso cada linha é seguida de uma linha vazia. Na segunda abordagem tal não acontece pois retiramos explicitamente o último caracter.

## 7.6 Exemplo

Regressemos agora ao problema dos dados climatéricos de algumas cidades de Portugal. Comecemos por uma versão muito simples do problema: ler um ficheiro onde estão guardadas as temperaturas médias mensais ao longo de um dado ano. Apenas pretendemos ler e mostrar através de um gráfico, os resultados. O código é o da listagem 7.7.

```
1 import matplotlib.pyplot as plt
2
3 def ler(ficheiro):
4 with open(ficheiro,'r') as f_ent:
5 dados_car = f_ent.read().split()
6 dados = []
7 for elem in dados_car:
8 dados.append(float(elem))
9 return dados
10
11 def mostra(xetiq, yetiq,tit,x,y):
12 plt.xlabel(xetiq)
13 plt.ylabel(yetiq)
14 plt.plot(x,y)
15
16
17 def main(ficheiro):
18 dados = ler(ficheiro)
19 mostra('Meses','Temperatura','','',range(1,13),dados)
```

```

20 plt.show()
21
22
23 if __name__ == '__main__':
24 main('/data/dados_simples.txt')

```

Listagem 7.7: Ler e mostrar temperaturas

O programa principal executa em sequência as duas tarefas: primeiro lê os dados e depois mostra os resultados por recurso ao módulo **matplotlib**. A leitura do ficheiro não oferece dificuldades. O ficheiro é aberto para leitura e o contexto é definido com o recurso à instrução **with**. Os dados são lidos de uma vez só e separados pelo método **split** (linha 5). Entre as linhas 6 e 9 transformamos as cadeias de caracteres que representam os números em número em vírgula flutuante.

O código desta rotina pode ainda ser simplificado neste último aspecto pois estamos perante um padrão conhecido: um ciclo que acrescenta no final de uma lista, inicialmente vazia, o resultado de uma certa operação. Podemos por isso recorrer a listas por compreensão. Acresce que, se formos eliminando as variáveis que são usadas como memória temporária, o programa fica mais pequeno ainda. A listagem 7.8 mostra as diferentes alternativas.

```

1 def ler_1(ficheiro):
2 with open(ficheiro, 'r') as f_ent:
3 dados_car = f_ent.read().split()
4 dados = [float(elem) for elem in dados_car]
5 return dados
6
7 def ler_2(ficheiro):
8 with open(ficheiro, 'r') as f_ent:
9 dados = [float(elem) for elem in f_ent.read().split()]
10 return dados
11
12 def ler_3(ficheiro):
13 with open(ficheiro, 'r') as f_ent:
14 return [float(elem) for elem in f_ent.read().split()]

```

Listagem 7.8: Ler ficheiro: alternativas

Está na hora de tornar o problema mais interessante, considerando que no nosso ficheiro estão as temperaturas de várias cidades de Portugal. A listagem 7.9 mostra o código inicial com duas funções. A primeira, lê o ficheiro linha a linha, guardando os sucessivos resultados, até o ficheiro ter

sido todo lido. Usa a segunda função, que lê a próxima linha ainda não lida, devolvendo -1 caso não haja mais nada para ler. Funciona em três passos. Começa por procurar a primeira linha significativa (linhas 19 a 21), isto é, com números. Depois testa se efectivamente encontrou uma linha com dados. Se não encontrou devolve -1, se encontrou fabrica a lista dos números e devolve o resultado.

```

1 def le_todas_temperaturas(fich):
2 """
3 Extrai os dados de temperaturas relativos a Portugal.
4 """
5 with open(fich,'r') as f_ent:
6 portugal = list()
7 dados = le_uma_temperatura(f_ent)
8 while dados != -1:
9 portugal.append(dados)
10 dados = le_uma_temperatura(f_ent)
11 f_ent.close()
12 return portugal
13
14 def le_uma_temperatura(f_ent):
15 """
16 Ler dados da temperatura de uma cidade.
17 Devolve -1 se fim de ficheiro
18 """
19 linha = f_ent.readline()
20 while (linha != '') and (linha == '\n'):
21 linha = f_ent.readline()
22 if linha == '':
23 return -1
24 else:
25 linha = linha[:-1].split()
26 return [float(dado) for dado in linha]
27
28 if __name__ == '__main__':
29 dados = ler_todas_temperaturas('/data/temperaturas.txt')
30 print(dados)
```

Listagem 7.9: Todas as temperaturas

Podemos visualizar os dados com pequenas alterações ao código, como mostra a listagem 7.10.

```
1 import matplotlib.pyplot as plt
2
3 def le_todas_temperaturas(fich):
4 """
5 Extrai os dados de temperaturas relativos a Portugal.
6 """
7 with open(fich, 'r') as f_ent:
8 portugal = list()
9 dados = le_uma_temperatura(f_ent)
10 while dados != -1:
11 portugal.append(dados)
12 dados = le_uma_temperatura(f_ent)
13 f_ent.close()
14 return portugal
15
16 def le_uma_temperatura(f_ent):
17 """
18 Ler dados da temperatura de uma cidade.
19 Devolve -1 se fim de ficheiro
20 """
21 linha = f_ent.readline()
22 while (linha != '') and (linha == '\n'):
23 linha = f_ent.readline()
24 if linha == '':
25 return -1
26 else:
27 linha = linha[:-1].split()
28 return [float(dado) for dado in linha]
29
30 def mostra_todas(xetiq,yetiq,tit,dados):
31 plt.xlabel(xetiq)
32 plt.ylabel(yetiq)
33 plt.title(tit)
34 for cidade in dados:
35 plt.plot(cidade)
36
37
38 def main(ficheiro):
39 dados = le_todas_temperaturas(ficheiro)
40 mostra_todas('Meses','Temperatura','Temperaturas Médias
das Cidades',dados)
```

```

41 plt.show()
42
43 if __name__ == '__main__':
44 main('/data/temperaturas.txt')

```

Listagem 7.10: Temperaturas

Ao correr o programa obtemos o gráfico da figura ??.

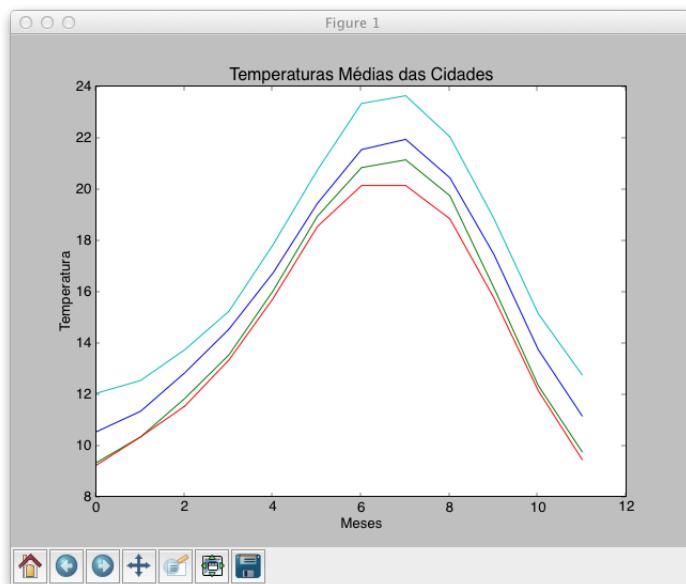


Figura 7.4: Temperaturas das cidades de Portugal

A solução apresentada cumpre o seu papel mas não deixa de ter um defeito muito grande: não sabemos a que cidade corresponde cada curva. Mas existe uma solução simples. Alteramos o ficheiro de modo a que, no início de cada linha, esteja o nome da cidade. Depois é só alterar a função que lê uma linha e a função que mostra os resultados, como indicamos na listagem 7.11.

```

1 def le_uma_temperatura(f_ent):
2 """
3 Ler dados da temperatura de uma cidade.
4 Devolve -1 se fim de ficheiro
5 """
6 linha = f_ent.readline()
7 while (linha != '') and (linha == '\n'):
8 linha = f_ent.readline()

```

```

9 if linha == '':
10 return -1
11 else:
12 linha = linha[:-1].split()
13 cidade = linha[0]
14 dados = [float(dado) for dado in linha[1:]]
15 return cidade, dados
16
17 def mostra_todas(xetiq,yetiq,tit,dados):
18 plt.xlabel(xetiq)
19 plt.ylabel(yetiq)
20 plt.title(tit)
21 for dado in dados:
22 cidade = dado[0]
23 plt.plot(dado[1], label=cidade)
24 plt.legend()

```

Listagem 7.11: Gráfico com legendas

Agora quando executamos o programa o gráfico já nos mostra toda a informação relevante.

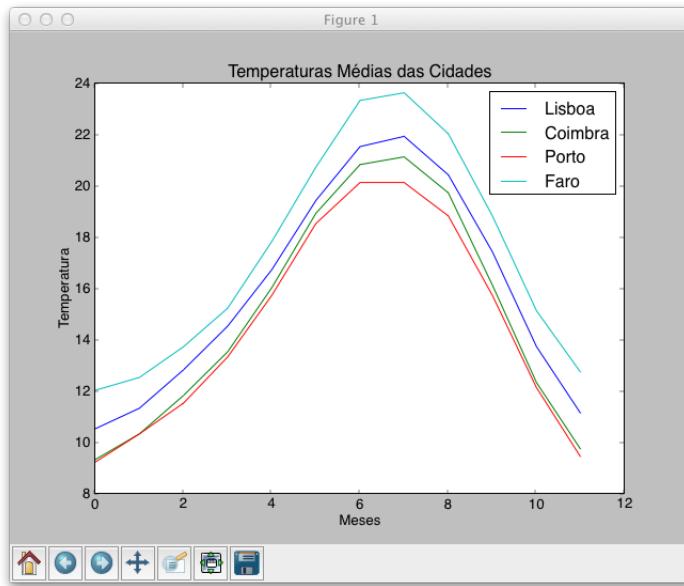


Figura 7.5: Gráfico com legenda

Para concluir este exemplo, vamos considerar a situação em que o ficheiro é mais complexo pois contém informação sobre a temperatura e a pluviosidade. Em concreto, admitamos que a informação por cada cidade está na forma.

```

1 Lisboa
2 Pluviosidade 95.2 86.7 84.7 59.5 44.4 17.9 4.3 5.2
 33.0 74.7 99.6 96.7
3 Temperatura 10.5 11.3 12.8 14.5 16.7 19.4 21.5 21.9
 20.4 17.4 13.7 11.1

```

Assim, na primeira linha temos o nome da cidade, na segunda linha o identificador pluviosidade seguido dos dozes valores mensais e, na terceira linha a informação correspondente à temperatura. A nossa solução desdobra-se em duas partes: primeiro obtemos os dados e depois visualizamos o resultado.

```

1 def main(fich):
2 dicio = dados_portugal(fich)
3 mostra(dicio)

```

A extracção dos dados é feita retirando a informação relevante por cada cidade e colocando-a num dicionário. Este dicionário tem por chaves os nomes das cidades. A cada uma delas está associado um outro dicionário com duas chaves: pluviosidade e temperatura. Os respectivos valores são guardados numa lista.

```

1 def dados_portugal(fich):
2 """
3 Extrai os dados relativos a Portugal.
4 """
5 f_ent = open(fich, 'r')
6 portugal = dict()
7
8 ficha = le_cidade(f_ent)
9 while ficha != -1:
10 cidade,pluviosidade,temperatura = ficha
11 portugal.update({cidade:{'pluviosidade':pluviosidade,
12 'temperatura':temperatura}})
13 ficha = le_cidade(f_ent)
14 f_ent.close()
15 return portugal

```

Para obter os dados de cada cidade

```

1 def le_cidade(f_ent):

```

```

2 """
3 Lê os dados de uma cidade.Devolve -1 se alcançou o fim de
4 ficheiro
5 """
6
7 # procura primeira linha significativa
8 linha = f_ent.readline()
9
10 while (linha != '') and (linha == '\n'):
11 linha = f_ent.readline()
12
13 if linha == '':
14 return -1
15 else:
16 # extrai dados
17 cidade = linha[:-1]
18 pluviosidade = [float(dado) for dado in f_ent.readline()
19 ()[:-1].split('\t')[1:]]
20 temperatura = [float(dado) for dado in f_ent.readline()
21 ()[:-1].split('\t')[1:]]
22
23 return (cidade,pluviosidade,temperatura)

```

Para obter os dados de uma cidade começamos por procurar a primeira linha com texto, obrigatoriamente o nome da cidade. Depois lemos as duas linhas seguintes, retiramos a palavra que identifica o tipo da informação e convertemos o resto para uma lista de números em vírgula flutuante. Falat agora resolver a quesão da visualização. Vamos usar o módulo `matplotlib`.

```

1 def mostra(dados):
2 """
3 dados é um dicionário. A chave é o nome da cidade, o valor
4 é outro dicionário
5 de chaves 'pluviosidade' e 'temperatura'
6 """
7
8 meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Maio',
9 'Junho', 'Julho', 'Agosto', 'Setembro', 'Outubro', ' '
10 'Novembro', 'Dezembro']
11
12 chuva = []
13 cidades = []
14 temp = []
15
16 for c,v in dados.items():
17 chuva.append(v['pluviosidade'])
18 temp.append(v['temperatura'])

```

```
13 cidades.append(c)
14
15 figura = plt.figure()
16 fig_1 = figura.add_subplot(211)
17 plt.title('Cidades de Portugal')
18 fig_2 = figura.add_subplot(212)
19
20
21 for indice in range(len(cidades)):
22 fig_1.plot(chuva[indice])
23 fig_2.plot(temp[indice])
24
25 fig_1.set_ylabel('Pluviosidade (mm)')
26 fig_2.set_ylabel('Temperatura (C)')
27 plt.xticks(range(0,12),meses, rotation=17)
28 plt.legend(cidades, loc=0)
29
30 plt.show()
```

Quando executamos o programa obtemos o resultado da figura 7.6.

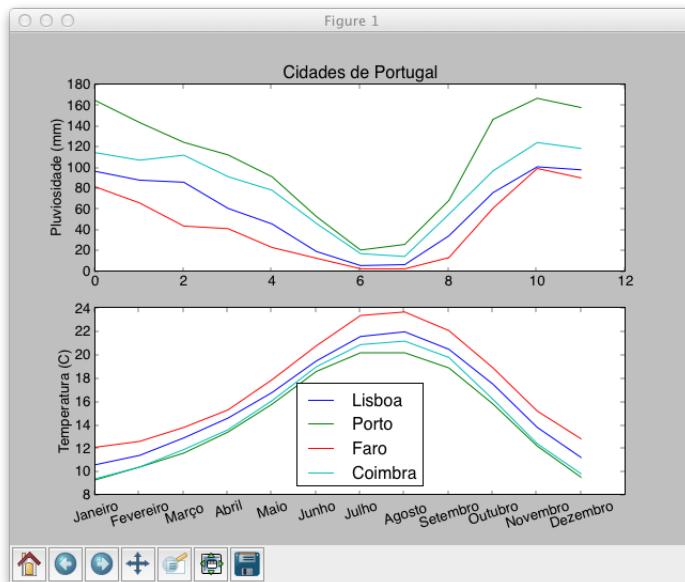


Figura 7.6: Temperatura e pluviosidade

## 7.7 De um ficheiro de palavras a um dicionário de frequências

Suponhamos que temos um dicionário com palavras e queremos construir um dicionário cujas chaves são inteiros que traduzem o tamanho das palavras e cujos valores são listas de palavras com esse tamanho. Apresentemos primeiro uma solução simples:

```

1 def fich_dic_a(ficheiro):
2 """ Lê o conteúdo do ficheiro e constrói um dicionário
3 com chave um inteiro e valor uma lista com as palavras de
4 comprimento igual ao valor da chave.
5 """
6 f_in = open(ficheiro, 'r')
7 lista_pal = f_in.read().split()
8 dic = {}
9 for palavra in lista_pal:
10 comp = len(palavra)
11 dic[comp] = dic.get(comp, []) + [palavra]
12 return dic

```

Agora uma solução que prevê o uso de símbolos especiais.

```

1 def fich_dic_b(ficheiro):
2 """ Lê o conteúdo do ficheiro e constrói um dicionário
3 com chave um inteiro e valor uma lista com as palavras de
4 comprimento igual ao valor da chave.
5 """
6 f_in = open(ficheiro, 'r')
7 lista_pal = f_in.read().split()
8 dic = {}
9 especiais = ['\n', '.', ',', '!', '?']
10 for palavra in lista_pal:
11 if palavra[-1] in especiais:
12 palavra = palavra[:-1]
13 comp = len(palavra)
14 if comp:
15 dic[comp] = dic.get(comp, []) + [palavra]
16 return dic

```

Notar o uso de uma lista onde guardamos caracteres especiais que não devem fazer parte da palavra. Notar ainda o teste ao comprimento: se um símbolo estiver isolado, ao ser retirado passa a ser uma cadeia sem caracteres

e não deve ser incluída.

Finalmente, uma solução em que as **palavras repetidas** só são incluídas um vez.

```
1 def fich_dic_c(ficheiro):
2 """ Lê o conteúdo do ficheiro e constrói um dicionário
3 com chave um inteiro e valor uma lista com as palavras de
4 comprimento igual ao valor da chave.
5 """
6 especiais = ['\n', '.', ',', '!', '?']
7 f_in = open(ficheiro, 'r')
8 lista_pal = f_in.read().split()
9 # filtra
10 lista_final = []
11 for palavra in lista_pal:
12 # símbolos especiais fora
13 if palavra in especiais:
14 continue
15 elif palavra[-1] in especiais:
16 palavra = palavra[:-1]
17 # repetições fora
18 if palavra not in lista_final:
19 lista_final.append(palavra)
20 # Constrói dicionário
21 dic = {}
22 for palavra in lista_final:
23 comp = len(palavra)
24 dic[comp] = dic.get(comp, []) + [palavra]
25 return dic
```

## 7.8 Outros Tipos de Ficheiros

Em **Python** é possível manipular de modo simples informação guardada em ficheiros organizados de modo específico. Vamos analisar dois módulos que permitem aumentar as capacidades da linguagem no que se refere a certos tipos de ficheiros.

### 7.8.1 csv

Como sabe existem muitos módulos adicionais em **Python**. Um deles é o módulo **csv**, que permite a manipulação de ficheiros organizados por linhas,

e em que cada linha tem os seus elementos separados por um certo delimitador (*Comma Separated Values*). No caso dos ficheiros **csv** o delimitador é a vírgula, como se depreende pelo seu nome. Mas existe a possibilidade de trabalhar com outros delimitadores. Estes ficheiros têm a vantagem de serem simples, mas têm o inconveniente de existirem algumas variantes. É por isso que o módulo **csv** é interessante pois permite lidar com essas variantes. Nesta secção apenas apresentaremos exemplos simples, ficando para o leitor complementar os seus conhecimentos sobre o módulo recorrendo, por exemplo, ao manual da linguagem. O módulo **csv** permite essencialmente efectuar as operações de leitura e de escrita. Suponhamos que já existe um ficheiro com informação acerca das notas dos nossos alunos. A listagem ilustra um possível conteúdo

```

1 Nome,Testes,Projecto,Normal,Recurso,Nota
2 Ernesto Costa,75,60,45,52,?
3 Zeus Euclides,12,34,45,30,?
4 Anaximandro Heraclito, 36,47,67,12,?
```

Ler este ficheiro e devolver o resultado pode ser feito através de um pequeno programa.

```

1 import csv
2
3 def le_csv(nome_fich):
4 """ Lê um ficheiro em formato csv."""
5 with open(nome_fich) as fich:
6 csv_reader = csv.reader(fich)
7 dados = []
8 for linha in csv_reader:
9 dados.append(linha)
10 fich.close()
11 return dados
```

Como se vê pela listagem, abre-se um ficheiro da forma habitual e depois deixa-se ao método **reader** do módulo **csv** a tarefa de ler todo o ficheiro. Chamando a função sobre o ficheiro das notas o resultado é dado sob a forma de uma lista em que cada elemento é uma lista com os dados de cada linha.

```

1 [['Nome', 'Testes', 'Projecto', 'Normal', 'Recurso', 'Nota'],
2 ['Ernesto Costa', '75', '60', '45', '52', '?'],
3 ['Zeus Euclides', '12', '34', '45', '30', '?'],
4 ['Anaximandro Heraclito', '36', '47', '67', '12', '?']]
```

Realizada esta operação podemos querer introduzir os dados de um novo aluno, isto é, queremos escrever no ficheiro uma nova linha. O método **writer**

vem em nosso auxílio facilitando-nos a tarefa.

```

1 import csv
2
3 def insere_linha_csv(fich, linha):
4 """ Insere uma linha no fim do ficheiro."""
5 with open(fich, 'a') as nome_fich:
6 csv_writer = csv.writer(nome_fich)
7 csv_writer.writerow(linha)
8 nome_fich.close()

```

Mais uma vez usamos o processo de abrir para acrescentar convencional, e depois escrevemos a nova linha. Um exemplo de utilização do programa:

```

1 insere_linha_csv('/data/notas.csv', ['Calvin Hobbes', '90', '85',
 '93', '89', '?'])

```

Vamos agora criar um ficheiro, na realidade uma pequena base de dados, de raiz com informação sobre restaurantes (com os campos: Nome, Morada, Tipo e Custo) . Começamos por apresentar o programa genérico.

```

1 def escreve_csv(fich, dados,delimitador):
2 """ Escreve um ficheiro em formato csv."""
3 with open(fich,'w') as csv_fich:
4 csv_writer = csv.writer(csv_fich,delimiter=delimitador
5)
6 csv_writer.writerows(dados)
6 csv_fich.close()

```

Como se pode ver no exemplo, é possível escolher um delimitador diferente da vírgula. Claro que depois, para ler, essa informação tem que ser dada. Executando a função como no exemplo:

```

1 escreve_csv('/data/rest.csv', [['Nome', 'Morada', 'Tipo', 'Custo'],
 ['Manel dos Ossos', 'Coimbra', 'Portuguesa', '$'], ['Chez Lui', 'Paris', 'Francesa', '$$$$'], ['McMe': 'Burgos', 'Fast Food', '$$']], ':')

```

A base de dados seguinte é criada.

```

1 Nome:Morada:Tipo:Custo
2 Manel dos Ossos:Coimbra:Portuguesa:$
3 Chez Lui:Paris:Francesa:$$$$
4 McMe:Burgos:Fast Food:$$

```

Vamos considerar agora um problema mais realista e que consiste em definir a nota final do aluno conhecidas as notas parciais guardadas no ficheiro de notas.

```

1 def nota_final(fich):
2 """ Calcula nota final. """
3 # Lê
4 dados = le_csv(fich)
5 notas_parciais = dados[1:]
6 # Manipula
7 for i in range(len(notas_parciais)):
8 nome, teste, projecto, normal, recurso, nota =
9 notas_parciais[i]
10 notas_parciais[i][5] = str(0.2 * int(teste) + 0.2 *
11 int(projecto) + 0.6 * max(int(normal),int(recurso)))
12 dados = [dados[0]] + notas_parciais
13
14 # Escreve
15 escreve_csv(fich,dados,',')

```

A solução indicada pode dividir-se em três partes: leitura dos dados (linhas 3 a 5), cálculo da nota (linhas 6 a 11) e reescrita total do ficheiro com os novos valores (linha 13). Depois de correr o programa temos o ficheiro das notas actualizado.

```

1 Nome,Testes,Projecto,Normal,Recurso,Nota
2 Ernesto Costa,75,60,45,52,58.2
3 Zeus Euclides,12,34,45,30,36.2
4 Anaximandro Heraclito, 36,47,67,12,56.8
5 Calvin Hobbes,90,85,93,89,90.8

```

### 7.8.2 urllib

Vivemos no tempo da **Internet**, todos sabemos. Daí que, com naturalidade, a linguagem **Python** tenha construções que nos permitam interagir com informação guardada na grande rede global que é a Web. Um dos módulos que nos auxilia na tarefa é o módulo **urllib.request**. Vejamos, por exemplo, como posso obter a minha página web.

```

1 import urllib.request
2
3 meu_sitio = urllib.request.urlopen("http://ernesto.dei.uc.pt")
4 meus_bytes = meu_sitio.read()
5 minha_cadeia = meus_bytes.decode("utf8")
6 meu_sitio.close()

```

Este exemplo mostra algumas coisas importantes. Em primeiro lugar, podemos abrir uma página web de modo semelhante ao que fazemos para qualquer outro ficheiro (linha 3). Temos depois a possibilidade de ler toda a informação contida na página (linha 4). Só que agora o objecto devolvido pelo método `read` é de um tipo diferente: é uma cadeia de `bytes`. O que fazemos de seguida é converter essa cadeia para uma cadeia de caracteres *normal*, do tipo `str`, o que fazemos recorrendo ao método `decode` (linha 5). O conteúdo que é lido pode ser salvo localmente, e guardado com extensão `html`.

```

1 novo_fich = open('/Users/ernestojfcosta/tmp/urlteste.html', 'w')
2 novo_fich.write(minha_cadeia)
3 novo_fich.close()

```

### Exemplo

Vamos exemplificar um uso possível deste módulo. A ideia é ir ler na web as cotações de duas empresas, no caso a **Apple** e a **Coca-Cola**, num determinado período de tempo, e verificar se existe alguma correlação entre as respectivas variações. Para analisar a existência de correlação (ou não) vamos recorrer ao coeficiente de correlação de Pearson. O **Coeficiente de Correlação de Pearson**(CCP) é definido por:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{(n - 1)S_x S_y}$$

em que  $\bar{x}$  e  $\bar{y}$  são as médias das duas variáveis  $x$  e  $y$ ,  $S_x$  e  $S_y$  são os respectivos desvios padrão.

O CCP varia entre -1 (negativamente correlacionados) e 1 (positivamente correlacionado. A figura 7.7 mostra alguns exemplos típicos (sem correlação, correlação forte - positiva e negativa, e correlação média).

Escrever um programa que calcula o CCP não apresenta dificuldades de maior.

```

1 import math
2
3 def media(lista):
4 """ Calcula a média."""
5 med = sum(lista) / len(lista)
6 return med
7
8 def desvio_padrao(lista):

```

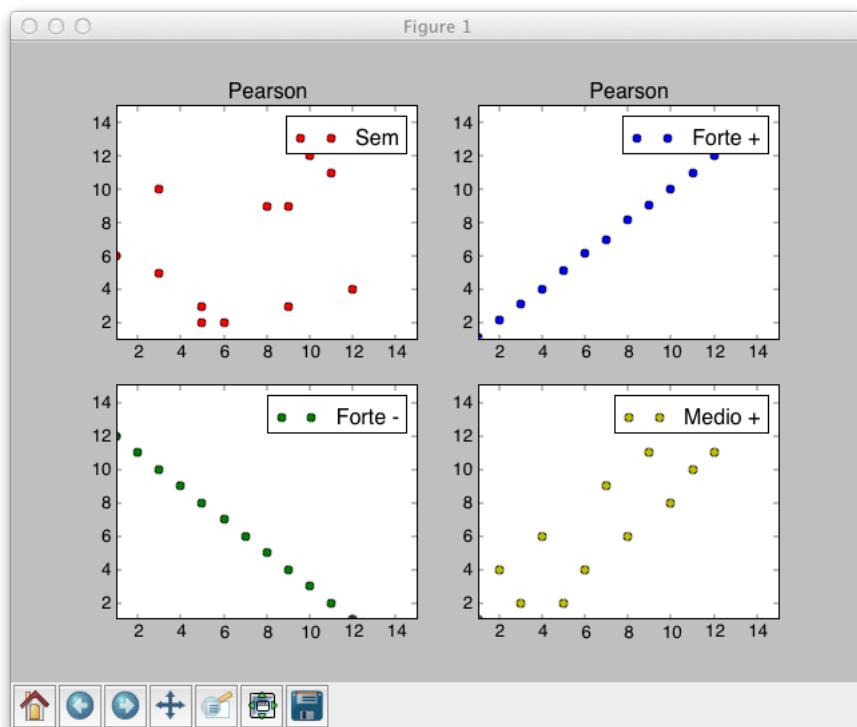


Figura 7.7: Coeficiente de Correlção de Pearson: exemplos

```

9 """ Calcula o desvio padrão."""
10 a_media = media(lista)
11 soma = 0
12 for elem in lista:
13 soma = soma + (elem - a_media) ** 2
14 desvio = math.sqrt(soma/(len(lista) - 1))
15 return desvio
16
17 def pearson(lista_a, lista_b):
18 """ Calcula o coeficiente de correlação entre duas listas
19 de valores."""
20 media_a = media(lista_a)
21 media_b = media(lista_b)
22 desvio_a = desvio_padrao(lista_a)
23 desvio_b = desvio_padrao(lista_b)
24 n = len(lista_a)
25 soma = 0
26
27 for i in range(n):
28 soma = soma + (lista_a[i] - media_a) * (lista_b[i] - media_b)
29
30 numerador = soma
31 denominador = desvio_a * desvio_b
32
33 resultado = numerador / denominador
34
35 return resultado

```

```

25 for indice in range(n):
26 soma = soma + (lista_a[indice] - media_a) * (lista_b[
27 indice] - media_b)
28 correlacao = soma / ((n - 1) * desvio_a * desvio_b)
29 return correlacao

```

Uma vez resolvida esta questão acessória, podemos concentrarmo-nos no problema da comparação concreta dos dados das cotações de fecho de duas empresas. Esses dados vão ser retirados directamente da Web. O formato do ficheiro é o seguinte:

```

1 Date,Open,High,Low,Close,Volume,Adj Close
2 2009-10-19,50.26,50.70,50.05,50.39,5141800,50.39
3 2009-10-16,50.27,50.33,49.65,50.08,6769200,50.08
4 2009-10-15,51.11,51.15,50.00,50.42,8629800,50.42

```

Existe uma primeira linha que descreve os atributos (data da cotação, valor na abertura, mais alto, mais baixo, fecho, volume transacções, próximo do fecho). As linhas seguintes dão os valores concretos. A cotação de fecho está na posição 4. Finalmente o código.

```

1 import urllib
2 import math
3 import matplotlib.pyplot as plt
4
5 # código para CCP omitido
6
7 def compara(url_1,url_2, valor_1,valor_2,elem):
8 """
9 Determina o coeficiente de correlação no período valor_1 -
10 - valor_2
11 relativamente ao elemento elem.
12
13 with urllib.request.urlopen(url_1) as handler_1:
14 dados_1 = handler_1.readlines()[valor_1:valor_2]
15 dados_elem_1 = [float(str(linha[:-1],'utf-8').split(',',
16)[elem]) for linha in dados_1]
17 nome_1 = url_1.split('=')[-1]
18
19 with urllib.request.urlopen(url_2) as handler_2:
20 dados_2 = handler_2.readlines()[valor_1:valor_2]
21 dados_elem_2 = [float(str(linha[:-1],'utf-8').split(',',
22)[elem]) for linha in dados_2]

```

```

20 nome_2 = url_2.split('=')[-1]
21
22 corre = pearson(dados_elem_1, dados_elem_2)
23 mostra(dados_elem_1,dados_elem_2,nome_1, nome_2, valor_1,
24 valor_2)
25
26
27 def mostra(dados_1, dados_2,nome_1,nome_2, data_1, data_2):
28 etiqueta = nome_1 + ' vs ' + nome_2 + ':' + '(' + str(
29 data_1) + ' - ' + str(data_2) + ')'
30 plt.plot(dados_1,dados_2,'ro', label= etiqueta)
31 plt.xlabel(nome_1)
32 plt.ylabel(nome_2)
33 plt.title('Pearson')
34 plt.legend(loc=0)
35 plt.show()
36
37 if __name__ == '__main__':
38 url_apple = 'http://ichart.finance.yahoo.com/table.csv?s=
39 AAPL'
40 url_coke = 'http://ichart.finance.yahoo.com/table.csv?s=
41 Coke'
42
43 print(compara(url_apple,url_coke,1,24,4))

```

Listagem 7.12: Compara cotações

O programa da listagem 7.12 exemplifica o caso concreto das cotações de fecho da Apple e da Coca-cola. Permite visualizar o resultado graças à função **mostra**. Está escrito para poder ser usado com qualquer empresa e entre qualquer período. Devemos ter em atenção que a primeira linha do ficheiro deve ser descartada, pelo que o período inicial terá que ser sempre maior ou igual a 1 (segunda linha do ficheiro). Notar uma vez mais a necessidade de fazer a descodificação para cadeia de caracteres (linhas 14 e 19).

A figura 7.8 mostra um dos resultados da análise para o período 1 a 120. O valor do CCP é de 0.523. Para o leitor fica a tarefa de estudar diferentes janelas temporais.

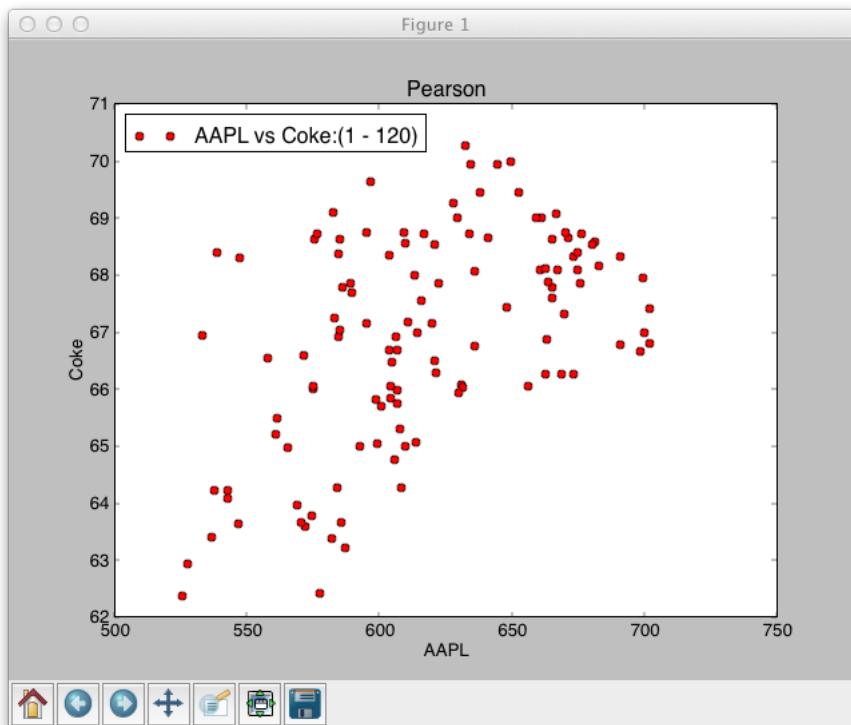


Figura 7.8: A Apple e a Coca-cola

## Sumário

Neste capítulo estivemos concentrados no conceito de ficheiro e das operações que com eles podemos fazer: leitura, escrita e navegação. Usámos esta oportunidade para introduzir a instrução `with`, o tipo de dados `bytes` e ainda os módulos `csv` e `urllib`.

## Teste os seus conhecimentos

Tente responder às seguintes questões, tratadas neste capítulo.

- Quais as características dos ficheiros?
- Como identifico um ficheiro vazio?
- Qual é o construtor do tipo ficheiro?
- Qual a diferença entre `seek` e `tell`?

- Só existem ficheiros de texto? Se não, que outros tipos e quais as diferenças?
- Os ficheiros de texto são uma longa cadeia de caracteres. Existe a operação de fatiamento para ficheiros?
- É possível ter um ficheiro aberto simultaneamente para leitura e para escrita?
- Qual a importância de usar sempre o método **close**?
- Para que serve a construção **with**
- O que é o tipo dados **bytes**
- O que é e para que serve o módulo **csv**
- O que é e para que serve o módulo **urllib**

## Exercícios

Alguns destes exercícios requerem alguns ficheiros específicos. Podem ser encontrados no sítio do livro em .

### Exercício 7.1 F

Desenvolva um programa que crie um ficheiro **primeiro.txt** com o seguinte conteúdo “Acabei de criar o meu primeiro ficheiro em Python.”. Depois de criado, use um editor para ver o que foi guardado. Pode escolher se, e como, divide o texto em linhas.

### Exercício 7.2 F

Desenvolva um programa que mostre uma sequência de caracteres, em número pré-definido, presentes no ficheiro **primeiro.txt** a partir de uma dada posição de referência.

### Exercício 7.3 F

Desenvolva um programa que adicione uma nova linha ao ficheiro **primeiro.txt** contendo a data de hoje, mas sem criar um novo ficheiro.

### Exercício 7.4 M

Desenvolva um programa que permita identificar se um ficheiro contém números. O resultado do programa deve ser uma lista dos números existentes

no ficheiro. Pode testar com um ficheiro de texto criado por si com um editor.

### Exercício 7.5 Módulo matplotlib M

Desenvolva um programa que analise as temperaturas médias de várias cidades portuguesas, guardadas no ficheiro **temperaturas.txt**<sup>5</sup>, determine os valores máximo e mínimo, e mostre o gráfico do resultado. Cada linha representa os dados referentes a uma cidade.

### Exercício 7.6 M

Escreva um programa que crie uma cópia de um ficheiro. O nome dos ficheiros origem e destino devem ser pedidos ao utilizador e, de seguida, deve ser chamada uma função que faça a cópia. Os nomes dos ficheiros deverão ser argumentos da função. Pode testar com um ficheiro de texto criado por si com um editor.

### Exercício 7.7 Módulo random Módulo turtle M

Desenvolva um programa que coloca, num ficheiro a criar, pares de números. Esses pares devem estar um por linha e ser gerados aleatoriamente. Admita que esses valores correspondem aos números (entre 1 e 6) de dois dados. Leia depois o ficheiro, interprete cada par como coordenadas num espaço 2D, e use o módulo turtle para desenhar a figura que resulta de unir esses pontos respeitando a ordem em que aparecem no ficheiro..

### Exercício 7.8 Módulo matplotlib Módulo turtle M

Usando o ficheiro de dados do exercício 7.8.2, desenvolva um programa que permita analisar a frequência dos valores que existem no ficheiro. A informação deve ser representada visualmente, e deve mostrar o número de vezes que cada um dos valores saiu. Utilize o módulo **matplotlib** ou o módulo **turtle** para fazer um diagrama de barras que represente o número de vezes que um valor saiu.

### Exercício 7.9 Módulo matplotlib Módulo turtle M

Usando o ficheiro obtido para os problemas 7.8.2 e 7.8.2, desenvolva um programa que permita analisar a frequência da **soma** de dois valores. A informação deve ser representada visualmente, e deve ser mostrada em termos de percentagens. Utilize o módulo **matplotlib** ou o módulo **turtle**, para fazer um diagrama de barras das **percentagens** referentes a soma dos dois

<sup>5</sup>Disponível em <http://ipr.net>.

valores.

### Exercício 7.10 Módulo matplotlib Módulo turtle M

Desenvolva um programa que permita analisar a frequência dos caracteres que existem num ficheiro. A informação deve ser mostrada visualmente. Pode socorrer-se do módulo **matplotlib** ou do módulo **turtle**. Pode testar com um ficheiro de texto criado por si com um editor.

### Exercício 7.11 M

Escreva um programa que permita gerir vendas a dinheiro. Num ficheiro de texto tem informação sobre vendas já efectuadas. Cada linha do ficheiro tem informação sobre uma transacção, na forma: número da transacção, nome da empresa, número de contribuinte, data e valor. O seu programa deve obter os elementos de uma nova transacção e actualizar o ficheiro. Deve também imprimir um documento onde constem os elementos referidos e ainda o nome do funcionário que fez a venda. A listagem ilustra o pretendido para a impressão.

```

1 Venda a Dinheiro No 100
2 -----
3 Empresa: Vendas&Vendas
4 N.C.: 987654321
5 Data: 6/Out/2008
6 Valor: 100.20 Euros
7 Vendedor: Manuel Antunes

```

**Exercício 7.12 M** Admita que tem um **ficheiro** de texto em memória. Pretende-se um programa que leia o ficheiro e construa um **dicionário** em que as chaves são inteiros e os valores a **lista** das palavras cujo comprimento é igual ao valor da chave. A título de **exemplo** do que se pretende, mostramos o resultado para o ficheiro cujo conteúdo é o seguinte (**fich\_dic** é o nome do programa que tem que implementar.):

```

1 isto pode ser uma conversa
2 banal, entre duas pessoas
3 banais e que, pasme-se, adoram dizer
4 banalidades! Entendido?

```

```

1 >>> print fich_dic('/tempo/data/ex_normal.txt')
2 {1: ['e'], 3: ['ser', 'uma', 'que'], 4: ['isto', 'pode', 'duas'],
 5: ['banal', 'entre', 'dizer'], 6: ['banais', 'adoram']}

```

```
[], 7: ['pessoas'], 8: ['conversa', 'pasme-se'], 9: ['Entendido'], 11: ['banalidades']}
```

<sup>3</sup> >>>

**Exercício 7.13 M** Admita que tem um ficheiro onde estão guardados números **inteiros**. Não sabe quantos existem por linha, nem quantos existem no total. Agora o que sabe é que podem existir números **repetidos** espalhados pelo ficheiro. Pretende-se um programa que leia o conteúdo do ficheiro e crie um **dicionário** em que as chaves são os números e os valores o número de vezes em que cada um aparece no ficheiro.

**Exercício 7.14 M** Suponha que tem um ficheiro com informação sobre dados pessoais. Mais concretamente em cada linha do ficheiro tem o nome, apelido, idade, código da profissão e código do estado civil. A figura 7.9 ilustra uma situação possível .

|                 |    |     |   |
|-----------------|----|-----|---|
| Ernesto Costa   | 55 | 102 | 1 |
| Ana Paz         | 44 | 411 | 2 |
| Carlos Ferreira | 20 | 203 | 2 |

Figura 7.9: Ficheiro de dados: entrada

Escreva um programa que leia este ficheiro e produza um novo no qual o nome e apelido foram substituídos pelas respectivas iniciais, a idade se manteve, e os códigos de profissão e estado civil alterados para os respectivos nomes. A figura 7.10 mostra a saída resultante de aplicar o programa ao ficheiro de entrada da figura 7.9. A relação entre os códigos e os nomes correspondentes (por exemplo, 102 corresponde a professor, 1 corresponde a casado) estão guardadas em dois **dicionários**. É evidente que pode definir a correspondência do modo que entender.

|    |    |           |          |
|----|----|-----------|----------|
| EC | 55 | Professor | Casado   |
| AP | 44 | Advogado  | Solteiro |
| CF | 20 | Estudante | Solteiro |

Figura 7.10: Ficheiro transformado

**Exercício 7.15 D**

Pretendemos saber se existe alguma **correlação** entre o crescimento do produto interno bruto ( **PIB**) e o **desemprego**, na zona Euro. Para isso temos dois ficheiros com essa informação. A informação cobre anos distintos nos dois ficheiros, e a informação presente tem periodicidade diversa.

```

1 # Desemprego na Zona euro
2 # Fonte: http://www.economagic.com
3 1993 01 1512·202
4 1993 02 4112·391
5 1993 03 8812·655
6 1993 04 4512·842
7 1993 05 2913·036
8 ...

```

Listagem 7.13: 'Desemprego'

```

1 # Crescimento do Produto Interno Bruto : Zona EURO
2 # fonte http://www.economagic.com
3 1996 01 502·7
4 1996 02 262·0
5 1996 03 11·6
6 1996 04 471·4
7 1997 01 591·4
8 ...

```

Listagem 7.14: 'Crescimento do PIB'

Desenvolva um programa que **dado um período de tempo**, analisa a informação, calcula o coeficiente de correlação e visualiza os dados.

### Exercício 7.16 D

Suponha que é dono de uma empresa e tem uma pequena base de dados com informações sobre os seus clientes. Essa informação, guarda num ficheiro, inclui, entre outras coisas, o nome, a data de nascimento, morada e número de telefone. Imagine que quer enviar aos clientes nascidos antes de um dado ano uma carta. A diferença nas cartas é apenas no cabeçalho, que deve personalizar pelo nome e pela morada. Admita que antes de enviar as cartas as guarda todas em ficheiros separados. Desenvolva a respectiva aplicação, isto é, um programa que leia o modelo da carta de um ficheiro, determine quais os clientes a quem deve enviar a carta, produza acarta para cada cliente e a guarde num ficheiro individual.

### Exercício 7.17 D

Admita que tem uma pequena base de dados com informação sobre a sua biblioteca de músicas. Cada música deve ter como informação: intérprete, título, tipo de música, duração e se está emprestado ou não. Desenvolva uma aplicação que lhe permita:

- introduzir um novo título
- marcar uma música como emprestada
- mostrar todas as músicas de um certo tipo

**Exercício 7.18 D** Admita que tem um ficheiro em que **cada linha** contém um endereço de Internet com o **URL** para a página pessoal de um utilizador, no formato habitual que o exemplo abaixo ilustra.

<http://eden.dei.uc.pt/~ernesto>

Notar que a parte final é sempre o nome do utilizador precedido pelo *til*. Suponha também que definiu um dicionário onde a cada nome de utilizador faz corresponder uma lista com o nome completo do utilizador e o seu endereço de correio electrónico (ver exemplo).

```
1 {'ernesto': ['Ernesto Costa', 'ernesto@dei.uc.pt'], ...}
```

Escreva um programa que, dados um dicionário e o endereço de um ficheiro, cada um com a informação acima descrita, faça a leitura deste último e, a partir dos dados obtidos, extraia o nome dos utilizadores, usando-o em conjunção com o dicionário para devolver uma lista com os **nomes completos** dos utilizadores presentes no ficheiro, **ordenada** por ordem alfabética.

**Exercício 7.19 Módulo csv D** O ficheiro *zoo.csv*<sup>6</sup> tem informação diversa sobre animais. Em cada linha encontra a descrição de um animal específico, o nome na primeira posição e a sua classificação na última. Leia o ficheiro e construa uma estrutura do tipo dicionário em que as chaves são a classe e o valor a lista com os nomes dos animais da classe.

**Exercício 7.20 Módulo urllib D**

Os ficheiros HTML têm no seu início um conjunto importante de informações. Eis um exemplo retirado do sitio <http://www.python.org>.

<sup>6</sup>Pode ser obtido no sitio da cadeira em <http://iprp.net>.

```
1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
2 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang=
 "en">
3
4 <head>
5 <meta http-equiv="content-type" content="text/html; charset=
 utf-8" />
```

Uma delas é o tipo de codificação usada no texto, identificado através de **charset**. Escreva um programa que lhe permita ir buscar a um ficheiro HTML essa informação.

# Metodologia da Programação

## Objectivos

- ✓ Revisão dos diferentes aspectos da linguagem **Python**
- ✓ Apresentação de princípios de construção de programas
- ✓ Exemplos de aplicação

### 8.1 Introdução

A actividade de programação está a meio caminho entre a ciência e a arte. Ciência, porque exige disciplina e rigor, arte, porque motiva para (e potencia) a actividade criativa. Para uns um programador é como um escultor que passo a passo transforma uma pedra numa estátua esbelta. Para outros o programador é alguém quem brinca com blocos de diferentes formas e funcionalidades como se fossem peças de *Lego*. Realmente programar não é fácil. Temos que por em jogo simultaneamente conhecimentos (1) sobre o domínio do problema, (2) sobre a linguagem de programação a usar e (3) sobre o processo de construção da solução. Mas toda esta complexidade pode ser dominada. Neste capítulo vamos estar concentrados na questão (3), sem prejuízo de continuarmos a olhar para as outras duas questões. No centro da metodologia que vamos propor está a identificação do modo como podemos decompor um problema em sub-problemas. No final queremos que os nossos programas tenham um conjunto de características bem definidas, como por exemplo, estejam correctos, sejam de fácil leitura, fáceis de manter, que exigam poucos recursos (espaço de memória, tempo de computação). Grande parte das vezes estes objectivos são contraditórios entre si. Por exemplo,

acontece com frequência que a legibilidade do código tenha que ser sacrificada em detrimento da eficiência. A única coisa de que não se pode abdicar é da ... correção. A construção descendente de programas ajuda a conseguir esse objectivo de correção, ao mesmo tempo que permite isolar as decisões de projecto que nos levaram até ao código final. É isso que procuraremos ilustrar recorrendo a exemplos simples.

## 8.2 Um Problema Simples

Suponhamos que alguém nos diz que pretende um programa para desenhar o objecto que vemos na figura 10.16.

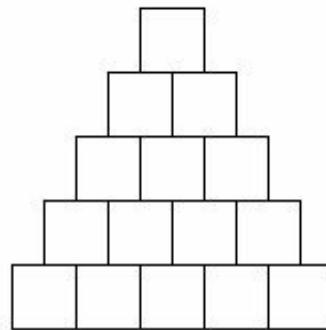


Figura 8.1: Que linda pirâmide

A primeira questão que seguramente nos colocamos é a de saber se é **exactamente esta** pirâmide cujo desenho se pretende. Dir-nos-ão que não. A altura da pirâmide pode ser qualquer. Feita esta clarificação o nosso segundo pensamento deverá ser algo como: quais são os **dados** do problema? Qual o **resultado** pretendido? O que sei sobre o **domínio** que me possa ajudar a transformar os dados no resultado? É aqui que temos que começar a alinhar umas ideias. Fazer um desenho pode ajudar. Assim a figura 10.17 tenta clarificar a questão.

Vamos supor que decidimos recorrer ao módulo **turtle**<sup>1</sup> para efectivar o

---

<sup>1</sup>Neste momento em todo o rigor a decisão pelo módulo **turtle** não era ainda necessária. Igualmente o facto de apresentarmos a solução já em **Python** não é fundamental. No nosso caso consideramos que a linguagem **Python** é a que melhor permite expressar psedo-código!

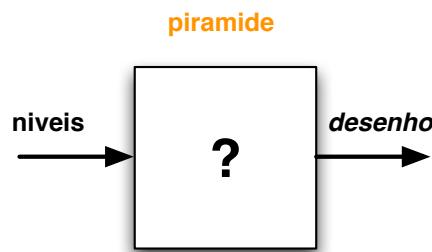


Figura 8.2: Um desenho diz mais do que mil palavras?

desenho. Isso permite um primeiro esboço do programa como se mostra na listagem 8.1.

```

1 from turtle import *
2
3 def piramide(niveis):
4 # desenha pirâmide
5 return 'Fim'

```

Listagem 8.1: 'Devagar se vai ao longe ...'

Chegou a altura de lembrar uma frase famosa do pai da linguística moderna, Ferdinand de Saussure, que dizia que é o ponto de vista que cria o objecto. Assim vou olhar para a pirâmide **como se fosse** uma sequência ordenada de linhas que crescem de cima para baixo. Implícito nesta forma de ver estão decisões (vou desenhar de cima para baixo) e abstrações (não vejo os quadrados mas apenas linhas de tamanho distinto). A partir daqui é fácil chegar à conclusão de que uma maneira simples de resolver o problema é dividi-lo em sub-problemas em que cada um consiste no desenho de uma linha. Daí a segunda versão do nosso programa.

```

1 from turtle import *
2
3 def piramide(niveis):
4 # desenha pirâmide
5 for i in range(1, niveis + 1):
6 desenha_linha(i)
7 return 'Fim'

```

Listagem 8.2: 'Continuemos...'

Não é difícil entender o recurso ao ciclo **for**: afinal trata-se de uma ação que é repetida um número conhecido de vezes! A figura 8.3 ilustra o

mecanismo de decomposição do problema e delegação da solução parcial.

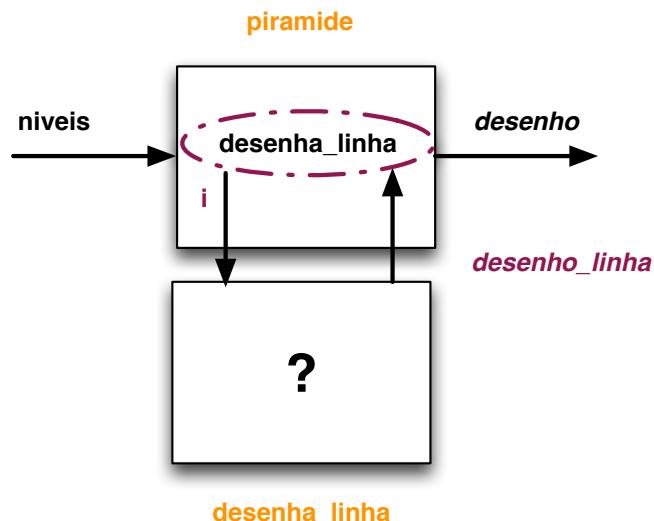


Figura 8.3: Onde estamos ...

É tempo de pensarmos nas linhas. Agora podemos olhar mais de perto e *reparar* que as linhas são feitas de ... quadrados! Como anteriormente é fácil concluir (basta olhar para a figura 10.16) que na linha  $i$  são desenhados  $i$  quadrados. Então já não nos espantaremos com a proposta de programa 8.3.

```

1 from turtle import *
2
3 def piramide(niveis):
4 # desenha pirâmide
5 for i in range(1, niveis + 1):
6 desenha_linha(i)
7 return 'Fim'
8
9 def desenha_linha(linha):
10 # desenha linha
11 for i in range(linha):
12 desenha_quadrado()
13 return 'Fim'

```

Listagem 8.3: 'Chegaram os quadrados ...'

Como desenhar um quadrado em **turtle**? Todos sabemos, espero bem. Eis uma hipótese de código:

```

1 def desenha_quadrado():
2 for i in range(4):
3 fd(20)
4 rt(90)
5 return 'Fim'
```

Listagem 8.4: 'Um quadrado de lado 20 ...'

Mas e se quisermos desenhar sucessivas pirâmides com níveis e lados distintos. Bom, uma solução muito simples será associar um nome ao valor do lado e depois usar o nome na definição<sup>2</sup>. Está na hora de juntar tudo e testar o programa. A listagem 8.5 mostra o programa, enquanto a figura 8.4 dá uma ideia das dependências entre partes do programa.

```

1 from turtle import *
2
3 lado = 20
4
5 def piramide(niveis):
6 # desenha pirâmide
7 for i in range(1, niveis + 1):
8 desenha_linha(i)
9 return 'Fim'
10
11 def desenha_linha(linha):
12 # desenha linha
13 for i in range(linha):
14 desenha_quadrado()
15 return 'Fim'
16
17 def desenha_quadrado():
18 for i in range(4):
19 fd(lado)
20 rt(90)
21 return 'Fim'
```

Listagem 8.5: 'Tudo junto ...'

---

<sup>2</sup>O leitor atento poderá perguntar porque não fazemos como no caso dos níveis tornando o **lado** um **parâmetro** do programa **desenha\_quadrado**? Podíamos fazer, sim senhor. Isso obriga a outras mudanças no resto do programa. Deixamos ao leitor a tarefa de seguir por esse caminho.

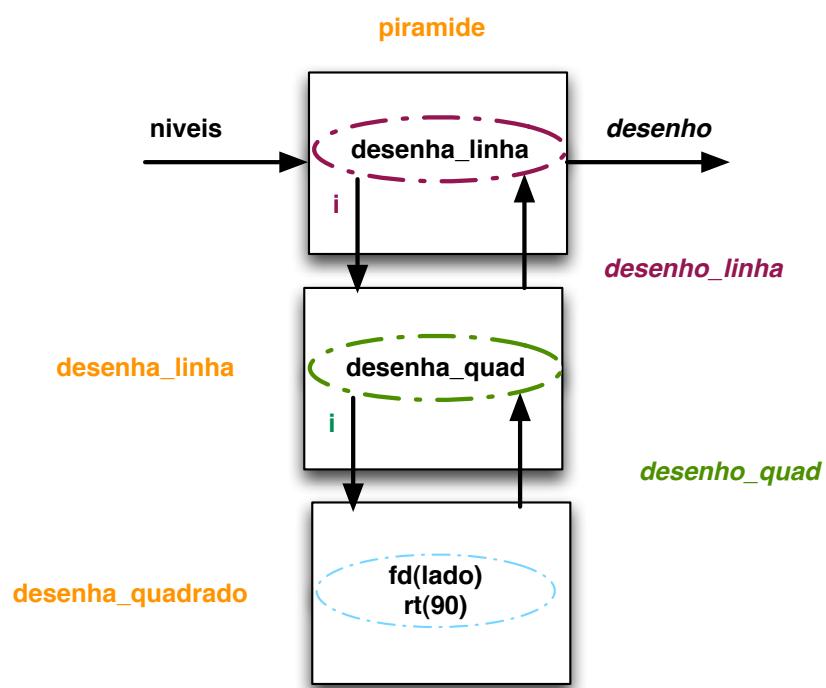


Figura 8.4: O primeiro produto

Quando executamos o programa acontece no entanto que vemos a nossa tartaruga, à maneira de Sísifo, a desenhar sempre o mesmo quadrado apresentando no final o resultado que a figura 8.5 ilustra. Aqui tem que entrar o nosso conhecimento sobre o módulo **turtle**. Acontece que a tartaruga é colocada inicialmente no ponto de coordenadas (0,0). Desenhado o primeiro quadrado da primeira linha a posição final da tartaruga é a mesma, pelo que ele começa a desenhar o quadrado seguinte na mesma posição. Temos portanto duas questões distintas: nem desenha linha a linha, nem, para uma dada linha desenha a sucessão de quadrados. Este é um problema que já sabíamos que era necessário resolver. Apenas não nos quisemos preocupar logo com esta questão. Está agora na hora da o fazer.

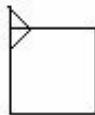


Figura 8.5: Mas que lindo ... quadrado!

Comecemos com o problema de desenhar uma linha de quadrados. A questão é que precisamos de fazer avançar a posição da tartaruga cada vez que se pretender desenhar um quadrado novo. Tratando-se de uma linha horizontal basta para tal alterar a coordenada **x** da tartaruga. De quanto? Bom pela figura 10.16 é evidente que esse valor é igual ao valor do lado! Feitas as alterações temos uma nova versão coma a listagem 8.6 ilustra.

```

1 from turtle import *
2
3 lado = 40
4 # posição inicial
5 pu()
6 setx(0)
7 sety(0)
8 pd()
9
10 def piramide(niveis):
11 # desenha pirâmide
12 for i in range(1, niveis + 1):
13 desenha_linha(i)
14 return 'Fim'
```

```

15
16 def desenha_linha(linha):
17 # desenha linha
18 for i in range(linha):
19 desenha_quadrado()
20 # nova posição
21 pu()
22 setx(xcor() + lado)
23 pd()
24 return 'Fim'
25
26 def desenha_quadrado():
27 for i in range(4):
28 fd(lado)
29 rt(90)
30 return 'Fim'
31
32 def main():
33 piramide(3)
34 raw_input()
35
36 if __name__ == '__main__':
37 main()

```

Listagem 8.6: 'Um problema já está resolvido ...'

Ao executarmos o programa (com o número de níveis igual a 3) o nosso programa desenha **numa mesma linha** seis quadrados, como se pode ver na figura 8.6.

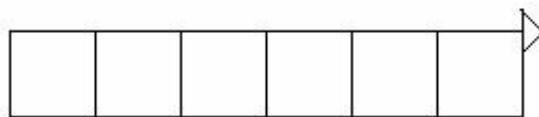


Figura 8.6: Tanto quadrado ... fora do sítio!

Este resultado era de esperar. É desenhado o quadrado da primeira linha, mais dois da segunda e mais três da terceira. É manifesto que temos que fazer o nosso programa *mudar de linha* no momento certo. E esse instante é sempre

que eu desenhar uma linha. Qual o valor que se deve *descer*? O valor do lado. Esta é a parte fácil.

```

1 def piramide(niveis):
2 # desenha pirâmide
3 for i in range(1, niveis + 1):
4 desenha_linha(i)
5 # actualiza posição
6 pu()
7 sety(ycor() - lado)
8 pd()
9 return 'Fim'
```

Listagem 8.7: 'Mudar de linha ...'

Mas se corremos de novo o programa o que sê é o que mostramos na figura 8.7. Está melhor mas não chega!

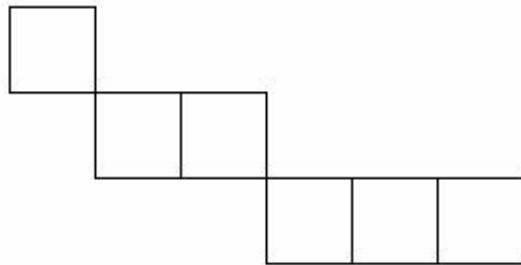


Figura 8.7: Mudar de linha só não chega ...

A figura 8.7 mostra que é preciso fazer *recuar* a coordenada **x** cada vez que se desenha uma linha. Mas quanto? Bom analisando a figura vemos que tal depende do tamanho da linha anteriormente desenhada. Daí que a solução correcta seja a que se apresenta na listagem 8.8.

```

1 from turtle import *
2
3 lado = 40
4 # posição inicial
5 pu()
6 setx(0)
7 sety(0)
8 pd()
```

```

9
10 def piramide(niveis):
11 # desenha pirâmide
12 for i in range(1, niveis + 1):
13 desenha_linha(i)
14 # actualiza posição
15 pu()
16 sety(ycor() - lado)
17 setx(xcor() - i * lado - float(lado)/2)
18 pd()
19 return 'Fim'
20
21 def desenha_linha(linha):
22 # desenha linha
23 for i in range(linha):
24 desenha_quadrado()
25 # nova posição
26 pu()
27 setx(xcor() + lado)
28 pd()
29 return 'Fim'
30
31 def desenha_quadrado():
32 for i in range(4):
33 fd(lado)
34 rt(90)
35 return 'Fim'
36
37 def main():
38 piramide(3)
39 ht()
40 raw_input()
41
42 if __name__ == '__main__':
43 main()

```

Listagem 8.8: 'Versão final'

Executando o programa obtemos o lindo resultado da figura 8.8.

Podemos continuar a melhorar o programa introduzindo variações a nosso gosto e relacionadas, neste caso, com a possibilidades oferecidas pelo módulo **turtle**. Ilustramos uma situação (listagem 8.9 e figura 8.9) em que introdu-

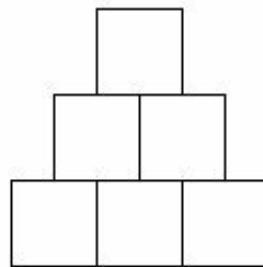


Figura 8.8: Agora sim!

zimos um pouco de cor e aumentámos a dimensão do traço da caneta.

```

1 from random import randint
2 from turtle import *
3
4 lado = 40
5 # posição inicial
6 pu()
7 setx(0)
8 sety(0)
9 pd()
10
11 def piramide(niveis):
12 # desenha pirâmide
13 for i in range(1, niveis + 1):
14 desenha_linha(i)
15 # actualiza posição
16 pu()
17 sety(ycor() - lado)
18 setx(xcor() - i * lado - float(lado)/2)
19 pd()
20 return 'Fim'
21
22 def desenha_linha(linha):
23 # desenha linha
24 for i in range(linha):
25 #desenha_quadrado() # preto e branco
26 quadrado() # a cores

```

```
27 # nova posição
28 pu()
29 setx(xcor() + lado)
30 pd()
31 return 'Fim'
32
33 def desenha_quadrado():
34 for i in range(4):
35 fd(lado)
36 rt(90)
37 return 'Fim'
38
39 def quadrado():
40 # desenha
41 r=randint(0,255)
42 g=randint(0,255)
43 b=randint(0,255)
44 fillcolor(r,g,b)
45 fill(True)
46 for i in range(4):
47 fd(lado)
48 rt(90)
49 fill(False)
50 return 'Fim'
51
52 def main():
53 pencolor('blue')
54 pensize(3)
55 colormode(255)
56 piramide(5)
57 ht()
58 raw_input()
59
60 if __name__ == '__main__':
61 main()
```

Listagem 8.9: 'Versão colorida'

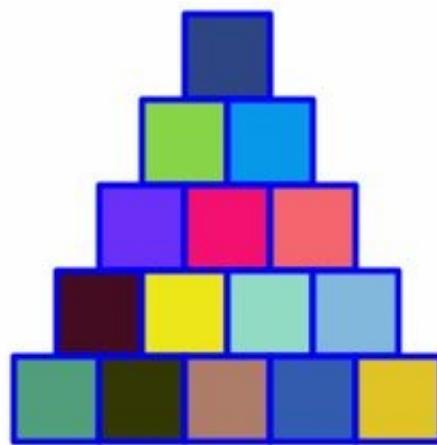


Figura 8.9: Pirâmide colorida

## O ponto de vista cria o objecto

Na secção 8.2 citámos Ferdinand de Saussure e a sua frase famosa *é o ponto de vista que cria o objecto*. Usámos a citação para explicar como é que o modo como olhamos para um problema determina em parte a forma como o resolvemos. No exemplo da secção 8.2 vimos primeiro uma pirâmide feita de linhas, linhas que mais tarde apareceram como sequências de quadrados. Mas imaginemos que olhávamos para a figura de outro modo. Suponhamos que consideramos que afinal estamos na presença de diagonais de tamanho decrescente (da esquerda para a direita), como ilustra a figura 8.10.

Seguindo o método já utilizado fácil é chegar a um primeiro esboço de programa como o que apresentamos na listagem 8.10.

```

1 from turtle import *
2
3 lado=30
4 setx(0)
5 sety(0)
6
7 def pir(niv):
8 for i in range(niv,0,-1):
9 diagonal(i)
10
11 def diagonal(n):
```

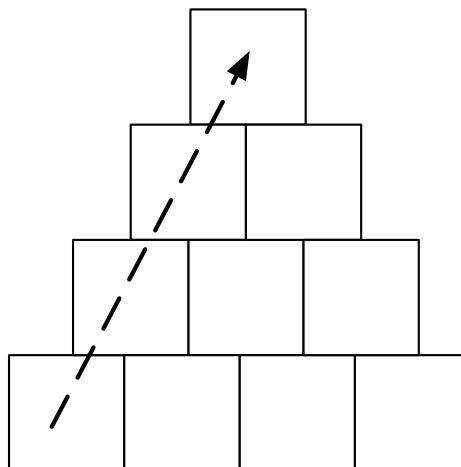


Figura 8.10: A pirâmide vista de outro modo ...

```

12 for i in range(n):
13 quadrado()
14
15
16 def quadrado():
17 for i in range(4):
18 fd(lado)
19 rt(90)

```

Listagem 8.10: 'Primeira aproximação'

Se mandarmos executar é evidente que não vai dar por razões semelhantes à versão baseada em linhas horizontais. É preciso que as diagonais *avancem* da esquerda para a direita e que sejam formadas por quadrados. Isso leva-nos à versão completa dada na listagem 8.11.

```

1 from turtle import *
2
3 lado=30
4 setx(0)
5 sety(0)
6
7 def pir(niv):
8 for i in range(niv,0,-1):
9 diagonal(i)
10 pu()

```

```

11 setx(xcor() - (i * float(lado)/2) + lado)
12 sety(ycor() - i * lado)
13 pd()
14
15 def diagonal(n):
16 for i in range(n):
17 quadrado()
18 pu()
19 setx(xcor() + float(lado)/2)
20 sety(ycor() + lado)
21 pd()
22
23 def quadrado():
24 for i in range(4):
25 fd(lado)
26 rt(90)
27
28
29 def main():
30 pir(5)
31 ht()
32 raw_input()
33
34
35
36 if __name__ == '__main__':
37 main()

```

Listagem 8.11: 'Versão final'

Fica ao cuidado do leitor explorar outras *visões*.

### 8.3 Uma questão de consenso

Vamos agora mudar de domínio, entrando no território da Biologia, mas mantendo a mesma ideia de introduzir princípios sólidos de construção de programas. Comecemos pelo enunciado do problema. Em Biologia existe algo que se chama **sequência de consenso**. Em termos simples (simplistas?) uma sequência de consenso é um **padrão** de ADN presente na região dos promotores de uma extensa cadeia de ADN e que determina a possibilidade de ligação com determinadas enzimas para se dar início ao processo de transcrição do

ADN ele próprio uma das etapas da transcrição genética. Dizemos padrão pois o que efectivamente ocorre na região do promotor não é exactamente a sequência de consenso mas uma sequência muito "parecida". Um Informático diria simplesmente que uma sequência de consenso é uma palavra (ou **cadeia de caracteres**) sobre um alfabeto de quatro letras  $\Sigma = \{A, T, C, G\}$ . Cada uma dessas letras simboliza uma base azotada: Adenina, Timina, Citosina e Guanina. O nosso exercício de programação pode ser **especificado** de modo simples: dada um conjunto de sequências de ADN, todas de igual tamanho, construir uma nova sequência do mesmo tamanho em que a base presente numa dada posição será a base que aparece com maior frequência nessa posição em todas as sequências iniciais. Vamos chamar a essa sequência, sequência de consenso.

Para clarificar apresentamos um exemplo.

Para a entrada:

```
S1='ATTC'

S2='ACTG'

S3='GGGG'

S4='ATCG'
```

a saída correspondente deve ser:

```
S='ATTG'
```

## A solução

Dissemos na introdução deste capítulo, que um programador tanto é visto como um escultor como alguém que brinca com *Legos*. Isso corresponde a dois estilos de programação: do topo para a base, por vezes designada por **Programação Descendente** ou da base para o topo, e aqui falamos de **Programação Ascendente**. Com o tempo e a experiência um programador socorre-se das duas abordagens. Referimos ainda que programar pressupõe três aspectos:

- Conhecimento sobre o domínio do problema
- Conhecimento sobre a linguagem de programação
- Conhecimento sobre o processo de programação

Para este problema o primeiro aspecto é pouco relevante. Trata-se de cadeias de caracteres que representam pedaços de ADN. Já os outros dois são fundamentais. Iremos começar por nos concentrar no último: o processo.

A primeira coisa em que um programador pensa, conhecida a especificação, é se pode construir uma descrição **abstracta** da solução. Nessa tarefa o programador deve tentar afastar ao máximo o seu pensamento da linguagem de programação concreta que vai usar. Admitamos que decidimos construir a sequência de consenso, posição a posição<sup>3</sup>. Por cada posição teremos que analisar todas as sequências e determinar a base mais frequente. Sabido qual é, acrescentamos à nossa solução parcial. Podemos descrever este processo de modo simples.

```

1 def consensus(lseq):
2 """
3 Constrói a sequência de consenso a partir de uma lista de
4 sequências
5 de ADN de igual comprimento.
6 """
7
8 # 1. Inicializa a sequência de consenso
9 # 2. Por cada posição da sequência
10 # 2.1. Calcula qual a base mais frequente para a
11 # posição corrente
12 # 2.2. Actualiza a sequência de consenso com a base
13 # encontrada
14 # 3. Devolve sequência de consenso completa
15 return # Sequência

```

Avancemos um pouco mais tornando claro que vamos trabalhar para uma dada posição por análise das sequências, uma a uma. Vejamos como podemos detalhar o cálculo da base mais frequente.

```

1 def consensus(lseq):
2 """
3 Constrói a sequência de consenso a partir de uma lista de
4 sequências

```

---

<sup>3</sup>Não esquecer que se trata de uma sequência, logo existe uma ordem.

```

4 de ADN de igual comprimento.
5 """
6 # 1. Inicializa a sequência de consenso
7 # 2. Por cada posição da sequência
8 # 2.1. Calcula qual a base mais frequente para a
9 posição corrente
10 # Inicializa contadores das bases
11 # Para cada sequência
12 # Determinina a base e actualiza o respectivo
13 contador
14 # Determina a base que ocorre mais vezes
15 # 2.2. Actualiza a sequência de consenso com a base
 encontrada
3. Devolve sequência de consenso completa
return # Sequência

```

Do código resulta fácil ver quais foram as opções tomadas: usar contadores para as bases que vão sendo actualizados à medida que determinamos qual a base presente numa dada posição de cada sequência.

Podemos começar a escolher a forma de guardar os objecto manipulados. A sequência de consenso será uma **cadeia de caracteres**. Ao olharmos para a figura 8.12 o leitor dirá que avançámos pouco. É verdade. Mas estamos a vançar de forma segura.

```

1 def consensus(lseq):
2 """
3 Constrói a sequência de consenso a partir de uma lista de
4 sequências
5 de ADN de igual comprimento.
6 """
7 # 1. Inicializa a sequência de consenso
8 cons = ''
9 # 2. Por cada posição da sequência
10 # 2.1. Calcula qual a base mais frequente para a
11 posição corrente
12 # Inicializa contadores das bases
13 # Para cada sequência
14 # Determinina a base e actualiza o respectivo
15 contador
16 # Determina a base que ocorre mais vezes

```

```

14 # 2.2. Actualiza a sequência de consenso com a base
15 # encontrada
16 cons = cons + base # A base é uma cadeia de
17 # comprimento um
3. Devolve sequência de consenso completa
return cons

```

Listagem 8.12: Usar cadeias de caracteres

Qual deve ser o nosso próximo passo? O mais importante consiste em usar um **dicionário** para guardar a informação sobre o número de ocorrências de cada base azotada numa dada posição para todas as sequências. Porquê um dicionário? Bem, sabíamos que precisávamos de estabelecer uma correspondência entre as bases e o seu número de ocorrências. A única estrutura em Python que nos permite fazer isso de modo eficiente e elegante são os dicionários. Aqui entra em jogo claramente o nosso conhecimento da linguagem (ponto (2) dos pré-requisitos para bem programar). Noutra linguagem poderia ser outra a opção. Isto porque esta opção faz com que a noção de ordem desapareça o que pode (e vai) tornar o problema de encontrar a base mais frequente uma questão mais delicada. Vejamos então como ficamos.

```

1 def consensus(lseq):
2 """
3 Constrói a sequência de consenso a partir de uma lista de
4 sequências
5 de ADN de igual comprimento.
6 """
7
8 # 1. Inicializa a sequência de consenso
9 cons = ''
10 # 2. Por cada posição da sequência
11 for pos in range(len(lseq[0])):
12 # 2.1. Calcula qual a base mais frequente para a
13 # posição corrente
14 # Inicializa contadores das bases
15 dicio_bases = {'A':0,'C':0,'T':0,'G':0}
16 # Para cada sequência
17 # Determinina a base e actualiza o respectivo

```

```

18 # 3. Devolve sequência de consenso completa
19 return cons

```

Tudo agora se precipita. Fica fácil a actualização do dicionário. No entanto deixamos ainda por resolver o problema de saber como é que retiro do dicionário a base mais frequente. Fica para tratarmos a seguir.

```

1 def consensus(lseq):
2 """
3 Constrói a sequência de consenso a partir de uma lista de
4 sequências
5 de ADN de igual comprimento.
6 """
7
8 # 1. Inicializa a sequência de consenso
9 cons = ''
10 # 2. Por cada posição da sequência
11 for pos in range(len(lseq[0])):
12 # 2.1. Calcula qual a base mais frequente para a
13 # posição corrente
14 # Inicializa contadores das bases
15 dicio_bases = {'A':0,'C':0,'T':0,'G':0}
16 # Para cada sequência
17 for seq in lseq:
18 # Determinina a base e actualiza o respectivo
19 # contador
20 dicio_bases[seq[pos]] = dicio_bases[seq[pos]] + 1
21 # Determina a base que ocorre mais vezes
22 base = max_ocorre(dicio_bases)
23 # 2.2. Actualiza a sequência de consenso com a base
24 # encontrada
25 cons = cons + base # A base é uma cadeia de
26 # comprimento um
27 # 3. Devolve sequência de consenso completa
28 return cons
29
30
31 def max_ocorre(dicio):
32 """Chave de máximo valor associado."""
33 pass

```

Vamos então tratar agora da questão em falta. Aqui o conhecimento da linguagem uma vez mais pode facilitar (ou dificultar, depende do ponto de vista) a nossa última tarefa. O problema que temos entre mãos é na realidade o de ordenar um dicionário por ordem decrescente dos valores para depois

ir extrair a chave respectiva. Mas como se os dicionários são objectos sem ordem? Bom, a solução mais simples é converter o dicionário para outro tipo de objecto onde a ordem seja relevante e manipular esse outro objecto. É o que fazemos com a instrução `items= list(dicio.items())`. E como calculamos o elemento de maior valor? Bem, ordenamos a lista de tuplos por ordem decrescente da segunda componente de cada tuplo. Fácil, certo?!

```

1 def max_ocorre(dicio):
2 """Chave de máximo valor associado."""
3 items = list(dicio.items())
4 items.sort(key=itemgetter(1), reverse=True)
5 return items[0][0]

```

Juntemos agora tudo. *Et voilá!*

```

1 from operator import itemgetter
2
3 def consensus(lseq):
4 """Constrói a sequência de consenso
5 a partir de uma lista de sequências de ADN
6 de igual comprimento.
7 """
8 # 1. inicializa sequência de consenso
9 cons = '' # vai ser uma string
10 # 2. por cada posição da sequência
11 for pos in range(len(lseq[0])): # entrada uma lista
12 # 2.1 calcula qual a base mais frequente para a posição
13 # corrente
14 # inicializa contadores das bases
15 dicio_bases={'A':0,'C':0,'T':0,'G':0} # uso um dicionário
16 # para contar
17 # para cada sequência
18 for seq in lseq:
19 # determina a base por cada sequência e actualiza o seu
20 # contador
21 dicio_bases[seq[pos]]=dicio_bases[seq[pos]] + 1
22 # determina a base que ocorre mais vezes
23 base=max_ocorre(dicio_bases)
24 # 2.2 actualiza a sequência de consenso na posição
25 # corrente
26 cons = cons + base # a base será um caracter
27 # 3. Devolve a sequência de consenso completa
28 return cons

```

```

25
26
27 def max_ocorre(dicio):
28 """Chave de máximo valor associado."""
29 items = list(dicio.items())
30 items.sort(key=itemgetter(1), reverse=True)
31 return items[0][0]

```

Listagem 8.13: 'Cálculo da sequência de consenso'

### Mais do que as quatro bases

Quando se resolve o problema do **alinhamento de sequências** pode acontecer que apareçam símbolos como – conhecido em inglês por *gap*. Numa base de dados sobre sequências de ADN podemos encontrar o símbolo N. Que fazer ao nosso programa para não se deixar impressionar por estes símbolos? É simples.

```

1 def consensus(lseq):
2 """Constrói a sequência de consenso
3 a partir de uma lista de sequências de ADN
4 de igual comprimento.
5 """
6 assert mesmo_tam(lseq) == True, 'CUIDADO: as sequências têm
7 que ter o mesmo tamanho'
8 # 1. inicializa sequência de consenso
9 cons = '' # vai ser uma string
10 # 2. por cada posição da sequência
11 for pos in range(len(lseq[0])): # entrada uma lista
12 # 2.1 calcula qual a base mais frequente para a posição
13 # corrente
14 # inicializa contadores das bases
15 dicio_bases={ 'A':0, 'C':0, 'T':0, 'G':0} # uso um dicionário
16 # para contar
17 # para cada sequência
18 for seq in lseq:
19 # determina a base por cada sequência e actualiza o seu
20 # contador
21 # ignora o que não interessa
22 if seq[pos] in ['-', 'N']:
23 continue
24 else:

```

```

21 dicio_bases[seq[pos]]=dicio_bases[seq[pos]] + 1
22 # determina a base que ocorre mais vezes
23 base=max_ocorre(dicio_bases)
24 # 2.2 actualiza a sequência de consenso na posição
25 corrente
26 cons = cons + base # a base será um caracter
27 # 3. Devolve a sequência de consenso completa
 return cons

```

Listagem 8.14: 'Filtrar símbolos'

Uma boa construção do programa torna a sua manutenção mais fácil! Outra prova disso? Alterar de modo a que os caracteres possam ser maiúsculos ou minúsculos. Apenas uma linha adicional de código!!

```

1 def consensus(lseq):
2 """Constrói a sequência de consenso
3 a partir de uma lista de sequências de ADN
4 de igual comprimento.
5 """
6 assert mesmo_tam(lseq) == True, 'CUIDADO: as sequências têm
7 que ter o mesm o tamanho'
8 # 1. inicializa sequência de consenso
9 cons = '' # vai ser uma string
10 # 2. por cada posição da sequência
11 for pos in range(len(lseq[0])): # entrada uma lista
12 # 2.1 calcula qual a base mais frequente para a posição
13 # corrente
14 # inicializa contadores das bases
15 dicio_bases={'A':0,'C':0,'T':0,'G':0} # uso um dicionário
16 # para contar
17 # para cada sequência
18 for seq in lseq:
19 # determina a base por cada sequência e actualiza o seu
20 # contador
21 # ignora o que não interessa
22 seq=seq.upper()
23 if seq[pos] in ['-','N']:
24 continue
25 else:
26 dicio_bases[seq[pos]]=dicio_bases[seq[pos]] + 1
27 # determina a base que ocorre mais vezes
 base=max_ocorre(dicio_bases)

```

```

25 # 2.2 actualiza a sequência de consenso na posição
26 # corrente
27 cons = cons + base # a base será um carácter
28 # 3. Devolve a sequência de consenso completa
29 return cons

```

Listagem 8.15: 'O "tamanho" não é importante'

Pode testar à vontade. Já agora e antes de abandonar esta parte: porque é que usamos a instrução `seq=seq.upper()` e não apenas `seq.upper()`? Pense um pouco antes de ir ver a nota de rodapé.<sup>4</sup>

## 8.4 Protecções

A especificação diz-nos que as sequências têm que ter todas o mesmo comprimento. E se não tiverem, o que acontece? Depende. Devido ao modo como está implementado só se dá conta do problema se existir uma sequência com menos bases do que a primeira!

```

1 if __name__ == '__main__':
2 print consensus(['ATTCG', 'AGCTTT', 'TTCCTT'])
3 print consensus(['ATTCG', 'AGCT', 'TTCC'])
4
5
6 ### Execu\c cão
7
8 PyMate r8111 running Python 2.5.1 (/usr/bin/env python)
9 >>> programar.py
10
11 ATCCT
12 IndexError: string index out of range
13
14 module body in programar.py at line 58
15 function consensus in programar.py at line 28
16 Program exited.

```

Listagem 8.16: 'Um bug??'

Isto chama a atenção para duas coisas: devemos testar e proteger os programas. Para testar uma técnica simples é conhecida por **tests unitários**. Python tem mesmo um módulo para isso, inspirado na linguagem Smalltalk

---

<sup>4</sup>Porque as cadeias de caracteres são objectos imutáveis!

e chamado de `unittest`. Está fora de questão falarmos agora dele. Vamos por um caminho mais simples refugiando-nos no comando `assert()`.

```

1 if __name__ == '__main__':
2 assert(consensus(['ATTCG', 'AGCTTT', 'TTCCTT']) == 'ATCCT')
3 # G é o que ocorre mais: na realidade é o único!
4 assert(consensus(['ATTCG', 'AGCT', 'TTCC']) == 'ATCCG')
5
6 #--- Correr e apanhar o erro
7
8 PyMate r8111 running Python 2.5.1 (/usr/bin/env python)
9 >>> protege.py
10
11 IndexError: string index out of range
12
13 module body in protege.py at line 61
14 function consensus in protege.py at line 30
15 Program exited.
```

Listagem 8.17: 'Uso de assert'

A mensagem de erro não é famosa mas podemos alterar isso.

```

1 if __name__ == '__main__':
2 assert(consensus(['ATTCG', 'AGCTTT', 'TTCCTT']) == 'ATCCT')
3 # G é o que ocorre mais: na realidade é o único!
4 try:
5 consensus(['ATTCG', 'AGCT', 'TTCC'])
6 except IndexError:
7 print '*** ERRO *** \nAs sequências têm que ter os mesmo
8 comprimento'
9 #--- Correndo
10
11 PyMate r8111 running Python 2.5.1 (/usr/bin/env python)
12 >>> protege.py
13
14 *** ERRO ***
15 As sequências têm que ter os mesmo comprimento
16 Program exited.
```

Listagem 8.18: 'Um pouco melhor'

Sempre é uma mensagem mais inteligível. Além disso podemos verificar que os testes também podem servir de **documentação** do programa. Mas

podemos colocar a protecção no **interior** do programa. Admitamos que temos uma função que nos diz se numa sequência todos os elementos têm o mesmo comprimento<sup>5</sup>. Basta alterar o código do seguinte modo.

```

1 def consensus(lseq):
2 """Constrói a sequência de consenso
3 a partir de uma lista de sequências de ADN
4 de igual comprimento.
5 """
6 # Estou protegido...
7 assert mesmo_tam(lseq) == True, 'CUIDADO: as sequências têm
8 que ter o mesmo tamanho'
9 # 1. inicializa sequência de consenso
10 cons = '' # vai ser uma string
11 # 2. por cada posição da sequência
12 for pos in range(len(lseq[0])):
13 # 2.1 calcula qual a base mais frequente para a posição
14 # corrente
15 # inicializa contadores das bases
16 dicio_bases={'A':0,'C':0,'T':0,'G':0} # uso um dicionário
17 # para contar
18 # para cada sequência
19 for seq in lseq:
20 # determina a base por cada sequência e actualiza o seu
21 # contador
22 dicio_bases[seq[pos]]=dicio_bases[seq[pos]] + 1
23 # determina a base que ocorre mais vezes
24 base=max_ocorre(dicio_bases)
25 # 2.2 actualiza a sequência de consenso na posição
26 # corrente
27 cons = cons + base # a base será um carácter
28 # 3. Devolve a sequência de consenso completa
29 return cons
30
31
32 # -- Correr
33
34
35 if __name__ == '__main__':
36 print consensus(['ATTCG', 'AGCTTT', 'TTCCTT'])
37 print consensus(['ATTCG', 'AGCT', 'TTCC'])

```

---

<sup>5</sup>Pode ser um bom exercício ....

```

32 #-- Resultado
33
34 PyMate r8111 running Python 2.5.1 (/usr/bin/env python)
35 >>> protege.py
36
37 AssertionError: CUIDADO: as sequências têm que ter o mesmo
 tamanho
38
39 module body in protege.py at line 63
40 function consensus in protege.py at line 20
41 Program exited.

```

Listagem 8.19: 'Protega-se...'

## 8.5 Novo exemplo: quadrado mágico

Um **quadrado mágico** é uma matriz quadrada  $n \times n$ , contendo **todos** os números inteiros positivos de 1 a  $n^2$ , de tal modo que a soma dos valores em cada coluna, linha ou diagonal é o mesmo. A esse número chamamos **número mágico**. Eis um exemplo, com  $n = 3$ , em que o número mágico é igual a 15.

$$\begin{bmatrix} 4 & 9 & 2 \\ 3 & 5 & 7 \\ 8 & 1 & 6 \end{bmatrix}$$

O nosso objectivo é escrever um programa que nos permita **verificar** se um dado quadrado é, ou não, mágico. No caso da resposta ser positiva, queremos também identificar o respectivo número mágico.

### 8.5.1 Princípios

Como acontece com qualquer problema de programação, o nosso primeiro passo consiste em determinar se entendemos o enunciado. De seguida precisamos identificar os dados e o resultado pretendido e como serão comunicados (ou extraídos) ao (do) programa. No nosso caso vamos adoptar um padrão clássico:

```

1 def quad_magico(quadrado):
2 ...
3 return resposta, num_magico

```

O que mostraremos a seguir ilustra como podemos resolver o problema avançando por passos pequenos, mas seguros, e recorrendo a camadas da abstracção. Mas fica desde já o aviso: não há uma solução única. Na realidade a programação é também uma arte e um compromisso entre coisas por vezes conflituosas como a elegância, a eficiência ou a legibilidade.

### 8.5.2 Primeira Versão

Vamos começar por tentar chegar à solução **sem** pensar na representação. Proceder deste modo pode parecer contra natura, mas é um excelente método de programação. Traduz a ideia de construir o programa com base em camadas de abstracção. Fixemo-nos por agora apenas no facto de a verificação que temos que fazer obrigar a analisar **no máximo** três situações: (1) as somas por linha têm todas o mesmo valor, o número mágico; (2) as somas por colunas têm todas o mesmo valor, o número mágico; (3) as somas das duas diagonais têm todas o mesmo valor, o valor do número mágico. Esta decomposição segue naturalmente o enunciado. Torna-se claro que precisamos de calcular em primeiro lugar o valor do **número mágico**. Daqui resulta um primeiro **esqueleto** de programa:

```

1 def quadrado_magico(quadrado):
2 num_magico = nm(quadrado)
3 # Verifica linhas
4 # Verifica colunas
5 # Verifica diagonais
6 return resposta, num_magico

```

Não avançamos muito, é verdade. Mas olhando para este primeiro esboço temos a confiança de que não fizemos nada de errado. Para que a resposta ao problema seja positiva é preciso que as **três** condições se verifiquem. Mas para que a resposta seja negativa basta que **uma** delas não se verifique. Isso leva-nos a propor a seguinte solução:

```

1 def quadrado_magico(quadrado):
2 num_magico = nm(quadrado)
3 # Verifica linhas
4 if not linhas(quadrado, num_magico):
5 return False, num_magico
6 # Verifica colunas
7 elif not colunas(quadrado, num_magico):
8 return False, num_magico
9 # Verifica diagonais
10 else:

```

```
11 return diagonais(quadrado,num_magico), num_magico
12 #return resposta, num_magico
```

Esta solução leva-nos a alterar ligeiramente a solução anterior, pois desaparece a última instrução uma vez que o resultado é dado dentro do `if`. O que temos de nos convencer é que esta solução funciona, no pressuposto de que as definições auxiliares `nm`, `linhas`, `colunas` e `diagonais` funcionam correctamente. A primeira deve devolver o valor do número mágico, enquanto as restantes três devolvem um valor booleano: `True`, se todas as somas (linhas, colunas ou diagonais) forem iguais ao número mágico, `False`, se essa situação não se verificar. O interessante é que com este modo de proceder até podemos testar o programas globalmente, mesmo sem ter definido em concreto as funções auxiliares. Igualmente podemos testar a lógica do nosso programa sem termos decidido o modo como representamos o quadrado.

```
1 def quadrado_magico(quadrado):
2 num_magico = nm(quadrado)
3 # Verifica linhas
4 if not linhas(quadrado,num_magico):
5 return False, num_magico
6 # Verifica colunas
7 elif not colunas(quadrado, num_magico):
8 return False, num_magico
9 # Verifica diagonais
10 else:
11 return diagonais(quadrado,num_magico), num_magico
12 #return resposta, num_magico
13
14 def nm(quadrado):
15 pass
16
17 def linhas(quadrado,num_magic):
18 pass
19
20 def colunas(quadrado, num_magic):
21 pass
22
23 def diagonais(quadrado,num_magic):
24 pass
25
26
27 if __name__ == '__main__':
```

```
28 quadrado = []
29 print(quadrado_magico(quadrado))
```

Notar como se pode definir uma função que não faz nada, colocando no seu corpo a instrução `pass`. Ao executarmos o programa o resultado será:

```
1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
 2336.1.00)]
3 Type "help", "copyright", "credits" or "license" for more
 information.
4 >>> [evaluate my_quad_magic.py]
5 (False, None)
6 >>>
```

Vamos prosseguir com a nossa solução. Concentremo-nos no caso da definição auxiliar `linhas`. Será que podemos construir esta função também por etapas, usando camadas de abstracção? A resposta é afirmativa. A nossa estratégia será a de verificar se todas as linhas têm uma soma igual à do número mágico, abandonando o programa com uma resposta negativa na primeira ocasião em que essa verificação dê um resultado negativo.

```
1 def linhas(quadrado,num_magic):
2 for linha in lin(quadrado):
3 if soma(linha) != num_magic:
4 return False
5 return True
6
7 def lin(quadrado):
8 return []
```

Notar que a função auxiliar `lin` tem que devolver provisoriamente uma lista vazia (ou um tuplo vazio, ou uma cadeia vazia), porque na nossa solução impomos que o que devolve seja um objecto iterável usado no ciclo `for`. Como é evidente a mesma estratégia pode ser usada para as duas funções auxiliares `colunas` e `diagonais`:

```
1 def linhas(quadrado,num_magic):
2 for linha in lin(quadrado):
3 if soma(linha) != num_magic:
4 return False
5 return True
6
7 def lin(quadrado):
```

```

8 return []
9
10 def colunas(quadrado, num_magic):
11 for coluna in col(quadrado):
12 if soma(coluna) != num_magic:
13 return False
14 return True
15
16 def col(quadrado):
17 return []
18
19 def diagonais(quadrado, num_magic):
20 for diagonal in diag(quadrado):
21 if soma(diagonal) != num_magic:
22 return False
23 return True
24
25 def diag(quadrado):
26 return []

```

A partir desta altura não podemos protelar mais a questão de saber como vamos **representar** o quadrado. Nesta versão vamos optar por fazê-lo através de uma **lista de listas**. Por exemplo:

```

1 quadrado = [[4,9,2],[3,5,7],[8,1,6]]

```

Será o modo de representar um quadrado mágico de  $3 \times 3$ , em que cada elemento da lista representa uma linha.

### 8.5.3 Linhas

Com esta representação podemos logo avançar para a solução da função **lin**:

```

1 def lin(quadrado):
2 return quadrado

```

Trivial! Na realidade, devido à representação usada esta função na prática não faz nada. Os outros casos já obrigam a um pouco de reflexão. Comecemos pelas colunas.

### 8.5.4 Colunas

Se olharmos para o quadrado como se fosse uma matriz (no sentido matemático do termo), então o nosso problema consiste em obter a matriz **trans-**

**posta**, ou seja, trocar as linhas pelas colunas:

```

1 quadrado = [[4,9,2],[3,5,7],[8,1,6]]
2 transposta = [[4,3,8],[9,5,1],[2,7,6]]
```

De um modo geral, temos que trocar o elemento  $mat[i][j]$  com o elemento  $mat[j][i]$ .

```

1 def col(quadrado):
2 mat = []
3 for i in range(len(quadrado)):
4 linha_i = []
5 for j in range(len(quadrado[0])):
6 linha_i.append(quadrado[j][i])
7 mat.append(linha_i)
8
9 return mat
```

Esta versão é simples mas pouco eficiente. Se pensarmos bem a transposta de uma matriz quadrada pode ser obtida fazendo uma troca entre elementos simétricos relativamente à diagonal principal (ver figura 8.11).

|                 |                 |                 |                 |
|-----------------|-----------------|-----------------|-----------------|
| a <sub>11</sub> | a <sub>12</sub> | a <sub>13</sub> | a <sub>14</sub> |
| a <sub>21</sub> | a <sub>22</sub> | a <sub>23</sub> | a <sub>24</sub> |
| a <sub>31</sub> | a <sub>32</sub> | a <sub>33</sub> | a <sub>34</sub> |
| a <sub>41</sub> | a <sub>42</sub> | a <sub>43</sub> | a <sub>44</sub> |

Figura 8.11: Simetrias

Uma solução mais económica é aquela que vai apenas percorrer tantas posições quantas as que existem numa das matrizes triangulares. Vejamos uma tentativa.

```

1 def col_d(quadrado):
2 mat = quadrado[:]
3 for i in range(len(quadrado)):
4 for j in range(i+1, len(quadrado[0])):
5 mat[i][j] = quadrado[j][i]
6 mat[j][i] = quadrado[i][j]
```

```
7 return mat
```

Atente-se no modo como o `j` varia, começando a partir de `i+1`, garantindo assim que é a matriz triangular que é percorrida. Parece tudo bem. Mas quando executamos, o que acontece?

```
1 Python 3.2.3 (default, Sep 5 2012, 20:52:27)
2 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
3 2336.1.00)]
4 Type "help", "copyright", "credits" or "license" for more
5 information.
6 >>> [evaluate my_quad_magic.py]
7 [[4, 9, 2], [3, 5, 7], [8, 1, 6]]
[[4, 3, 8], [3, 5, 1], [8, 1, 6]]
>>>
```

Faz bem o primeiro caso mas os resultados seguintes estão errados. O que se passa? Temos que olhar para o modo como os objectos estão **representados** na memória do computador.

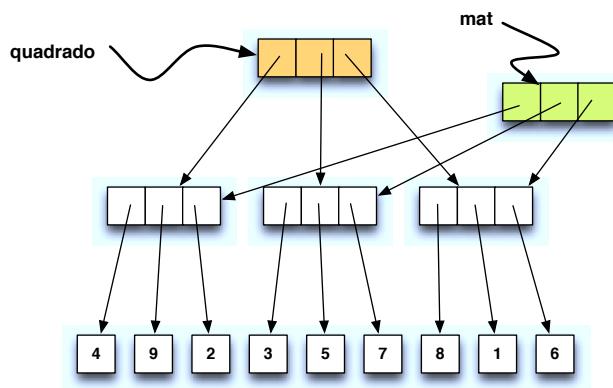


Figura 8.12: Representação de uma lista de listas

Da figura 8.12 resulta claro que se é verdade que a instrução `mat = quadrado[:]` associa `mat` a uma cópia do `quadrado`, essa cópia é apenas ao primeiro nível da lista de listas. Dizemos que fizemos uma **cópia de superfície** (*shallow copy*). Deste modo, qualquer modificação que é feita no **interior** da estrutura através de um dos nomes afecta **globalmente** o objecto. Como consequência, as alterações a `mat` também alteram `quadrado`.

Mas existe solução para o problema. O módulo `copy` tem uma função, `deepcopy`, que permite realizar uma cópia profunda, separando completa-

mente os objectos na memória. Agora apenas partilham os objectos primitivos imutáveis, neste caso os números inteiros. Usando-o podemos implementar correctamente a nossa solução.

```

1 import copy
2
3 def col_d(quadrado):
4 mat = copy.deepcopy(quadrado)
5 for i in range(len(quadrado)):
6 for j in range(i+1, len(quadrado[0])):
7 mat[i][j] = quadrado[j][i]
8 mat[j][i] = quadrado[i][j]
9
return mat

```

A figura 8.13 mostra o resultado de optarmos por `deepcopy`.

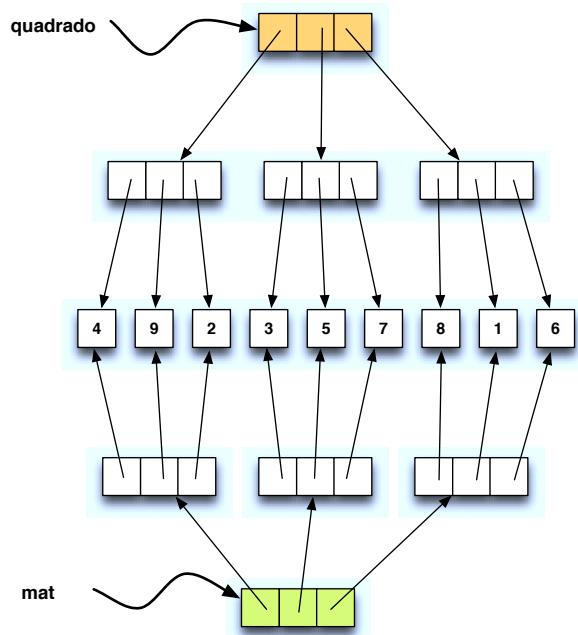


Figura 8.13: Representação de uma lista de listas: recurso a `deepcopy`

### 8.5.5 Diagonais

Ainda nos falta o problema das diagonais. Independentemente da dimensão do quadrado, só existem duas diagonais principais. Se nos reportarmos à

figura 8.11 verificamos facilmente a relação dos **índices** linha/coluna nas diagonais: ou são iguais ou a sua soma é igual à dimensão do quadrado menos um. Daqui resulta o programa:

```

1 def diag(quadrado):
2 diag_1 = []
3 diag_2 = []
4 for i in range(len(quadrado)):
5 for j in range(len(quadrado[0])):
6 if i == j:
7 diag_1.append(quadrado[i][j])
8 if (i+j) == (len(quadrado) - 1):
9 diag_2.append(quadrado[i][j])
10 return [diag_1, diag_2]

```

Veja-se com atenção a solução, em particular o recurso a dois **if**: existe um valor que pertence às duas diagonais.

### 8.5.6 Número Mágico

A obtenção do número mágico<sup>6</sup> é mais simples pois obedece à fórmula:

$$nm(n) = \sum_{i=0}^{n^2} i = \frac{1}{2} \times n \times (n^2 + 1) = \frac{1}{2} \times (n^3 + n)$$

Da fórmula resulta o programa:

```

1 def nm(quadrado):
2 linhas = len(quadrado)
3 colunas = len(quadrado[0])
4 return (linhas ** 3 + colunas) / 2

```

### 8.5.7 Juntando as peças

Ainda nos falta para concluir definir a função **soma**. Trata-se de um problema trivial, apresentando-se duas soluções possíveis<sup>7</sup>.

```

1 def soma(lista):
2 return sum(lista)
3
4 def soma_b(lista):

```

---

<sup>6</sup>Para um quadrado mágico dito normal.

<sup>7</sup>Ainda existem mais alternativas, mas fica para o leitor a sua descoberta.

```

5 total = 0
6 for elem in lista:
7 total = total + elem
8 return total

```

Posto isto, o programa final:

```

1 """Quadrado Mágico.
2 Programação descendente.
3 """
4
5 __author__ = 'Ernesto Costa'
6 __date__ = 'April 2013'
7
8 def quadrado_magico(quadrado):
9 num_magico = nm(quadrado)
10 # Verifica linhas
11 if not linhas(quadrado, num_magico):
12 return False, num_magico
13 # Verifica colunas
14 elif not colunas(quadrado, num_magico):
15 return False, num_magico
16 # Verifica diagonais
17 else:
18 return diagonais(quadrado, num_magico), num_magico
19
20 # -- Número Mágico
21 def nm(quadrado):
22 linhas = len(quadrado)
23 colunas = len(quadrado[0])
24 return (linhas ** 3 + colunas) / 2
25
26 # -- Linhas
27 def linhas(quadrado, num_magico):
28 for linha in lin(quadrado):
29 if soma(linha) != num_magico:
30 return False
31 return True
32
33 def lin(quadrado):
34 return quadrado
35

```

```
36 # -- Colunas
37 def colunas(quadrado, num_magic):
38 for coluna in col(quadrado):
39 if soma(coluna) != num_magic:
40 return False
41 return True
42
43 def col(quadrado):
44 mat = []
45 for i in range(len(quadrado)):
46 linha_i = []
47 for j in range(len(quadrado[0])):
48 linha_i.append(quadrado[j][i])
49 mat.append(linha_i)
50 return mat
51
52 # -- Diagonais
53 def diagonais(quadrado, num_magic):
54 for diagonal in diag(quadrado):
55 if soma(diagonal) != num_magic:
56 return False
57 return True
58
59 def diag(quadrado):
60 diag_1 = []
61 diag_2 = []
62 for i in range(len(quadrado)):
63 for j in range(len(quadrado[0])):
64 if i == j:
65 diag_1.append(quadrado[i][j])
66 if (i+j) == (len(quadrado) - 1):
67 diag_2.append(quadrado[i][j])
68 return [diag_1, diag_2]
69
70
71 def soma(lista):
72 return sum(lista)
73
74 # -- Para testar
75 if __name__ == '__main__':
76 quadrado_f = [[4,1,2],[3,5,7],[8,9,6]]
```

```

77 quadrado_3 = [[4,9,2],[3,5,7],[8,1,6]]
78 quadrado_4 =
 [[16,3,2,13],[5,10,11,8],[9,6,7,12],[4,15,14,1]]
79 quadrado_5 = [[17,24,1,8,15],[23,5,7,14,16],[4,6,13,20,22],
80 [10,12,19,21,3],[11,18,25,2,9]]
81
82 print quadrado_magico(quadrado_5)

```

## Sumário

Este texto é um discurso sobre programação. Recorrendo a exemplos conhecido simples mostrámos como se pode dominar a complexidade. Durante o desenvolvimento do programa falamos também de alternativas possíveis, que obrigam a conhecer um pouco mais dos conceitos da linguagem.

## Teste os seus conhecimentos

Este capítulo debruçou-se sobre o modo como se pode dominar a complexidade inerente à construção de programas por recurso a uma abordagem simples de decomposição de um problema em sub-problemas. Não existem elementos novos do ponto de vista conceptual pelo que deve procurar saber se entendeu ou não o processo.

- Que diferentes tipos de conhecimento são postos em jogo quando se está a desenvolver um programa.
- Que objectivos, por vezes contraditórios, se procura alcançar quando se constrói um programa.
- Como se tem confiança de que o programa final está correcto. Como se testa.
- Como se pode proteger um programa.

## Exercícios

**Exercício 8.1 F** Desenvolva um programa que lhe permita imprimir uma tabuada para os números até um dado  $n$ .

**Exercício 8.2 M** Suponha que tem que manter actualizada uma classificação de uma corrida de ciclismo do tipo contra-relógio individual. Ad-

mita, para facilitar, que cada concorrente é caracterizado pelo nome, equipa e tempo gasto. Desenvolva o respectivo programa.

**Exercício 8.3 D** Todos conhecemos o jogo do **Mastermind**. Um jogador A dispõe de um conjunto colorido de piões. Um segundo jogador, B, dispõe igualmente de um conjunto de piões coloridos dos quais escolhe um subconjunto que dispõe de forma ordenada sem conhecimento do primeiro jogador. O objectivo é o jogador A descobrir a sequência escolhida pelo jogador B, no menor número de tentativas. Por cada tentativa o jogador A recebe uma indicação do jogador B acerca da qualidade da sua tentativa: quantas cores e posições acertou e quantas cores acertou sem acertar na posição. Queremos implementar um **simulador** para este jogo. O computador faz o papel do jogador B, enquanto que um humano assumirá o papel de jogador A. O número de cores diferentes, o comprimento da sequência escondida e o número de jogadas máximas possível são parâmetros do programa. A figura 8.14 ilustra um caso simples após duas jogadas.

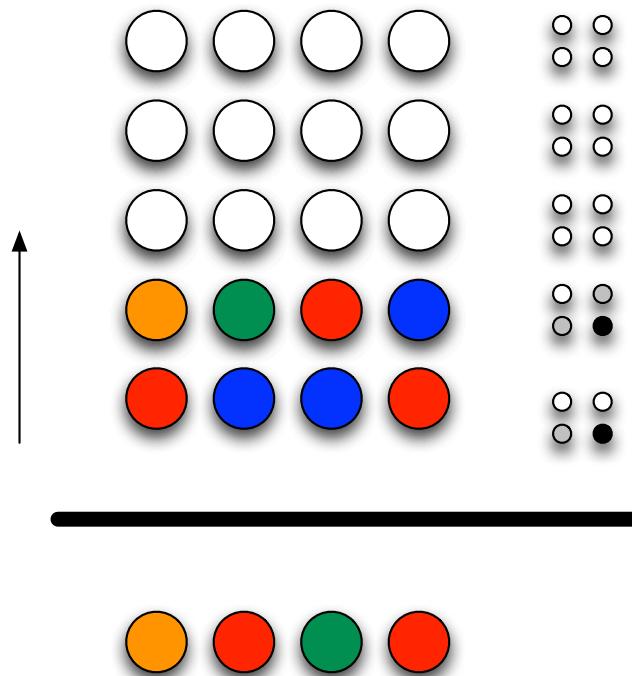


Figura 8.14: Mastermind

**Exercício 8.4**

**Exercício 8.5**

**Exercício 8.6**

**Exercício 8.7**

**Exercício 8.8**

**Exercício 8.9**

**Exercício 8.10**

# Capítulo 9

## Visões (II)

### Objectivos

- ✓ Exercitar o uso de estruturas de dados e mecanismos de controlo para objectos bidimensionais
- ✓ Introduzir conceitos básicos sobre processamento de imagens digitais
- ✓ Introduzir o módulo cImage
- ✓ Exercitar ao uso de funções como parâmetros

### 9.1 Introdução

Vivemos num mundo cada vez mais dominado pelas imagens. A manipulação das imagens permite-nos criar uma nova realidade e explorar novas combinações de formas e de cores. Com os computadores essa possibilidade de alteração e de jogo foi potenciada a uma escala nunca antes vista. Hoje, existem imensos programas que nos permitem exercitar a nossa imaginação artística, de que o **Photoshop** da Adobe é apenas um exemplo. Neste capítulo iremos ver como podemos nós próprios usar e transformar imagens, por recurso à linguagem **Python**. Mas começemos por clarificar o que são imagens e como podem ser guardadas num computador. De um modo informal uma imagem é uma estrutura bidimensional, uma matriz de pontos, cada um deles designado por **pixel**<sup>1</sup>. Uma representação comum das imagens em computador consiste em representar cada pixel separadamente, falando-se

<sup>1</sup>Acrónimo derivado do inglês *picture element*.

então em imagens *bitmap*<sup>2</sup>. Cada pixel, por sua vez, é codificado de acordo com um dado **modelo**. No caso do modelo **RGB**, a cor é decomposta em três componentes, ou **canais**<sup>3</sup>, uma para Vermelho (*Red*), outro para Verde (*Green*), e outro para azul (*Blue*), à semelhança do modo como nós humanos percebemos a cor. Cada canal é representado por um byte, o que significa que um pixel ocupa 3 bytes (24 bits) e podemos representar em teoria  $256^3 = 16777216$  cores diferentes, visto cada canal poder assumir  $2^8 = 256$  valores diferentes<sup>4</sup>.

A figura 9.1 mostra algumas cores e respectivas codificações RGB.

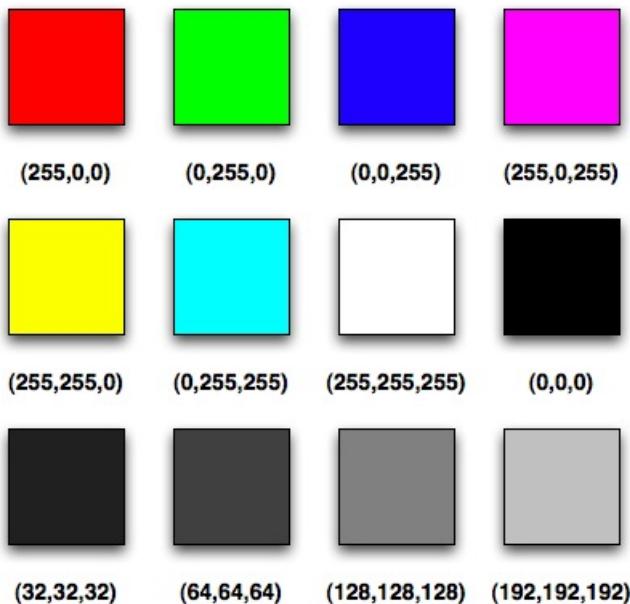


Figura 9.1: Mapeamento RGB cores

Na última linha da figura apresentamos diversos níveis de cinzento, que se obtém usando o mesmo valor para os três canais. No caso extremo temos a cor branca (255 em cada canal) e a cor preta (0 em cada canal).

---

<sup>2</sup>Uma alternativa designa-se por imagens **vectoriais**. De um modo grosseiro, isso significa que em vez da imagem o que é guardado e manipulado é o programa que produz a imagem.

<sup>3</sup>É possível a existência de mais canais, como seja o canal **alfa** para a transparência.

<sup>4</sup>Outros modelos de cor são o HSB e o CMYK, este último usado pelas impressoras.

As imagens ocupam muito espaço. A tabela 9.1 mostra alguns valores, isto é, o espaço ocupado para diferentes **resoluções** das imagens.

Tabela 9.1: Imagens e tamanhos

|             | $320 \times 240$ | $640 \times 480$ | $1024 \times 768$ |
|-------------|------------------|------------------|-------------------|
| 24-bits cor | 230400 bytes     | 921600 bytes     | 2359296 bytes     |

É de se esperar que com naturalidade que se procuraram métodos de compressão de imagem. Existem vários métodos de compressão, como é o caso da norma JPEG<sup>5</sup>, que especifica o modo como uma imagem é transformada numa sequência de bytes e o processo inverso. Tipicamente perde-se alguma informação durante a compressão/descompressão, mas nalgumas situações essa perda de qualidade é aceitável<sup>6</sup>. Existem algoritmos que tornam possível comprimir a imagem sem perda de informação, como acontece com a técnica conhecida por RLE<sup>7</sup>. Existem vários outros formatos de imagem, nomeadamente os que não provocam perdas como sejam o TIFF, GIF ou PNG.

## 9.2 Representação de imagens

A manipulação de imagens num computador envolve em geral a sua transferência de um ficheiro em disco para a memória interna. É preciso por isso encontrar uma boa representação, isto é um contentor. Do que já conhecemos de Python e da natureza das imagens *bitmap*, não nos surpreende que uma representação escolhida simples seja uma lista de listas, uma matriz. Cada lista interior representa uma **linha** da matriz. Por exemplo:

```
>>> matriz = [[1,2,3],[4,5,6],[7,8,9]]
```

No caso das imagens os elementos primitivos não serão números mas (a representação de) pixéis. No modelo RGB um modo simples de representar um pixel é através de um tuplo com o valor das três componentes, ou canais. Estes valores podem variar entre 0 e 255. Um exemplo de uma pequena imagem de 3 por 2 pixéis podia ser:

---

<sup>5</sup>Acrónimo derivado do inglês *Joint Photographic Experts Group*

<sup>6</sup>Em certas aplicações, e.g., na área médica, essa perda de qualidade não pode ser tolerada.

<sup>7</sup>Do inglês *run lenght encoding*.

```
imagem = [[(131, 27, 223), (4, 243, 68), (195, 107, 123)], [(246, 205, 141),
(154, 60, 154), (40, 31, 223)]]
```

Resolvida a questão da representação, passemos a alguns exemplos simples de manipulação<sup>8</sup>. Para começar, vamos ver como podemos visualizar os elementos de uma matriz quadrada qualquer. A questão essencial que temos que ter em conta é que, havendo duas dimensões, temos que criar **dois** ciclos, cada um deles responsável por percorrer de modo ordenado cada uma das **dimensões**. Vejamos o caso mais simples, ou seja, mostrar todos os elementos, por linhas.

```
1 def mostra_por_linhas(matriz):
2 """Indexação pelo conteúdo."""
3 for linha in matriz:
4 for coluna in linha:
5 print("%5d" % coluna, end=' ')
6 print()
7
8 def mostra_por_linhas_b(matriz):
9 """Indexação pela posição."""
10 for pos_linha in range(len(matriz)):
11 for pos_coluna in range(len(matriz[0])):
12 print("%5d" % matriz[pos_linha][pos_coluna], end=' ')
13 print()
```

O que têm estes dois programas de diferente? O simples facto de, no primeiro, percorremos a matriz usando o seu conteúdo, enquanto que, no segundo, usamos as posições. E de comum, que pontos devem ser salientados? Essencialmente o modo como fazemos a impressão: o comando **print** usa uma marca de formatação (**%5d**) e termina com **end =''**. É este último facto que permite colocar os elementos de uma linha todos ao lado uns dos outros. O segundo **print** sem argumento serve apenas para mudar de linha.

Admitamos agora que queremos mostrar a matriz por colunas e não por linhas.

```
1 def mostra_por_colunas(matriz):
2 """ Indexação pelo posição."""
3 for pos_coluna in range(len(matriz[0])):
4 for pos_linha in range(len(matriz)):
5 print("%3d" % matriz[pos_linha][pos_coluna], end='')
```

---

<sup>8</sup>Os exemplos que se seguem envolvem listas de listas de números. Adiante usaremos o que aprendemos para o tratamento de imagens.

### 6 print()

Bastou **trocar** a ordem dos ciclos: o primeiro, trata das colunas, enquanto o segundo, mais interior, trata das linhas! E se forem as matrizes triangulares superior?

```

1 def mostra_tri_sup(matriz):
2 """Matriz triangular superior. Indexação pela posição."""
3 for pos_linha in range(len(matriz)):
4 print(' ' * 4 * pos_linha, end='')
5 for pos_coluna in range(pos_linha, len(matriz[0])):
6 print("%4d" % matriz[pos_linha][pos_coluna], end=' ')
7 print()

```

Atente-se como tratamos de mostrar de modo conveniente usando uma expressão de formatação apropriada. Uma vez mais o comando **print** é essencial para esse objectivo.

Deixamos ao leitor a tarefa de testar estes programas.

Passemos ao problema não de visualizar mas de **criar** uma matriz conhecida a sua dimensão. Uma solução banal será:

```

1 def cria_mat(n,m,val):
2 """Cria uma matrix nXm sendo que todos os elementos são
3 iguais a val."""
4 mat = []
5 for j in range(n):
6 linha = []
7 for i in range(m):
8 linha.append(val)
9 mat.append(linha)
10 return mat

```

Mas com o que já sabemos de Python podemos fazer melhor.

```

1 def cria_mat_b(n,m,val):
2 """Cria uma matrix nXm sendo que todos os elementos são
3 iguais a val."""
4 mat = [[val for i in range(m)] for j in range(n)]
5 return mat
6
7 def cria_mat_c(n,m,val):
8 """Cria uma matrix nXm sendo que todos os elementos são
9 iguais a val."""
10 return[[val] * m] * n

```

A primeira alternativa baseia-se em listas por comprehensão, enquanto que a última no operador de repetição para listas. O recurso a dois ciclos imbricados explícitos é um **padrão** que pode ser usado em muitas situações.

São várias as operações que podemos fazer com matrizes e que nos obrigam a percorrer os seus elementos de acordo com uma determinada ordem. O exemplo mais simples é talvez o da soma de duas matrizes. O programa que se segue pressupõe que as matrizes têm a mesma dimensão.

```

1 def soma_matriz(mat_1,mat_2):
2 """ Soma duas matrizes da mesma dimensão."""
3 n_linhas = len(mat_1)
4 n_colunas = len(mat_1[0])
5 mat = cria_mat(n_linhas,n_colunas,0)
6 # Soma
7 for i in range(n_linhas):
8 for j in range(n_colunas):
9 mat[i][j]= mat_1[i][j]+ mat_2[i][j]
10 return mat

```

Do mesmo modo podemos efectuar a multiplicação de duas matrizes  $C = A \times B$ , sabendo que:

$$c_{ij} = \sum_{k=1}^n a_{ik} * b_{kj}$$

```

1 def mult_matriz(mat_1,mat_2):
2 """Multiplica duas matrizes iXk e kXj."""
3 n_linhas_1 = len(mat_1)
4 n_colunas_1 = len(mat_1[0]) # igual a n_linhas_2
5 n_colunas_2 = len(mat_2[0])
6 mat_prod = cria_mat(n_linhas_1,n_colunas_2,0)
7 # Multiplica
8 for i in range(n_linhas_1):
9 for j in range(n_colunas_2):
10 val=0
11 for k in range(n_colunas_1):
12 val = val + mat_1[i][k]* mat_2[k][j]
13 mat_prod[i][j]=val
14 return mat_prod

```

Em qualquer destes casos criámos uma matriz onde depois guardamos o resultado da operação. Mas podemos ter operações em que isso não é um requisito e portanto o resultado da operação resulta na alteração de uma das matrizes. Suponhamos que queremos modificar uma matriz alterando o seu conteúdo, por exemplo somando uma dada constante a todos os elementos nas posições ímpares.

```

1 def prod_const_mat(mat, val):
2 """Multiplica os elementos nas colunas ímpares por val."""
3 n_linhas = len(mat)
4 n_colunas = len(mat[0])
5 for linha in range(n_linhas):
6 for col in range(1, n_colunas, 2):
7 mat[linha][col] *= val
8 return mat

```

Todos estes exemplos mostram a importância fundamental dos dois ciclos **for** imbricados para percorrer a estrutura bi-dimensional, no interior dos quais se encontra a função de manipulação. Em alguns casos precisamos de criar uma matriz nova, com a dimensão necessária para guardar o resultado da manipulação. Todos estes elementos estarão presentes nos programas de manipulação de imagens que são estruturas a duas dimensões.

## 9.3 O módulo cImage

Regressemos às imagens e ao problema da sua construção, consulta e manipulação. Vamos utilizar o módulo **cImage**<sup>9</sup> que disponibiliza uma interface de alto nível para tratamento de imagens. Este módulo socorre-se de outros dois: o módulo nativo **Tkinter** e o módulo **PIL**<sup>10</sup>. Enquanto o **Tkinter** nos permite definir uma Interface de Utilizador Gráfica<sup>11</sup>, com o **PIL** podemos manipular imagens de diferentes formatos, como seja jpeg, eps, gif, png, ou tiff, só para mencionar alguns. Para poder funcionar o **PIL** socorre-se de um processo de codificação/descodificação mais ou menos complexo, e cuja descrição sai fora do âmbito deste livro.. No momento em que escrevemos, o módulo **PIL** derivou em **Pillow**<sup>12</sup> o que permite ser compatível com a versão 3 da linguagem. Estes módulos dependem ainda de um conjunto importante

---

<sup>9</sup>TBD:...colocar informação sobre o modo de obter...

<sup>10</sup>Acrônimo de *Python Imaging Library*. Pode ser obtido em <http://www.pythonware.com>.

<sup>11</sup>Em inglês *Grapgical User Interface* (GUI).

<sup>12</sup>Mais informações sobre o módulo em <https://pypi.python.org/pypi/Pillow/2.0.0>.

de bibliotecas para os diversos *codecs* necessários e que devem estar (ou ser) instalados no computador.

O módulo permite criar, consultar e manipular três grandes tipos de objectos: janelas, imagens e pixéis. As janelas funcionam como contentores para as imagens, enquanto estas são formadas por pixéis. Existem três (sub-)tipos de imagens: imagens de ficheiro, imagens vazias, imagens de listas de dados (ver figura 9.2). As primeiras são imagens pré-existentes em disco num dos formatos permitidos, as segundas são imagens por nós criadas, pixel a pixel, e as terceiras são imagens dadas no formato lista de listas de pixéis, semelhante ao acima referido para matrizes. Nestes dois últimos casos o formato em que as imagens são guardadas externamente depende do sufixo usado no nome, sendo que na ausência de sufixo o formato por defeito é o jpeg.

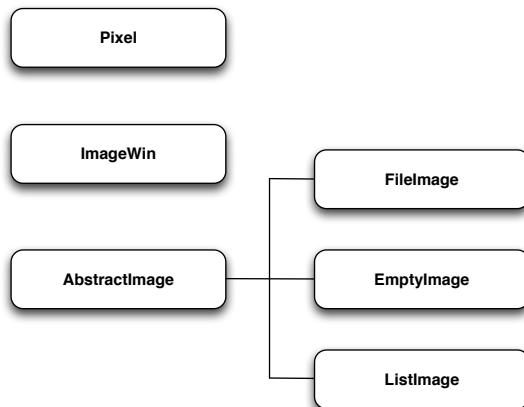


Figura 9.2: Os tipos de cImage

## Janelas

O construtor do tipo janela chama-se `ImageWin()`. Tem por argumentos o nome da janela, a sua largura e a sua altura. Existem vários outros métodos como sejam o que permite obter a largura da janela (`getWidth()`), a altura da janela (`getHeight()`), as coordenadas do rato (`getMouse()`), ou definir a cor de fundo (`setBackground()`).

### Imagens

Para cada tipo de imagem existe um construtor. Assim temos três construtores específicos: `FileImage()`, `EmptyImage()` e `ListImage()`. As imagens podem ser desenhadas numa janela (`draw()`), salvas (`save()`), podemos obter a sua largura (`getWidth()`) e/ou a sua altura (`getHeight()`), modificar um determinado pixel (`setPixel()`), entre outras operações.

### Pixeis

O construtor designa-se por `Pixel()`. Podemos consultar cada um dos três canais (`getRed()`,`getGreen()`,`getBlue()`), ou modificar cada uma das componentes (`setRed()`,`setGreen()`,`setBlue()`).

## 9.4 Exemplos Básicos

O primeiro exemplo envolve simplesmente a criação de uma janela. Apenas [Janelas](#) temos que indicar o seu nome e a sua resolução (largura e altura em pixeis).

```

1 import cImage
2
3 def cria_janela(nome, largura, altura):
4 janela= cImage.ImageWin(nome, largura, altura)
5 janela.exitOnClick()
6
7 if __name__ == '__main__':
8 cria_janela('Janela Indiscreta', 320,240)
```

A execução do código produz a imagem da figura 9.3.O método `exitOnclck()` actua sobre objectos do tipo `ImageWin()` e permite encerrar a janela e abandonar a execução. Tipicamente é a ultima acção a realizar pelos programas.

Como se verifica a janela tem um fundo branco. Mas podemos criar uma janela em que a cor de fundo é, por exemplo, vermelha. Vejamos agora alguns aspectos básicos envolvendo janelas e o desenho de formas simples. O primeiro exemplo mostra a criação de uma janela em que a cor de fundo é vermelha<sup>13</sup>.

```

1 def cria_janela_cor(nome, largura, altura, cor):
2 janela= cImage.ImageWin(nome, largura, altura)
```

---

<sup>13</sup>A cor também pode ser definida em hexadecimal ou através de um tuplo (r,g,b). Por exemplo, a cor vermelha podia ter sido definida pela cadeia de caracteres '#ff0000' ou por (255,0,0).



Figura 9.3: Um janela simples

```
3 janela.setBackground(cor)
4 janela.exitOnClick()
5
6 if __name__ == '__main__':
7 cria_janela_cor('Teste de Cor de Fundo', 320,240,'red')
```



Figura 9.4: Fundo vermelho

## Imagens

As janelas existem, como referimos, como contentores para objectos que são imagens. Estas imagens ou são criadas por nós ou existem guardadas externamente em disco. O caso mais simples é o de uma imagem por nós criada ... sem nada.

```
1 import cImage
2
```

```
3 def cria_imagem_vazia(largura,altura):
4 imagem = cImage.EmptyImage(largura,altura)
5 return imagem
6
7 def mostra_imagem_simples(imagem):
8 # Dimensão
9 largura = imagem.getWidth()
10 altura = imagem.getHeight()
11 # Cria janela
12 janela = cImage.ImageWin('Imagen', 2*largura,2*altura)
13 # Mostra imagem na janela
14 imagem.draw(janela)
15 # Termina
16 janela.exitOnClick()
```

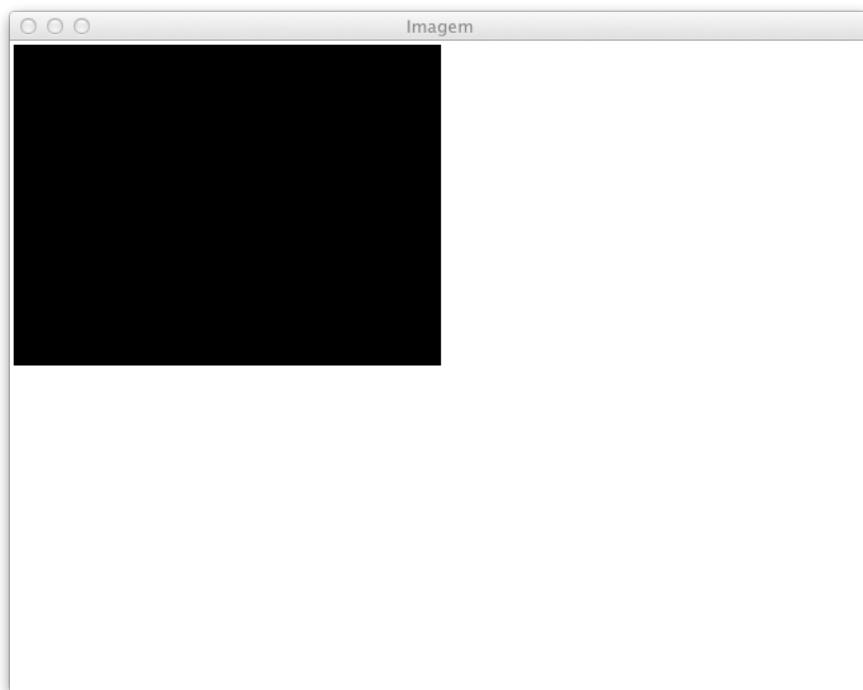


Figura 9.5: Uma imagem em branco ... é preta!

Alguns aspectos a referir. Em primeiro lugar, criámos uma janela que tem o dobro do tamanho em cada dimensão do que o da imagem criada (linha 12) e o posicionamento da imagem é feito por defeito a partir do canto

superior esquerdo, que corresponde às coordenadas (0, 0). Em segundo lugar, a imagem criada é preta. Estes dois aspectos podem ser alterados, ou seja podemos posicionar a imagem noutro local da janela e podemos alterar a cor da imagem. Sem mexer no código de modo substantivo podemos alterar a sua cor (linha 5) e posicionar a imagem (linha 6) e como se pode ver na figura 9.6.

```
1 if __name__ == '__main__':
2 largura = 320
3 altura = 240
4 imagem = cImage.EmptyImage(largura,altura)
5 imagem.setSolidColor((0,0,255))
6 imagem.setPosition(largura//2,altura//2)
7 mostra_imagem_simples(imagem)
```

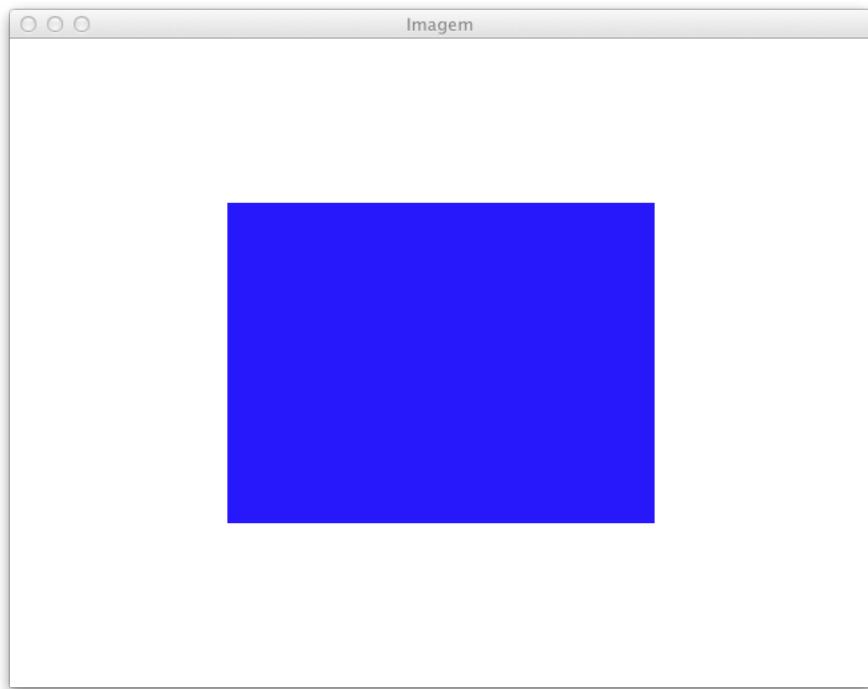
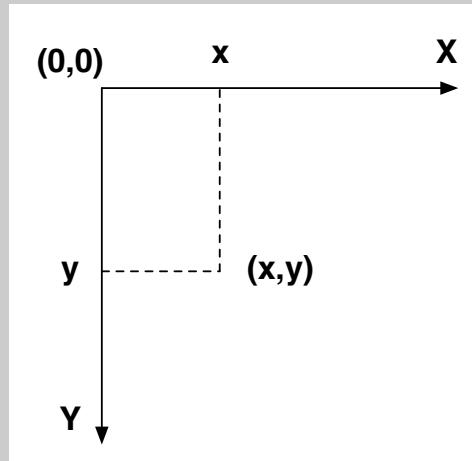


Figura 9.6: Lidar com a cor e a posição



## Coordenadas

Cada pixel tem associado as suas coordenadas. As coordenadas  $(x, y) = (0, 0)$  situam-se no canto superior esquerdo da imagem. Os valores de  $x$  crescem horizontalmente para a direita, enquanto os valores de  $y$  crescem verticalmente para baixo. A figura ilustra a situação.



Falta apenas referir como podemos alterar os pixels individualmente. O [Pixel](#)s caso que apresentamos refere-se a uma situação simples em que traçamos uma linha horizontal vermelha, sobre fundo preto.

```

1 def desenha_linha(imagem):
2 altura = imagem.getHeight()
3 largura = imagem.getWidth()
4 janela = cImage.ImageWin('Imagen', largura, altura)
5 pix = cImage.Pixel(255,0,0)
6 for col in range(largura):
7 imagem.setPixel(col, altura//2,pix)
8 imagem.draw(janela)
9 janela.exitOnClick()
```

Dados estes exemplos sabemos agora que as questões centrais a resolver no tratamento de imagens são essencialmente três: definir uma janela, definir uma imagem (construindo-a, ou carregando-a do disco seguido de eventual manipulação), e mostrar a imagem. Apresentamos de seguida um exemplo simples, sem manipulação dos pixels, que envolve estes três aspectos.

```

1 import cImage
2
3 def mostra_imagem(img_fich):
```

```
4 # Carrega a imagem do disco
5 imagem = cImage.FileImage(img_fich)
6 # Obtem Componentes
7 largura = imagem.getWidth()
8 altura = imagem.getHeight()
9 # Cria janela
10 janela = cImage.ImageWin('Imagen', largura,altura)
11 # Mostra imagem na janela
12 imagem.draw(janela)
13 # Termina
14 janela.exitOnClick()
15
16 if __name__ == '__main__':
17 mostra_imagem('/images/calvin_leia.jpg')
```

Este exemplo pretende ser um modelo para o programa principal dos exemplos que apresentamos de seguida. O código, com os seus comentários, é auto-explicativo. Ao executar obtemos a imagem da figura 9.7.

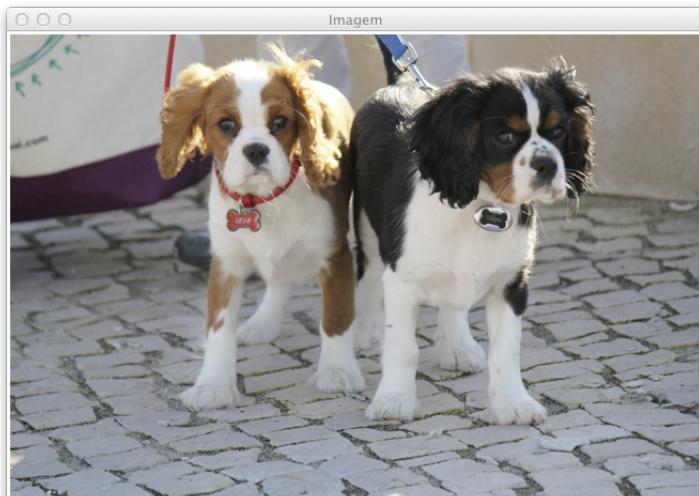


Figura 9.7: Reproduzir uma imagem

## 9.5 Manipulações simples

### Negativo

O primeiro exemplo simples em que uma imagem é manipulada consistirá na obtenção do seu negativo. Sabemos, por exemplo, que o preto se transforma em branco, o verde em magenta, o vermelho em ciano, o azul em amarelo. A forma de conseguir esta transformação consiste em substituir o valor da intensidade em cada canal pela sua diferença para o valor máximo 255. Deste modo a nossa solução é trivial: depois de carregar a imagem, vamos percorrer-la pixel a pixel transformando cada um no seu equivalente negativo.

```
1 import cImage
2
3 # Negativo de imagem
4 def main_negativo(imagem_ficheiro):
5 """Constrói e vizualiza o negativo de uma imagem."""
6 # Obtém imagem
7 imagem = cImage.FileImage(imagem_ficheiro)
8 # Fabrica o negativo
9 imagem_nova = negativo_imagem(imagem)
10 # Define janela
11 largura = imagem.getWidth()
12 altura = imagem.getHeight()
13 janela = cImage.ImageWin('Negativo', 2*largura, altura)
14 # vizualiza
15 imagem.draw(janela)
16 imagem_nova.setPosition(largura+1, 0)
17 imagem_nova.draw(janela)
18 # Termina
19 janela.exitOnClick()
20
21 def negativo_imagem(imagem):
22 """ Negativo de uma imagem."""
23 largura = imagem.getWidth()
24 altura = imagem.getHeight()
25 imagem_nova = cImage.EmptyImage(largura, altura)
26 # percorre pixel a pixel
27 for coluna in range(largura):
28 for linha in range(altura):
29 # transforma
30 pixel_original = imagem.getPixel(coluna, linha)
```

```

31 novo_pixel = negativo_pixel(pixel_original)
32 imagem_nova.setPixel(coluna,linha,novo_pixel)
33 return imagem_nova

```

Vale a pena perder algum tempo com este código pois ele ilustra um padrão para percorrer uma imagem semelhante ao apresentado para o percurso de matrizes<sup>14</sup>. Começamos por definir o nosso programa principal (**main\_negativo**) que decompõe numa sequência de cinco passos a concretização da solução para o nosso problema: obtenção da imagem, fabrico do negativo, definição da janela onde vão ser colocadas as duas imagens, a visualização das imagens e o abandono do programa. O negativo é fabricado sem destruição do original, sendo criada uma imagem vazia que depois vai receber os pixels modificados. O uso do método **setPosition()** permite-nos colocar as duas imagens neste caso o ao lado uma da outra (ver figura 9.8). Como se pode ver pelo código da função **negativo\_imagem**, abstraímos numa definição (i.e., **negativo\_pixel**) a transformação de um pixel no seu negativo. Isto permite melhor legibilidade e pôr em evidência um padrão geral de tratamento de **todos** os pixels de uma imagem pela mesma função de transformação: dois ciclos imbricados que vão gerando de modo ordenado as coordenadas dos pixels.



Figura 9.8: Negativo de uma imagem

---

<sup>14</sup>Devemos ter em atenção no entanto que quando accedemos a uma matriz, na forma de listas de listas, primeiro indicamos a linha e depois a coluna. No caso das imagens, indicamos primeiro a posição ao longo do eixo dos x (largura) e depois ao longo do eixo dos y (altura).

### Cinzentos

Esta ideia de abstracção pode ser ilustrada se agora pretendermos transformar a imagem de colorida para escala de cinzentos. Basta substituir no programa principal a função de transformação por uma que forma o valor na escala de cinzentos. A forma mais simples de o fazer é usar como valor de todos os canais a média dos valores na imagem original:

$$cinza = \frac{red + green + blue}{3}$$

```

1 import cImage
2
3 def main_cinzeno(imagem_ficheiro):
4 """Constrói e vizualiza a escala de cinzentos de uma
5 imagem."""
6 # Obtém imagem
7 imagem = cImage.FileImage(imagem_ficheiro)
8 # Fabrica a escala de cinzentos
9 imagem_nova = cinzeno_imagem(imagem)
10 # Define janela
11 largura = imagem.getWidth()
12 altura = imagem.getHeight()
13 janela = cImage.ImageWin('Cinzeno', 2*largura, altura)
14 # vizualiza
15 imagem.draw(janela)
16 imagem_nova.setPosition(largura+1, 0)
17 imagem_nova.draw(janela)
18 # Termina
19 janela.exitOnClick()
20
21 def cinzeno_imagem(imagem):
22 """ Escala de cinzentos de uma imagem."""
23 largura = imagem.getWidth()
24 altura = imagem.getHeight()
25 imagem_nova = cImage.EmptyImage(largura, altura)
26 # percorre pixel a pixel
27 for coluna in range(largura):
28 for linha in range(altura):
29 # transforma
30 pixel_original = imagem.getPixel(coluna, linha)
31 novo_pixel = cinzeno_pixel(pixel_original)
```

```

31 imagem_nova.setPixel(coluna,linha,novo_pixel)
32 return imagem_nova
33
34 def cinzento_pixel(pixel):
35 """ Converte um pixel para escala de cinzentos."""
36 vermelho = pixel.getRed()
37 verde = pixel.getGreen()
38 azul = pixel.getBlue()
39 int_media = (vermelho + verde + azul) // 3
40 novo_pixel = cImage.Pixel(int_media,int_media, int_media)
41 return novo_pixel

```

Executando o código para a imagem 9.7 obtemos o resultado da figura 9.9.



Figura 9.9: Escala de cinzentos

Esta solução simplifica um pouco o problema da escala de cinzentos, pois a visão humana tem uma percepção da luminância diferente e dependente do canal considerado (vermelho, verde e azul). O código que se segue calcula a média pesada dos três canais.

```

1 def cinzento_pixel(pixel):
2 """ Converte um pixel para escala de cinzentos tendo em
3 atenção a diferença dos canais."""
4 vermelho = pixel.getRed()
5 verde = pixel.getGreen()
6 azul = pixel.getBlue()
7
7 int_media = int(0.299*vermelho + 0.587*verde + 0.114*azul)
8 // 3

```

```

8 novo_pixel = cImage.Pixel(int_media,int_media, int_media)
9 return novo_pixel

```

Usando esta versão obtemos uma imagem em escala de cinzento como se pode ver na figura 9.10.



Figura 9.10: Mais cinzento

Mais uma vez, o código apresentado põe em relevo um padrão de transformação de toda uma imagem pixel a pixel: percorremos as colunas e, para uma dada coluna todas as linhas; identificado o pixel transformamo-lo.

### Sepia

Todos conhecem aquele tom amarelado típico das fotografias antigas. Esse efeito pode ser obtido mediante o recurso a fórmulas de transformação do valor de cada canal:

$$\begin{aligned}
 r_n &= r \times 0.393 + g \times 0.769 + b \times 0.189 \\
 g_n &= r \times 0.349 + g \times 0.686 + b \times 0.168 \\
 b_n &= r \times 0.272 + g \times 0.534 + b \times 0.131
 \end{aligned}$$

Daqui resulta o programa seguinte (onde se tem que ter o cuidado de manter os novos valores dentro do intervalo (0,255)).

```

1 def sepia_pixel(pixel):
2 """ Tempo do passado. """
3 r = pixel.getRed()
4 g = pixel.getGreen()
5 b = pixel.getBlue()

```

```

6 novo_r = int((r * 0.393 + g * 0.769 + b * 0.189))
7 novo_g = int((r * 0.349 + g * 0.686 + b * 0.168))
8 novo_b = int((r * 0.272 + g * 0.534 + b * 0.131))
9 if novo_r > 255: novo_r = r
10 if novo_g > 255: novo_g = g
11 if novo_b > 255: novo_b = b
12 novo_pixel = cImage.Pixel(novo_r,novo_g,novo_b)
13 return novo_pixel

```

Para obter a transformação de uma imagem para sepia só precisamos de usar o modelo já conhecido. O resultado é mostrado na figura 9.11.



Figura 9.11: Sepia

## 9.6 Intermezzo: abstracção

Os exemplos anteriores envolvem a alteração de cada pixel de uma imagem de acordo com um determinado efeito pretendido: passar a negativo, passar a escala de cinzentos, passar a sepia. Para resolver esta questão implementámos a função de transformação do pixel e criámos uma função geral para cada caso. Mas podemos aplicar o **princípio da abstracção** baseados na ideia de que só a função é que muda. O objectivo é ter uma única definição para **todos** os casos. Isto consegue-se passando a função como argumento do programa geral. Lembre-se que as definições também são objectos! Daí o novo código que se segue.

```

1 def transforma_imagem(imagem, funcao):
2 """ Manipula uma imagem de acordo com uma função."""
3 largura = imagem.getWidth()

```

```

4 altura = imagem.getHeight()
5 nova_imagem = cImage.EmptyImage(largura,altura)
6 for coluna in range(largura):
7 for linha in range(altura):
8 pixel = imagem.getPixel(coluna,linha)
9 novo_pixel = funcao(pixel)
10 nova_imagem.setPixel(coluna,linha, novo_pixel)
11 return nova_imagem

```

O programa específico de transformação é usado por um programa principal que se encarrega de mostrar as imagens numa janela.

```

1 def main_funcao(imagem_ficheiro, funcao):
2 """Transforma uma imagem de acordo com a funcao."""
3 # Obtém imagem
4 imagem = cImage.FileImage(imagem_ficheiro)
5 # Transforma a imagem
6 imagem_nova = transforma_imagem(imagem, funcao)
7 # Define janela
8 largura = imagem.getWidth()
9 altura = imagem.getHeight()
10 janela = cImage.ImageWin(funcao.__name__,2*largura,
11 altura)
12 # vizualiza
13 imagem.draw(janela)
14 imagem_nova.setPosition(largura+1,0)
15 imagem_nova.draw(janela)
16 # Termina
17 janela.exitOnClick()

```

Notar o modo como definimos o título da janela usando o atributo `__name__` do objecto `funcao`.

## Brilho

Nem sempre podemos usar este padrão em toda a sua pureza. Escurecer, ou tornar mais clara, um imagem resume-se a diminuir, ou a aumentar, respectivamente, de um certo valor a intensidade de cada canal. Como se trata de uma operação que envolve toda a imagem, basta então definir a função de transformação de um pixel e usar o mesmo modelo geral dos casos anteriores.

```

1 def brilho_pixel(pixel,brilho):

```

```

2 """ Altera o brilho do pixel."""
3 r = restringe(pixel.getRed() + brilho, 0, 255)
4 g = restringe(pixel.getGreen() + brilho, 0, 255)
5 b = restringe(pixel.getBlue() + brilho, 0, 255)
6 novo_pixel = cImage.Pixel(r,g,b)
7 return novo_pixel
8
9 def restringe(canal,inf,sup):
10 if canal > sup:
11 canal = sup
12 elif canal < inf:
13 canal = inf
14 return canal

```

Para escurecer o valor do brilho deve ser negativo e para aumentar deve ser positivo. As imagens que se seguem (ver figura 9.12) resultam de uma variação de 100. Notar que os valores têm que ser eventualmente truncados, recorrendo à função genérica `restringe`, para não saírem do intervalo permitido, no caso de imagens (0,255).

## 9.7 Exemplos complementares

Apresentados os princípios básicos e alguns exemplos de programas que modificam de modo uniforme uma imagem, dando origem a um padrão de programa, está na altura de passar a exemplos diferentes.

### Colorir o chão

Suponha que pretende cobrir o chão da sua cozinha com quadrados coloridos. As cores devem ser colocadas de acordo com um padrão. A figura 9.13 mostra um exemplo possível. As duas cores são parte da especificação, ou seja devem ser consideradas como parâmetros do programa.

Olhando para a figura acima vemos que existe um **padrão**: os quadrados são colocados de modo alternado. Por outro lado, como é dito as formas são quadradas. Finalmente, embora na imagem a cozinha tenha uma dimensão  $4 \times 3$ , nós queremos uma solução **geral**, ou seja, para cozinhas de dimensão  $n \times m$ . Do ponto de vista informático o que precisamos fazer? De acordo com o princípio geral deste tipo de aplicações vamos ter que responder a três sub-problemas:

- Definir uma janela onde possam ser colocados os azulejos;

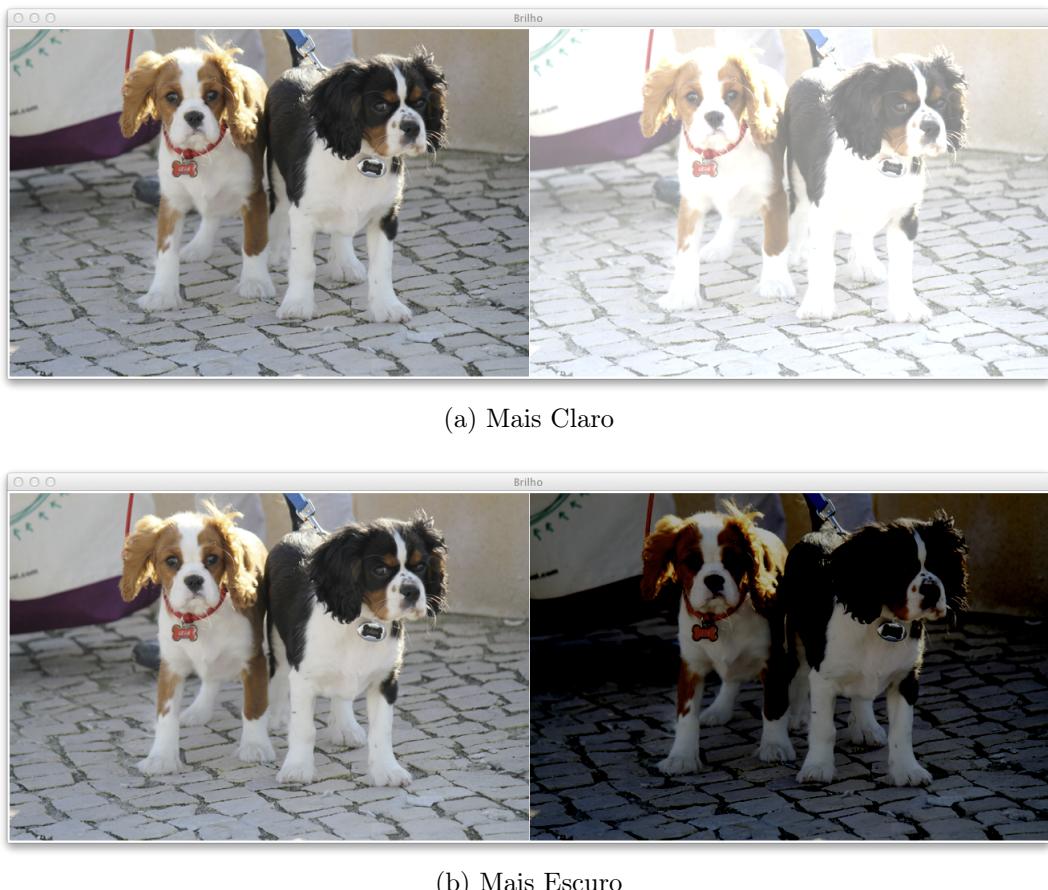


Figura 9.12: Alterando o brilho

- Gerar os quadrados de duas cores
- Posicionar os quadrados na janela e mostrar a imagem

A primeira questão, obriga-nos a saber a dimensão dos quadrados e as dimensões  $n \times m$  da cozinha. Claro que as duas cores também são importantes. Mas disso trataremos de seguida. Daí o nosso programa principal poder ser definido do seguinte modo:

```

1 def main1(nx,ny,lado,cores):
2 janela = cImage.ImageWin('Ladrilhos',nx * lado, ny*lado)
3 # -- resto do programa a definir aqui
4 janela.exitOnclick()

```

A segunda questão, remete para a geração de quadrados coloridos. Trata-se de uma questão geral, pelo que a solução para o problema de gerar quadrados de **duas ou mais** cores diferentes pode ser resolvido por um único

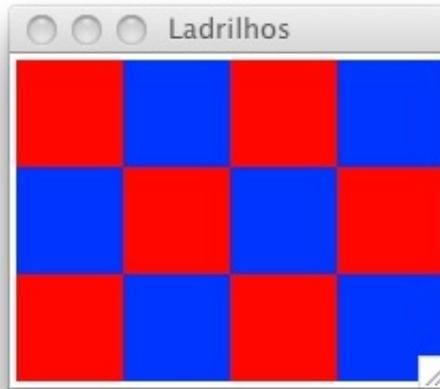


Figura 9.13: O chão da cozinha

programa que gera um quadrado de uma cor específica<sup>15</sup>. Esse programa pode depois ser chamado tantas vezes quantas as necessárias sendo a cor um parâmetro do programa.

```

1 def quadrado(lado,cor):
2 """ Cria um quadrado colorido de lado. """
3 imagem = cImage.EmptyImage(lado,lado)
4 pixel = cImage.Pixel(cor[0],cor[1],cor[2])
5 for linha in range(lado):
6 for coluna in range(lado):
7 imagem.setPixel(coluna,linha,pixel)
8 return imagem

```

Como se pode ver, começamos por criar uma imagem vazia, com as dimensões apropriadas. Depois definimos um pixel com a cor indicada. A cor é dada por um tuplo com os valores (r,g,b). Finalmente, colocamos o pixel em cada ponto da imagem.

Falta agora a parte mais difícil, a questão três: posicionar os quadrados e mostrar a imagem. Pensemos um pouco e olhemos para a figura 9.14, que ilustra o caso de quadrados 2 x 2, para serem colocados numa cozinha 4 x 3.

Os círculos alaranjados indicam os pontos onde cada quadrado deve começar a ser desenhado. Correspondem aos índices de uma matriz 4 x 3. (0,0), (1,0), (2,0) .... É aí que devemos colocar as imagens. Mas, é claro, que

---

<sup>15</sup>Aplicamos aqui de novo o princípio da abstracção procedural.

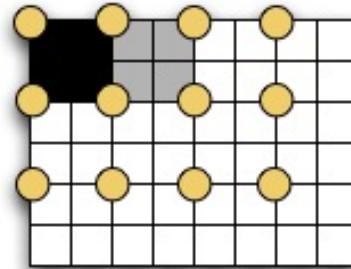


Figura 9.14: O problema do posicionamento

neste caso estes pontos estão afastados entre si do valor do lado. Assim, na realidade, em termos da imagem, esses pontos são  $(0,0)$ ,  $(\text{lado},0)$ ,  $(2*\text{lado}, 0)$ , ..... Assim o problema reduz-se a colocar  $4 \times 3 = 12$  imagens nas posições referidas. Isso faz-se com dois ciclos **for**, um dentro do outro, em que são gerados os **índices** da matriz, calculando-se em função deles o posicionamento na **imagem**. Para simplificar admitamos, por agora, que os quadrados são todos da mesma cor:

```

1 def cozinha_mono(nx,ny,lado, cor, janela):
2 """ Desenha os azulejos todos da mesma cor."""
3 for coluna in range(nx):
4 for linha in range(ny):
5 imagem = quadrado(lado, cor)
6 imagem.setPosition(coluna * lado,linha * lado)
7 imagem.draw(janela)

```

Se executarmos o programa apenas vemos desenhar uma cozinha com o chão todo da mesma cor. Não é muito impressionante! Então com alternar entre duas cores?? Um modo possível é notar que a soma das posições  $(x,y)$  do canto superior esquerdo das imagens são, alternadamente, pares e ímpares. E sabermos como determinar se um número é par. Logo:

```

1 def cozinha_poli(nx,ny,lado, cores, janela):
2 """ Desenha os azulejos com duas cores alternadas."""
3 for coluna in range(nx):
4 for linha in range(ny):
5 if (coluna + linha)%2 == 0:
6 # par
7 imagem = quadrado(lado, cores[0])
8 else:
9 # ímpar

```

```

10 imagem = quadrado(lado, cores[1])
11
12 imagem.setPosition(coluna * lado, linha * lado)
13 imagem.draw(janela)

```

Juntando agora todas as peças temos o programa completo:

```

1 import cImage
2
3 def cozinha_poli(nx,ny,lado, cores, janela):
4 """ Desenha os azulejos com duas cores alternadas."""
5 for coluna in range(nx):
6 for linha in range(ny):
7 if (coluna + linha)%2 == 0:
8 # par
9 imagem = quadrado(lado, cores[0])
10 else:
11 # ímpar
12 imagem = quadrado(lado, cores[1])
13
14 imagem.setPosition(coluna * lado,linha * lado)
15 imagem.draw(janela)
16
17 def quadrado(lado,cor):
18 """
19 Cria um quadrado colorido de lado.
20 """
21 imagem = cImage.EmptyImage(lado,lado)
22 pixel = cImage.Pixel(cor[0],cor[1],cor[2])
23 for linha in range(lado):
24 for coluna in range(lado):
25 imagem.setPixel(coluna,linha,pixel)
26 return imagem
27
28 def main1(nx,ny,lado,cores):
29 janela = cImage.ImageWin('Ladrilhos',nx * lado, ny*lado)
30 cozinha_poli(nx,ny,lado,cores, janela)
31 janela.exitOnClick()
32
33 if __name__=='__main__':
34 main1(4,3,50,[(255,0,0),(0,0,255)])

```

### Distorcer uma imagem

Pretendemos implementar uma função que nos permita ampliar uma imagem eventualmente com distorção. A figura 9.15 ilustra o que se pretende. No exemplo apresentado a imagem foi ampliada duas vezes em largura e três vezes em altura.

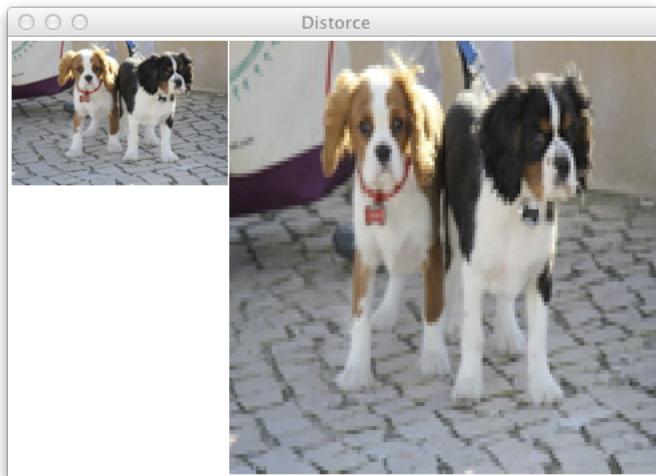


Figura 9.15: Distorcer uma imagem

Como podemos resolver esta questão? Uma ideia muito simples consiste em *ampliar cada pixel* de acordo com dois factores: um para a largura ( $factor_x$ ) e outro para a altura ( $factor_y$ ). Ou seja, Um pixel vai ser clonado um número de vezes igual a  $factor_x \times factor_y$ . A figura 9.16 exemplifica a ideia para o caso de  $3 \times 2$ .

Entendida a ideia fica a questão de saber como determinar as posições dos pixéis na imagem final. Após alguma reflexão não é difícil chegar à seguinte solução semelhante à encontrada para o problema dos ladrilhos (ver uma vez mais a figura 9.16):

```

1 pixel = imagem.getPixel(coluna, linha)
2 # repete o pixel numa área definida pelos factores
3 for i_x in range(factor_x):
4 for i_y in range(factor_y):
5 nova_imagem.setPixel(factor_x * coluna +
 i_x, factor_y * linha + i_y, pixel)

```

Neste modo, um pixel vai ser clonado numa área rectangular definida pelos dois factores de distorção. Este processo deve ser repetido para todos

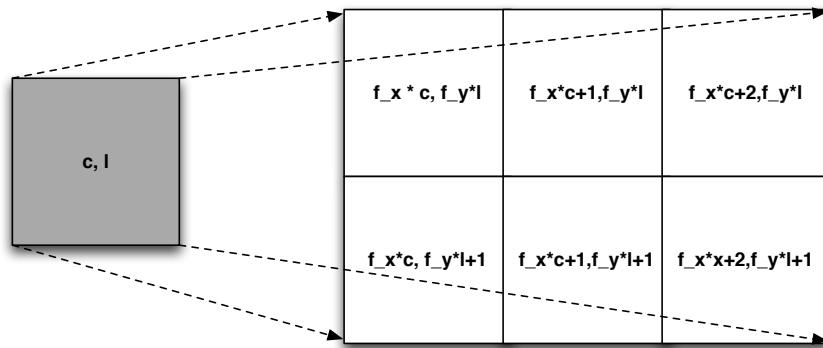


Figura 9.16: Clonar um pixel

os pixels da imagem original. Os exemplos anteriores mostram como se pode percorrer uma imagem pixel a pixel. Daí o código:

```

1 def distorcer(imagem, factor_x, factor_y):
2 """
3 Altera a imagem de acordo com os factores.
4 Estes devem ser números inteiros positivos.
5 """
6 largura = imagem.getWidth()
7 altura = imagem.getHeight()
8 nova_imagem = cImage.EmptyImage(factor_x * largura,
9 factor_y * altura)
10 for coluna in range(largura):
11 for linha in range(altura):
12 pixel = imagem.getPixel(coluna, linha)
13 # repete o pixel numa área definida pelos factores
14 for i_x in range(factor_x):
15 for i_y in range(factor_y):
16 nova_imagem.setPixel(factor_x * coluna +
17 i_x, factor_y * linha + i_y, pixel)
18 return nova_imagem

```

O que falta fazer resume-se: (1) a criar uma janela onde se possam guardar as imagens e, (2) mostrar o resultado.

```

1 def ampliar(imagem, factor_x, factor_y):
2 """
3 Distorce uma imagem de acordo com os factores indicados.

```

```

4 Cada pixel vai darorigem a um rectângulo de dimensões
5 factor_x X factor_y.
6 """
7 # Cria imagens
8 img = cImage.FileImage(imagem)
9 nova_img = distorcer(img, factor_x,factor_y)
10 # Cria janela
11 largura = img.getWidth()
12 altura = img.getHeight()
13 janela = cImage.ImageWin('Distorce', (factor_x + 1) *
14 largura , factor_y * altura)
15 # Coloca imagens
16 img.draw(janela)
17 nova_img.setPosition(largura + 1,0)
18 nova_img.draw(janela)
19 # Termina
janela.exitOnClick()

```

Note-se como se determina o tamanho da janela e a forma como se procede ao posicionamento das imagens.

### Espelho

Vamos agora resolver o problema de pegar numa metade de uma imagem efectuar uma cópia e juntar as duas partes como se uma fosse a imagem no espelho da outra. Vamos supor que usamos a metade esquerda da nossa imagem original. A figura 9.17 mostra o efeito pretendido.



Figura 9.17: Espelho vertical

A questão que se coloca é a de saber onde vai ficar a cópia de um dado pixel da metade esquerda. Uma observação evidente é de que, dado o facto de o espelho ser feito em função de um eixo vertical, a linha do pixel e da cópia é a mesma. Já em relação à coluna note-se que estes dois pixeis devem estar a igual distância da extremidades (ou do meio) da imagem final. A figura 9.18 tenta mostrar a situação.

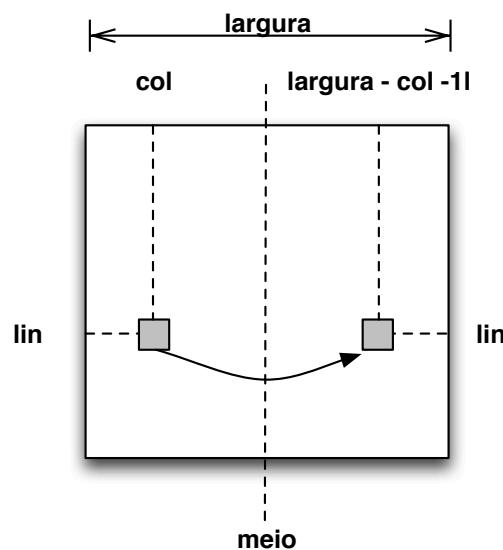


Figura 9.18: Onde colocar os pixeis?

Com a informação e os exemplos que já acumulámos vamos chegar sem dificuldades a uma solução informática.

```
12
13
14 def espelhar(imagem_fich):
15 imagem = cImage.FileImage(imagem_fich)
16 largura = imagem.getWidth()
17 altura = imagem.getHeight()
18 janela = cImage.ImageWin('Espelho Vertical - Esquerda', 2*
19 largura,altura)
20 nova_imagem = espelho_v_e(imagem)
21 nova_imagem.setPosition(largura + 1, 0)
22 imagem.draw(janela)
23 nova_imagem.draw(janela)
24 janela.exitOnClick()
```

A primeira função cria a janela, cria a nova imagem e coloca tudo na janela. A segunda função (**espelhar**) faz o trabalho de manipulação da imagem de acordo com a estratégia delineada.

## 9.8 Filtros

Acontece com frequência vermos imagens que sofrem de um problema conhecido como pixelização: de um modo simples esse efeito caracteriza-se por conseguirmos ver os pixels. Muitas vezes tal resulta, por exemplo, de termos deliberadamente manipulado a imagem para esconder certa informação ou de termos ampliado a imagem. A baixa resolução origina o efeito de pixelização. Uma maneira de melhorar a qualidade da imagem baseia-se numa técnica que permite suavizar as transições entre intensidades de pixels vizinhos. A figura 9.19 mostra uma imagem pixelizada e o resultado da suavização das transições.

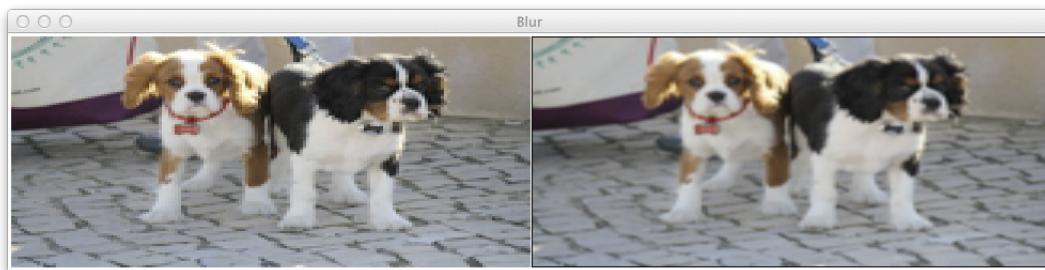


Figura 9.19: Tratamento da pixelização

É notório o desaparecimento do efeito de escada nas linhas inclinadas. Vamos ver como podemos implementar o respectivo programa de suavização da imagem. Para tal iremos modificar o valor da intensidade de um pixel em função das intensidades dos seus 9 vizinhos (ver figura 9.20).

|             |           |             |
|-------------|-----------|-------------|
| $(x-1,y-1)$ | $(x,y-1)$ | $(x+1,y-1)$ |
| $(x-1,y)$   | $(x,y)$   | $(x+1,y)$   |
| $(x+1,y+1)$ | $(x,y+1)$ | $(x+1,y+1)$ |

Figura 9.20: Vizinhança

A função que vamos considerar substitui a intensidade de cada canal pelo valor **médio** das intensidades dele e dos seus vizinhos. Comecemos por resolver essa questão.

```

1 def media(coluna, linha, imagem):
2 """Calcula o valor médio dos pixels na vizinhança do pixel
3 (coluna,linha)."""
4 # Extrai pixels
5 vizinhos = []
6 for c in [-1,0,1]:
7 for l in [-1,0,1]:
8 vizinhos.append(imagem.getPixel(coluna+c,linha+l))
9 # Calcula pixel "médio" por canal
10 r = sum([vizinhos[i].getRed() for i in range(9)])//9
11 g = sum([vizinhos[i].getGreen() for i in range(9)])//9
12 b = sum([vizinhos[i].getBlue() for i in range(9)])//9
13 # Constrói Pixel
14 novo_pixel = cImage.Pixel(r,g,b)
15 return novo_pixel

```

A nossa solução desdobra-se em três passos: (1) extrair a lista dos 9 pixels que formam a vizinhança (linhas 4 a 7); (2) calcular o valor médio por canal (linhas 9 a 11); e (3) construir o novo pixel (linha 13). A extracção dos pixels vizinhos é feita fabricando o endereço de cada pixel através da sua posição **relativa** ao pixel que estamos a considerar. No segundo passo, vamos identificar todos os valores de cada canal específico (*red*, *green* e *blue*) e calcular a sua média. O uso de listas por compreensão ajuda-nos a resolver a questão e a tornar o programa mais legível.

A solução apresentada resulta de uma decomposição em sub-problemas que se traduz por primeiro determinar **todos** os vizinhos e só depois ir fazer a extracção da intensidade por canal. Esta é uma abordagem simples e que nos permite diminuir a complexidade do problema inicial. É evidente que podemos proceder de outro modo juntando as duas fases: cada pixel é extraído e actualiza-se a soma.

```

1 def media_2(coluna, linha, imagem):
2 """Calcula o valor médio dos pixels na vizinhança do pixel
3 (coluna,linha)."""
4 r,g,b = 0, 0, 0
5 for c in [-1,0,1]:
6 for l in [-1,0,1]:
7 nova_coluna = coluna+c
8 nova_linha = linha+l
9 pixel = imagem.getPixel(nova_coluna, nova_linha)
10 # Actualiza pixel por canal
11 r += pixel.getRed()
12 g += pixel.getGreen()
13 b += pixel.getBlue()
14
15 novo_pixel = cImage.Pixel(r//9,g//9,b//9)
16 return novo_pixel

```

Deixamos ao leitor a reflexão acerca de qual das duas soluções é a melhor.

Sabemos agora como obter o valor médio de um pixel quando se considera a sua vizinhança. Falta agora percorrer a imagem, pixel a pixel, calcular o valor médio e construir a nova imagem. Mas isso já temos vindo a fazer ao longo do capítulo. Vejamos então o resultado.

```

1 def suaviza(imagem):
2 """Suaviza uma imagem."""
3 # Inicializa
4 largura = imagem.getWidth()

```

```

5 altura = imagem.getHeight()
6 nova_imagem = cImage.EmptyImage(largura,altura)
7 # Percorre a imagem e calcula
8 for coluna in range(1,largura-1):
9 for linha in range(1,altura-1):
10 novo_pixel = media(coluna,linha,imagem)
11 nova_imagem.setPixel(coluna,linha,novo_pixel)
12 return nova_imagem

```

Esta solução tem um aspecto menos simpático. Os pixeis da primeira e última linha e os da primeira e última coluna não têm os 9 vizinhos. Optámos então por não calcular os respectivos valores médios, razão pela qual os dois ciclos imbricados do programa evitam esses elementos. Deixamos ao leitor o cuidado de pensar uma solução em que esse problema desapareça (ver exercício 9.9).

## Convolução

A operação que descrevemos para cada pixel pode ser apresentada de outro modo. O novo valor de cada pixel é calculado como:

$$pixel(x, y) = \frac{1}{9} \times \left( \sum_{c=-1}^1 \sum_{l=-1}^1 pixel(x + c, y + l) \right)$$

Esta expressão pode ser manipulada para se obter.

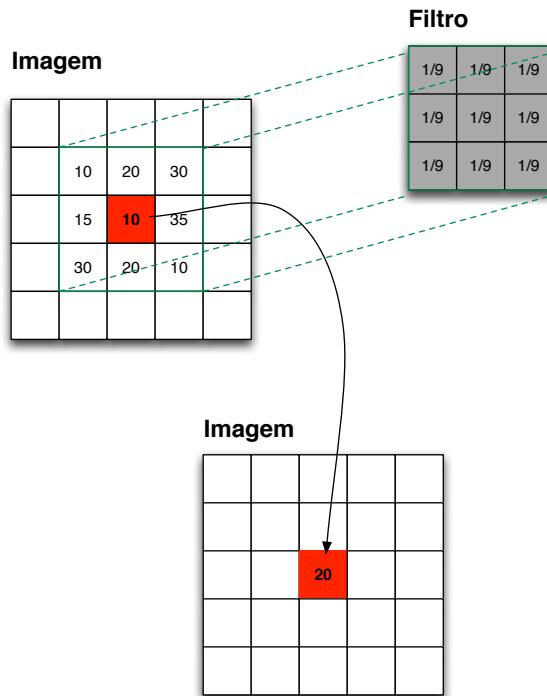
$$pixel(x, y) = \left( \sum_{c=-1}^1 \sum_{l=-1}^1 \frac{1}{9} \times pixel(x + c, y + l) \right)$$

Isto significa que a **soma do produto** de  $\frac{1}{9}$  pelo valor de cada elemento da vizinhança. Podemos ainda visualizar esta operação de um modo simples, com se vê na figura 9.21. Na figura, para simplificar, apresentamos apenas os valores hipotéticos para um dos canais.

### Filtro

Dizemos que aplicamos um **filtro** à imagem<sup>16</sup>. O procedimento consiste percorrer a imagem pixel a pixel, alinhando o **centro** do filtro com o pixel cujo novo valor queremos calcular e efectuando os cálculos com toda a vizinhança de acordo com a fórmula apresentada. Podemos generalizar esta operação considerando a possibilidade de os elementos do filtro terem valores diferentes. A figura 9.22 mostra um exemplo da família dos filtros ditos **gaussianos**. Os valores aparecem na forma de frações em que o denominador (no exemplo 16) é igual à soma de todos numeradores.

<sup>16</sup>Também designado por máscara ou núcleo.



$$\text{pixel}(x, y) = \left( \sum_{c=-1}^1 \sum_{l=-1}^1 \frac{1}{9} \times \text{pixel}(x + c, y + l) \right)$$

Figura 9.21: Aplicar um filtro a uma imagem

Podemos adaptar a fórmula anterior de modo trivial:

$$\text{pixel}(x, y) = \left( \sum_{c=-1}^1 \sum_{l=-1}^1 \text{filtro}(c, l) \times \text{pixel}(x + c, y + l) \right)$$

Em termos gerais designamos esta operação de **convolução**. A partir da última fórmula facilmente se chega ao código que efectua a operação de convolução a um pixel.

```

1 def convolucao(x,y, imagem, filtro):
2 index = len(filtro) // 2
3 r,g,b = 0, 0, 0
4 for i in range(-index, index+1):
5 for j in range(-index, index+1):
6 pixel = imagem.getPixel(x+j,y+i)
7 r += pixel.getRed() * filtro[j+index][i+index]
8 g += pixel.getGreen() * filtro[j+index][i+index]
```

|      |      |      |
|------|------|------|
| 1/16 | 2/16 | 1/16 |
| 2/16 | 4/16 | 2/16 |
| 1/16 | 2/16 | 1/16 |

Figura 9.22: Filtro gaussiano

```

9 b += pixel.getBlue() * filtro[j+index][i+index]
10 r = restringe(int(r),0,255)
11 g = restringe(int(g),0,255)
12 b = restringe(int(b),0,255)
13 novo_pixel = cImage.Pixel(r, g,b)
14 return novo_pixel

```

Note-se como se efectua a conversão das coordenadas do filtro relativas ao seu centro em coordenadas do filtro encarado como uma matriz, devido à sua representação como lista de listas. Atente-se ainda na necessidade de verificar se os valores obtidos se encontram na gama permitida, isto é, neste caso no intervalo (0, 255). Finalmente, verifique que o programa funciona desde que os filtros sejam matrizes quadradas  $n \times n$ , com  $n$  um número ímpar. Como se pode ver pela imagem da figura 9.23 este filtro melhora a qualidade da imagem contrariando o efeito da pixelização.



Figura 9.23: Aplicando um filtro gaussiano

Para correr o programa necessitamos apenas de criar o programa envolvente que percorre a imagem pixel a pixel e desenha as imagens numa janela.

```
1 import cImage
2
3 def convolucao(x,y, imagem, filtro):
4 index = len(filtro) // 2
5 r,g,b = 0, 0, 0
6 for i in range(-index, index+1):
7 for j in range(-index, index+1):
8 pixel = imagem.getPixel(x+j,y+i)
9 r += pixel.getRed() * filtro[j+index][i+index]
10 g += pixel.getGreen() * filtro[j+index][i+index]
11 b += pixel.getBlue() * filtro[j+index][i+index]
12 r = restringe(int(r),0,255)
13 g = restringe(int(g),0,255)
14 b = restringe(int(b),0,255)
15 novo_pixel = cImage.Pixel(r, g, b)
16 return novo_pixel
17
18
19 def mostra_convol(imagem, filtro):
20 """ Convolução de uma imagem."""
21 # Cria imagens
22 img = cImage.FileImage(imagem)
23 nova_img = transforma_convol(img,filtro)
24 # Cria janela
25 largura = img.getWidth()
26 altura = img.getHeight()
27 janela = cImage.ImageWin('Convolução',2 * largura, altura
28)
29 # Coloca imagens
30 nova_img.setPosition(largura+1,0)
31 img.draw(janela)
32 nova_img.draw(janela)
33 # Termina
34 janela.exitOnClick()
```

### Extrair características

Existem muitos filtros cada um com a sua finalidade<sup>17</sup>. Vamos agora expor de modo simples o uso conjugado da duas máscaras para a extracção de lados numa imagem. As máscaras em questão são conhecidas por operadores de Sobel. A figura 9.24 mostra as respectivas matrizes. A máscara da esquerda permite identificar linhas verticais e a da direita linhas horizontais.

|    |   |   |
|----|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

**Vertical**

|    |    |    |
|----|----|----|
| -1 | -2 | -1 |
| 0  | 0  | 0  |
| 1  | 2  | 1  |

**Horizontal**

Figura 9.24: Operadores de Sobel

O algoritmo de detecção trabalha com uma imagem em escala de cinzentos. Funciona aplicando o filtro vertical,  $val_x$ , seguido do horizontal,  $val_y$  a cada pixel. Depois calcula a intensidade recorrendo à fórmula  $\sqrt{val_x^2 + val_y^2}$ <sup>18</sup>. Restringe o valor obtido ao intervalo (0, 255) e define o correspondente pixel.

A implementação do algoritmo é a seguinte.

```

1 import cImage
2 import math
3
4 def extrai_caract(imagem, mascara_1, mascara_2, limiar):
5 """ Convolução de uma imagem."""
6 # Cria imagens
7 img_cor = cImage.FileImage(imagem)
8 img = imagem_cinzentos(img_cor)
9 nova_img = transforma_convol(img, mascara_1, mascara_2,
10 limiar)
11 # Cria janela
12 largura = img.getWidth()
13 altura = img.getHeight()
```

<sup>17</sup> Ao leitor interessado sugerimos a consulta de um livro sobre processamento de imagens digitais. Nele encontrará seguramente toda a teoria que explica o uso dos filtros.

<sup>18</sup> A justificação deste método está fora do âmbito deste texto.

```

13 janela = cImage.ImageWin('Extrai Características',2 *
14 largura, altura)
15 # Coloca imagens
16 nova_img.setPosition(largura+1,0)
17 img.draw(janela)
18 nova_img.draw(janela)
19 # Termina
20 janela.exitOnClick()
21
22 def transforma_convol(imagem,mascara_1, mascara_2,limiar):
23 """Imagem em escala de cinzentos."""
24 largura = imagem.getWidth()
25 altura = imagem.getHeight()
26 nova_imagem = cImage.EmptyImage(largura, altura)
27 index = len(mascara_1)//2
28 for coluna in range(index,largura-index):
29 for linha in range(index,altura-index):
30 val_x = convolve(coluna,linha,imagem, mascara_1)
31 val_y = convolve(coluna,linha,imagem, mascara_2)
32 val = restringe(math.sqrt(val_x**2 + val_y**2), 0,
33 255)
34 novo_pixel = preto_branco_pixel(cImage.Pixel(val,
35 val, val), limiar)
36 nova_imagem.setPixel(coluna,linha,novo_pixel)
37 return nova_imagem
38
39 def convolve(x,y, imagem, filtro):
40 """Imagem em escala de cinzentos."""
41 index = len(filtro) // 2
42 r = 0
43 for i in range(-index, index+1):
44 for j in range(-index, index+1):
45 pixel = imagem.getPixel(x+j,y+i)
46 r += pixel.getRed() * filtro[j+index][i+index]
47 r = restringe(int(r),0,255)
48 return r

```

Executando o programa para diferentes valores do limiar obtemos resultados diversos (ver figura 9.25). A qualidade das resultado também depende das características das imagens.

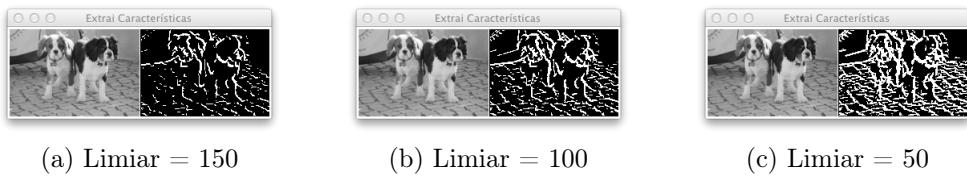


Figura 9.25: Operadores de Sobel

## 9.9 O formato de imagem PPM

Uma imagem pode ser armazenada em ficheiro de forma bastante simples, recorrendo ao formato PPM (Portable Pixel Map) que pode existir no modo RAW ou no modo ASCII. É sobre ficheiros neste último modo que nos iremos debruçar. Um ficheiro PPM em modo ASCII começa por uma linha com a palavra “P3”, que identifica o formato PPM nesse modo. A segunda linha do mesmo ficheiro armazena um comentário, normalmente utilizado para identificar a aplicação responsável pela criação do ficheiro. A terceira linha armazena dois valores inteiros, separados por um espaço: o primeiro valor representa a largura da imagem e o segundo a sua altura. A quarta linha do ficheiro armazena um valor inteiro, que indica o valor máximo de cor presente na imagem. A partir da quinta linha, são armazenados os valores dos componentes RGB (Red, Green, Blue) de cada píxel da imagem. Por exemplo, a quinta linha armazena o valor de **vermelho** do primeiro píxel, a sexta linha o valor de **verde** do mesmo píxel e a sétima linha o seu valor de **azul**. As 3 linhas seguintes no ficheiro armazenam os valores RGB do segundo píxel da imagem, e assim sucessivamente. O exemplo seguinte ilustra as primeiras 10 linhas de um ficheiro PPM no formato ASCII. Trata-se neste exemplo de uma imagem criada pela aplicação GIMP e com dimensão de 200x123 píxeis:

```
P3
CREATOR: GIMP PNM Filter Version 1.1
200 123
255
133
120
117
123
134
212
...
...
```

As imagens no formato PPM podem ser visualizadas recorrendo a várias aplicações, entre as quais o popular GIMP<sup>19</sup>. O que nos interessa aqui é mostrar como podemos implementar um conversor entre imagens no formato que temos vindo a usar, em particular o formato jpg, e no formato ppm.

### De ppm para jpg

A tarefa de converter de ppm para o formato jpg será feito tendo por base o conhecimento que temos do formato destes ficheiros no modo ASCII. Na prática temos que implementar um pequeno analisador sintático. Vejamos como resolvemos esta questão, socorrendo-nos do módulo cImage.

```

1 import cImage
2
3
4 def ppm_to_pil(imagem):
5 # carrega ficheiro
6 with open(imagem,'r') as ppm_img:
7 # extrai cabeçalho (modo e comentário)
8 ppm_img.readline()
9 ppm_img.readline()
10 # extrai dimensão
11 dim = ppm_img.readline().split()
12 largura = int(dim[0])
13 altura = int(dim[1])
14 # valor máximo de cor
15 max_cor = ppm_img.readline()
16 # cria imagem
17 pil_img = cImage.EmptyImage(largura,altura)
18 # converte
19 for lin in range(altura):
20 for col in range(largura):
21 r = int(ppm_img.readline().rstrip('\n'))
22 g = int(ppm_img.readline().rstrip('\n'))
23 b = int(ppm_img.readline().rstrip('\n'))
24 pixel = cImage.Pixel(r,g,b)
25 pil_img.setPixel(col,lin,pixel)
26 ppm_img.close()
27 return pil_img

```

<sup>19</sup>Pode ser obtido em <http://www.gimp.org>.

```

28
29 def mostra_ppm(imagem):
30 """ Procede à diminiuição da pixelização da imagem."""
31 # carrega e Converte
32 img = ppm_to_pil(imagem)
33 # Cria janela
34 largura = img.getWidth()
35 altura = img.getHeight()
36 janela = cImage.ImageWin('PPM to PIL', largura, altura)
37 # Salva imagem no disco
38 nome_ficheiro = imagem.split('.')[0] + '_to' + '.jpg'
39 img.save(nome_ficheiro)
40 # Coloca imagem
41 img.draw(janela)
42 # Termina
43 janela.exitOnClick()

```

Sendo um ficheiro ASCII ele é aberto para leitura do modo convencional (linha 6). Depois, entre as linhas 7 e 15, lemos a informação que descreve o ficheiro, guardando apenas o que é relevante: a dimensão.

### De jpg para PPM

O processo inverso, ou seja converter uma imagem jpg para PPM ASCII, também não oferece muitas dúvidas e permite-nos rever algumas instruções envolvidas na criação de ficheiros de texto.

```

1 def pil_to_ppm(imagem):
2 """Constrói e salva uma imagem em formato PPM, modo ASCII.
3 """
4 # carrega ficheiro origem
5 pil_img = cImage.FileImage(imagem)
6 largura = pil_img.getWidth()
7 altura = pil_img.getHeight()
8 nome_completo = imagem.split('/')[-1]
9 nome = '/Users/ernestojfcosta/Desktop/' + nome_completo.
10 split('.')[0] + '.ppm'
11 ficheiro = open(nome,'w')
12 # constrói cabeçalho (modo, comentário, dimensão, cor
13 máxima
14 ficheiro.write('P3\n')
15 ficheiro.write('# Criado por Ernesto Costa.\n')

```

```

13 ficheiro.write(str(largura) + ' ' + str(altura) + '\n')
14 ficheiro.write('255\n')
15 # Guarda pixeis
16 for lin in range(altura):
17 for col in range(largura):
18 pixel = pil_img.getPixel(col,lin)
19 r = str(pixel.getRed()) + '\n'
20 ficheiro.write(r)
21 g = str(pixel.getGreen()) + '\n'
22 ficheiro.write(g)
23 b = str(pixel.getBlue()) + '\n'
24 ficheiro.write(b)
25 ficheiro.close()
26 return ficheiro

```

Começamos por carregar o ficheiro fonte e extrair as suas dimensões (linhas 4 a 6). De seguida definimos o nome do novo ficheiro e abrimos para leitura (linhas 7 a 9). Escrevemos de seguida o cabeçalho, isto é as cinco primeiras linhas que irão conter por esta ordem: o código **P3** que identifica o ficheiro como PPM ASCII, um comentário, a largura, a altura e o valor máximo da cor. Notar a necessidade de terminar cada linha por **n**. De seguida retiramos cada pixel da imagem original, obtemos os valores dos três canais e escrevemos esses valores em três linhas consecutivas. Este processo é repetido um número de vezes igual à dimensão da imagem (linhas 16 a 24).

Com esta interface podemos agora efectuar todas as manipulações anteriores com mais um formato de imagem!

## Sumário

Neste capítulo introduzimos alguns conceitos sobre imagens, o modo como podem ser representadas e várias maneiras de as processar. Pusemos em relevo a existência de um padrão que envolve a travessia das imagens através de dois ciclos **for**. Discutimos ainda alguns aspectos de abstracção procedimental que se traduziu no uso de funções passadas como parâmetros. Introduzimos o módulo **cImage**, os seus tipos e operações. Criamos de seguida a nova imagem no formato pretendido (linha 17), por enquanto vazia. De seguida entramos num ciclo formado por dois **for** imbricados segundo uma lógica que já conhecemos. No interior do ciclo vamos buscar repetidamente sequências de três valores que formarão um pixel que iremos guardar na nova

imagem (linhas 21 a 25). A função `mostra_ppm` permite visualizar guardar a imagem no disco (linhas 38 e 39) e ainda visualizar a imagem criada (linha 41).

## Teste os seus conhecimentos

Verifique se conhece os conceitos discutidos neste capítulo e consegue responder às questões colocadas.

- O que entende por imagem *bitmap*
- O que entende por pixel?
- O que entende por resolução de uma imagem?
- O que entende por modo RGB?
- A ordem e o modo de percorrer uma imagem são irrelevantes. Concorda com esta afirmação?
- Quais as vantagens/inconvenientes de usar o mecanismo de abstracção procedural?
- O que entende por filtro ou máscara?
- O que entende por convolução
- O que entende por operadores de Sobel
- Quanto maior for o filtro usado melhor se consegue o efeito pretendido. Concorda com esta afirmação?
- Quando se usa um filtro, por que razão se cria uma nova imagem?

## Exercícios

20

### Exercício 9.1 F

Desenvolva um programa que lhe permita mostrar os elementos que formam a matriz triangular inferior de uma da matriz à semelhança do que foi

---

<sup>20</sup>Para não sobrecarregar não indicaremos de forma explícita que se fará uso do módulo `cImage`.

feito para o caso da matriz triangular superior. A listagem abaixo mostra o pretendido para a matriz

$$mat = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]$$

```

1 1
2 5 6
3 9 10 11

```

### Exercício 9.2 M

Desenvolva um programa que lhe permita extrair uma sub-matriz de uma matriz dada, conhecido o elemento inicial e as dimensão da sub-matriz. Por exemplo, se chamarmos o programa com:

```
1 print(sub_matriz([[1,2,3,4],[5,6,7,8],[9,10,11,12]],1,1,2,2))
```

o resultado obtido deverá ser:

```
1 [[6, 7], [10, 11]]
```

### Exercício 9.3 F

Desenvolva um programa que lhe permita gerar uma matriz de tuplos em que cada tuplo contém possíveis valores de pixeis. Os parâmetros do programa são a largura e a altura da imagem.

### Exercício 9.4 M

Desenvolva um programa que lhe permita desenhar uma linha recta numa janela, conhecidas as coordenadas de dois dos seus pontos. Relembreamos que a equação de uma recta é dada por:

$$y = \frac{y_2 - y_1}{x_2 - x_1} \times (x - x_1) + y_1$$

### Exercício 9.5 M

Desenvolva um programa que lhe permita desenhar um arco de circunferência com uma dada amplitude, conhecidos também as coordenadas do centro e o raio. Deve ser genérico: no limite deve permitir desenhar uma circunferência!

### Exercício 9.6 F

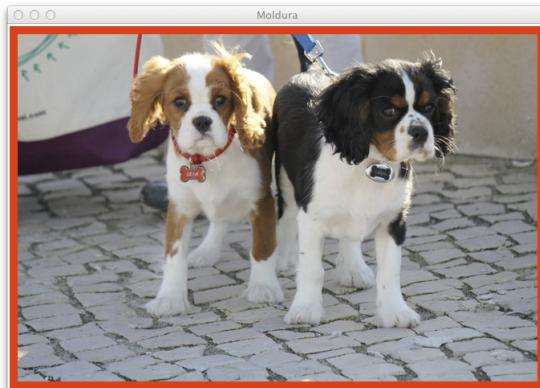


Figura 9.26: Acrescentar uma moldura

Pretende-se implementar um programa que permita adicionar uma moldura a uma imagem. Deve ser possível definir o tamanho da moldura e a sua cor. A figura 9.26 ilustra o pretendido.

#### Exercício 9.7 F

Implemente um programa que permita obter um corte de uma imagem. Para além da imagem devem ser fornecidos ao programa o ponto de início do corte e as dimensões. A figura 9.27 ilustra a ideia pretendida.



Figura 9.27: Corte de imagem

#### Exercício 9.8 M

Discutimos um modelo para transformar toda uma imagem alterando um a um os seus pixels de acordo com a mesma função de transformação. Vamos

aplicar esse modelo ao caso da transformação de uma imagem a cores numa a **preto e branco**. A figura 9.28 ilustra o pretendido. **Sugestão:** Para obter a imagem a preto e branco converta cada pixel para cinzento e depois compare com um limiar (por exemplo 128). Os maiores que o limiar são transformados em branco (ou seja (255,255,255)) e os menores ou igual em preto (ou seja em (0,0,0)).



Figura 9.28: A preto e branco

### Exercício 9.9 M

Desenvolva um programa que lhe permita variar a intensidade dos pixels de modo **independente**, isto é, a variação no vermelho, no verde e no azul pode ser diferente.

### Exercício 9.10 D

Vimos um exemplo de como ampliar uma imagem, eventualmente com distorção. O problema agora é o inverso. Implementar um programa que permita reduzir uma imagem, uma vez mais com eventual distorção. A figura 9.29 ilustra o que se pretende.

### Exercício 9.11 M

Vimos no texto como pegar na metade esquerda de uma imagem, criar uma cópia e juntar as duas partes como se fossem a imagem no espelho uma da outra. Pretende-se agora fazer algo semelhante, mas em que usamos a metade superior da imagem e o espelho é feito segundo uma perspectiva horizontal. A figura 9.30 mostra o efeito pretendido.

### Exercício 9.12 M

Muitas vezes acontece termos que reduzir as cores de uma imagem a um número limitado de cores. O processo de redução passa por determinar para

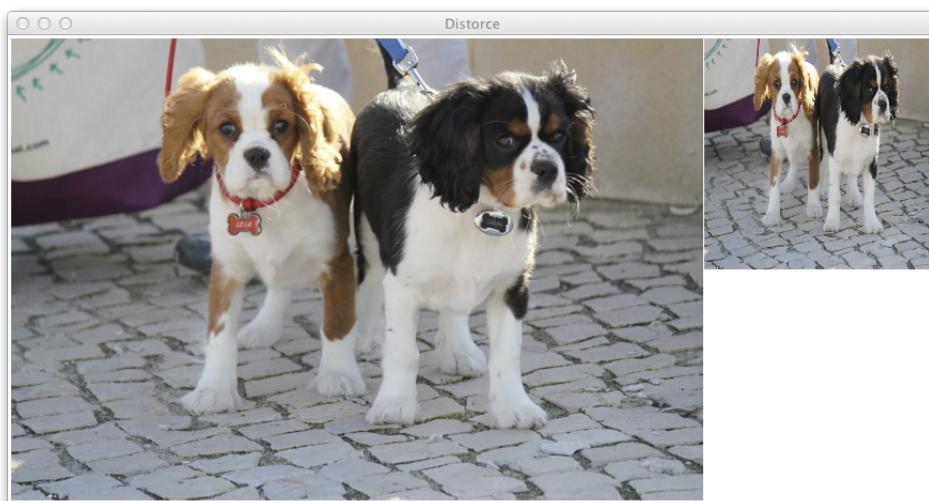


Figura 9.29: Redução de uma imagem

cada pixel qual a cor mais próxima de entre as cores disponíveis. Implemente o programa que lhe permite fazer essa transformação. Por exemplo, para a paleta de cores:

```
1 palete_cores = [(255,0,0), (0,255,0), (0,0,255), (0,0,0),
 (255,255,255), (255,255,0), (0,255,255), (255,0,255)]
```

A execução do nosso programa para a imagem com os dois cães, que temos vindo a usar ao longo do capítulo, resulta no que a figura 9.31 ilustra.

### Exercício 9.13 M

Pretendemos desenvolver um programa que permita fazer uma colagem a partir de várias imagens. As imagens podem ter tamanhos diferentes, terem sido manipuladas para criar diversos efeitos, e ser colocadas em posições específicas da janela de visualização.

### Exercício 9.14 M

Estudámos o problema de minimizar o efeito da pixelização. A nossa solução evita considerar o bordo da imagem o que se traduz por uma fina moldura de cor preta. Reveja o programa e altere-o por forma que tal não aconteça. A figura 9.32 mostra o resultado pretendido. **Sugestão:** no bordo calcule a média dos vizinhos existentes.

### Exercício 9.15 M

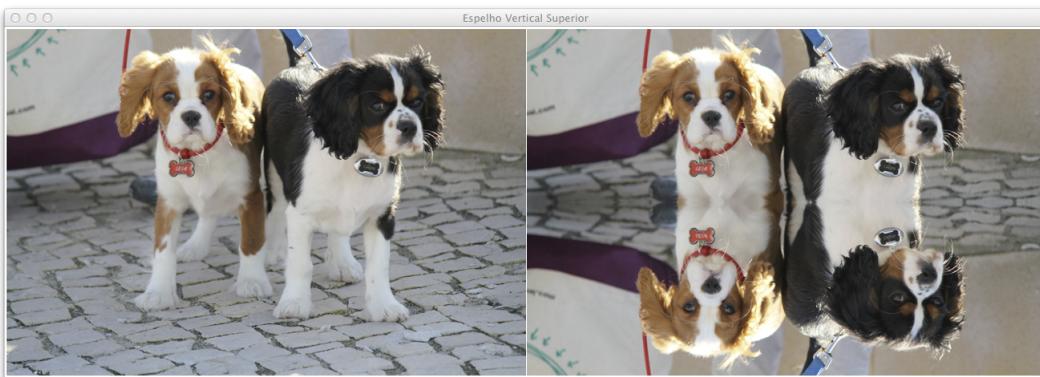


Figura 9.30: Espelho horizontal

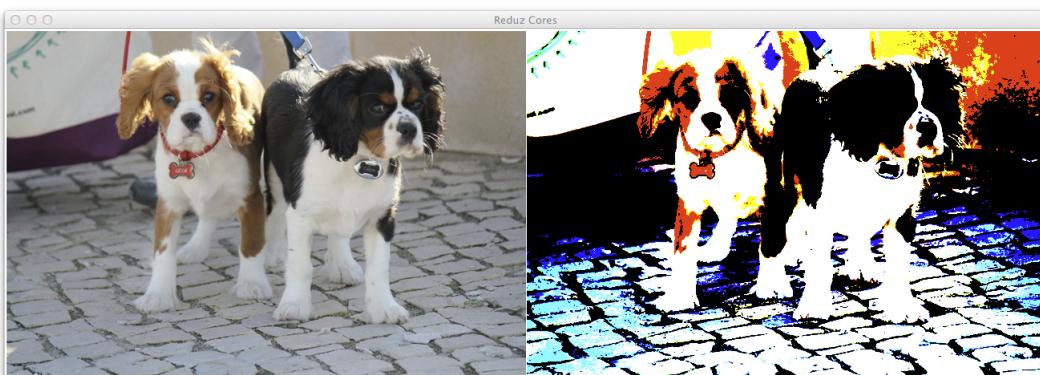


Figura 9.31: Redução de cores

Nas aulas falámos no conceito de Kernel (ou filtro, ou máscara) e mostrámos como o conceito podia ser usado para detectar os contornos de uma figura. Vamos agora explorar vários tipos de filtros. Faça uma pesquisa no **Google** pelas palavras **convolution** e **kernel** e descubra filtros diferentes dos apresentados neste capítulo. Escolha as imagens do seu agrado e aplique sobre elas o respectivo filtro. Visualize o resultado.

### Exercício 9.16 M

Um modo de eliminar o ruído de uma fotografia consiste em considerar para cada pixel e cada canal a mediana dos valores do pixel e dos seus vizinhos. O cálculo da mediana faz-se ordenando os valores e considerando o

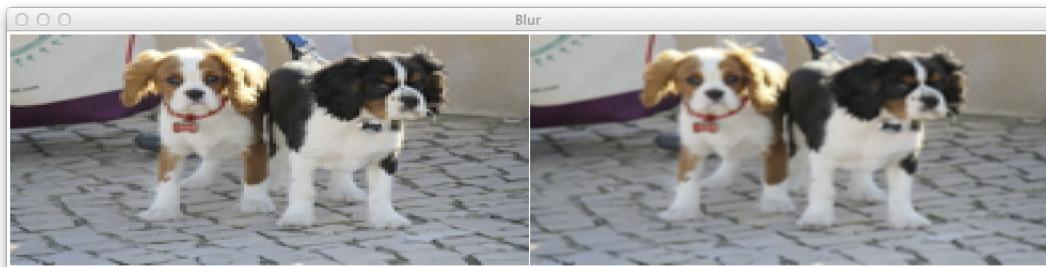


Figura 9.32: Suavizar sem moldura

que está na posição do meio. Implemente o respectivo programa.

### Exercício 9.17 M

Pretende-se um programa que permita pegar em duas imagens e fabricar uma terceira. Desenvolva o programa de forma modular de modo a controlar o processo que usa para combinar os pixéis das imagens. A título de exemplo, na imagem 9.33 apresentamos uma combinação entre Einstein e Gandhi.

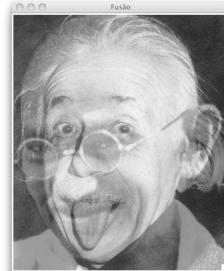


Figura 9.33: Gandstein

### Exercício 9.18 D

Desenvolva um programa que dada uma imagem a roda de 90 graus. O sentido é à sua escolha.

### Exercício 9.19 MD

Neste exemplo pretendemos resolver uma questão semelhante à do exemplo 9.9. Só que, agora, pretende-se que a rotação possa ser de um ângulo arbitrário. **Nota:** A resolução desta questão obriga a saber algo sobre trans-

lação de sistemas de eixos, de trigonometria e de álgebra.

**Exercício 9.20 MD** Muitas vezes queremos enviar imagens encriptadas. Para isso vamos criar um método que consiste em alterar a ordem das linhas da imagem original de modo aleatório. Quem receber a mensagem e conhecer o modo como se misturaram as linhas deve ser capaz de reconstruir a imagem original. Implemente os respectivos programas. A figura 9.34 ilustra o pretendido.

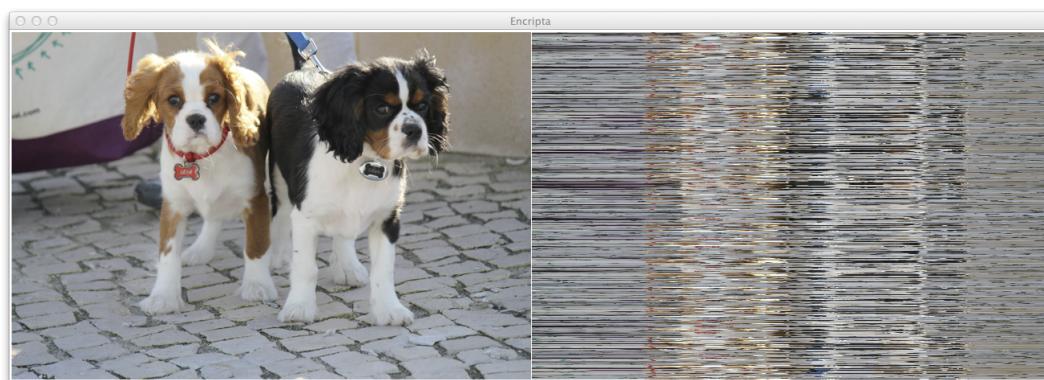


Figura 9.34: Encriptar uma imagem



# Capítulo 10

## Recursividade

**recursividade:** ver  
*recursividade*

---

*in* Dicionário

### Objectivos

- ✓ Introduzir o conceito de recursividade
- ✓ Explorar a recursividade por recurso a exemplos simples
- ✓ Perceber as virtudes (e os problemas) da recursividade

### 10.1 Conceitos

Já todos tivemos a experiência de nos colocarmos entre dois espelhos paralelos. E o que vemos sempre nos fascinou: a nossa imagem que se repete infinitas vezes. Também seguramente já olhámos para manifestações artísticas ou científicas em que a parte e o todo se não distinguem. À semelhança com o que se passa com a imagem no espelho temos a sensação que há um motivo que se repete *de fora para dentro*. Por exemplo na imagem 10.1 observamos o denominado *Sierpinski Gasket*. No triângulo mais externo marcámos os pontos médios dos lados. Esse pontos foram aproveitados para construir três novos triângulos. A cada um deles agora aplicamos o mesmo mecanismo e o resultado é aquilo que os nossos olhos vêm. A figura 10.2 mostra o processo usado nas três primeiras etapas<sup>1</sup>.

Estamos perante exemplos do conceito de recursividade: quando um ob-

[Recursividade](#)

---

<sup>1</sup>As escalas das duas figuras são diferentes.

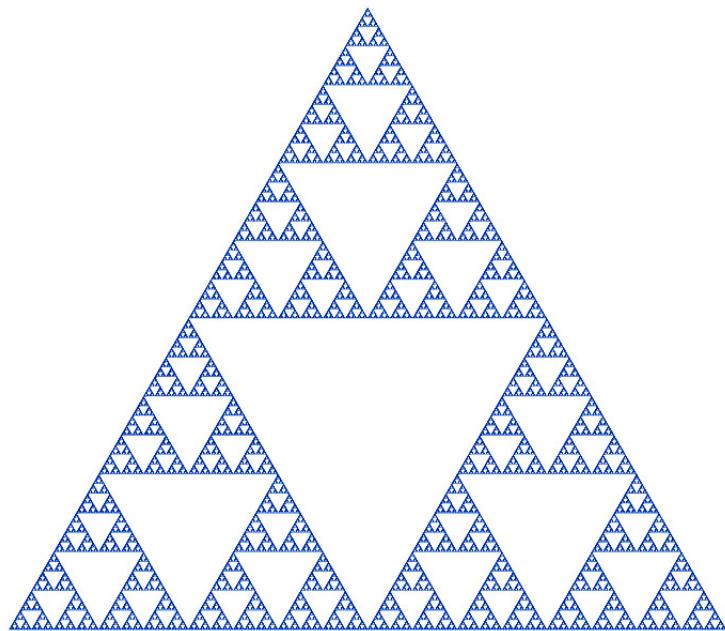


Figura 10.1: *Sierpinski Gasket*

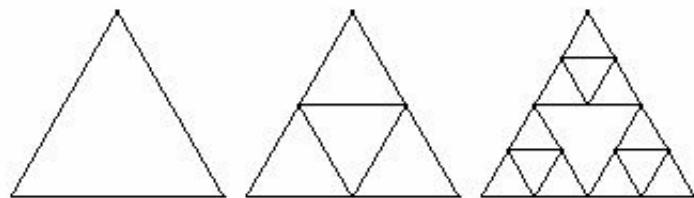


Figura 10.2: Sierpinski: construção

jecto se define em função dele próprio. Na literatura aparecem acrónimos recursivos. Por exemplo, **GNU**, que significa *GNU's Not Unix* ou ainda *GOD* que, no famoso livro de Douglas Hofstadter, significa *GOD Over Djinn*<sup>2</sup>.

E o que é que isso tem que ver com a programação? Bom, já referimos que para resolver um dado problema uma abordagem metodológica interessante é dividi-lo em sub-problemas mais simples cuja solução encontramos. Juntando as soluções parciais chegamos à solução global. E o que acontece quando um (ou mais) desses problemas é **semelhante** ao problema original? Para tornar a discussão mais concreta vamos introduzir um problema muito famoso denominado *Torres de Hanói*. Na figura 10.3 vemos três varas colocadas num suporte. Numa das varas estão três discos (embora possa ser qualquer número) ordenados pelo seu diâmetro. O problema é determinar a sequência de movimentos que me permite deslocar os discos da vara da esquerda para a vara da direita usando a do meio como auxiliar. As restrições são: (1) só podemos deslocar um disco de cada vez e (2) nunca podemos ter uma situação em que um disco está por cima de outro de diâmetro inferior. Convidamos o leitor a resolver o problema. Conseguiu? Tente agora aumentar o número de discos e verá como a tarefa se torna muito, mas mesmo muito, difícil.

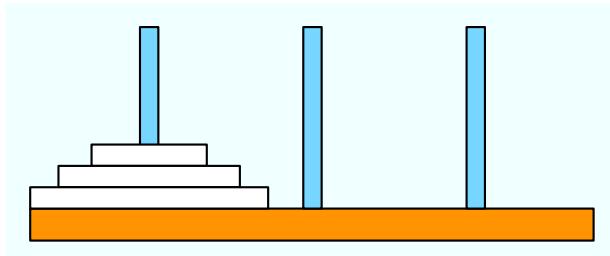


Figura 10.3: Torres de Hanói

Está na hora em pedir ajuda ao computador. Que programa nos pode imprimir a sequência de movimentos? Tente, com os conhecimentos que tem pensar numa solução. Uma vez mais verá que a sua vontade vai ser a de ... desistir. É então que uma nova forma de pensar vem em seu auxílio. Consiste no seguinte. Vamos dividir o problema original em **três** sub problemas. Dois deles são semelhantes ao original: deslocar discos entre as varas de acordo com as regras. Por **semelhante** queremos dizer neste caso que o sub problema consiste em deslocar um número **menor** de discos e as varas cumprem papéis diferentes. A figura 10.4 mostra a situação em que conse-

Semelhança

<sup>2</sup>Como discutiremos mais adiante estas últimas manifestações de recursividade têm um problema importante: nunca nos permitem parar.

guimos deslocar dois discos da vara da esquerda para a do meio respeitando as regras do jogo.

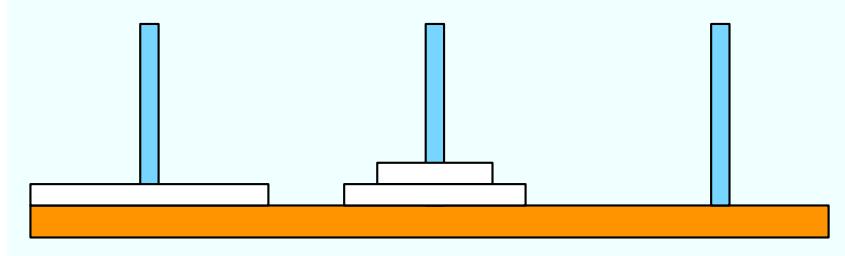


Figura 10.4: Resolução de um sub problema semelhante

Se tivermos conseguido resolver o sub problema do modo indicado,<sup>3</sup> é fácil ver que temos o disco maior liberto para ser deslocado directamente para a sua posição final. Para concluir basta agora resolver o terceiro sub problema: deslocar os dois discos da vara do meio para a vara da direita usando a esquerda como auxiliar. Fazendo isso obtemos a solução que apresentamos na figura 10.5.

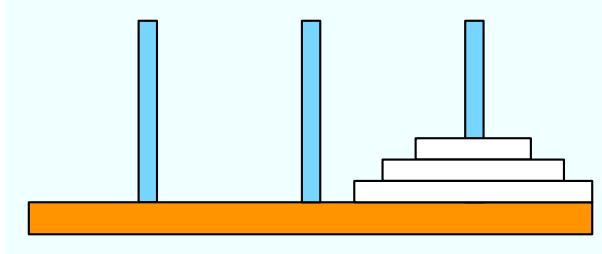


Figura 10.5: Torres de Hanói: solução final

Admitimos que o leitor pode achar a abordagem interessante mas que, naturalmente, coloca a si próprio a questão de saber como é que se resolvem os sub problemas semelhantes. Afinal a nossa solução começa por afirmar a possibilidade de o fazer e é graças a isso que resolvemos o problema. O **passe de mágica** consiste em aplicar a mesma metodologia aos dois sub problemas. Como no *Sierpinski Gasket*. Ou seja, pegamos em cada sub problema que dividimos em três novos sub problemas, dois dos quais serão de novo semelhantes ao inicial. Claro que não podemos fazer isto infinitamente, pelo que terá que haver uma situação em que não necessitamos de

---

<sup>3</sup>Por agora vamos admitir que sim, embora ainda não saibamos bem como foi isso possível!

decompor o problema em sub problemas pois conseguimos resolvê-lo directamente. Chamamos a essa situação o **caso de base**. Para este exemplo, o caso mais simples, caso de base, acontece quando apenas temos um disco para movimentar. Em resumo: dado um problema ou o conseguimos resolver directamente (caso de base) ou o dividimos em sub problemas, alguns deles semelhantes, a quem aplicamos a mesma metodologia. Esta segunda situação é conhecida por **caso recursivo**. Juntando todas estas peças dispersas chegamos à versão que a listagem 10.1 ilustra.

```

1 def torres_hanoi(n,a,b,c):
2 "Implementação das torres de Hanói"
3 if n == 1: #Caso de Base
4 print ("Move disco %d de %s para %s" % (n,a,c))
5 else: # Caso recursivo
6 torres_hanoi(n-1,a,c,b)
7 print("Move disco %d de %s para %s" % (n,a,c))
8 torres_hanoi(n-1,b,a,c)

```

Listagem 10.1: 'Torres de Hanói'

Notar a ordem das varas nas três situações. A vara na posição inicial é a de partida, a segunda é a auxiliar e a terceira a de chegada. Executando o programa para o caso de três discos obtemos a solução dada na listagem 10.2. Os três primeiros movimentos correspondem ao primeiro sub problema com dois discos, é seguido pela passagem directa do disco maior, o terceiro, da vara esquerda para a vara direita, e conclui-se com os três movimentos gerados pelo último sub problema semelhante.

```

1 Move disco 1 de Esquerda para Direita
2 Move disco 2 de Esquerda para Meio
3 Move disco 1 de Direita para Meio
4 Move disco 3 de Esquerda para Direita
5 Move disco 1 de Meio para Esquerda
6 Move disco 2 de Meio para Direita
7 Move disco 1 de Esquerda para Direita

```

Listagem 10.2: 'Exemplo de sessão'

De notar que a decomposição sucessiva do problema em sub problemas até chegar ao caso de base é tarefa que deixamos ao programa. Igualmente vai ser ele, o programa, responsável por juntar a cada passo as soluções parciais obtidas até podermos construir a solução final.

Pensamos ser agora claro para o leitor o que queremos dizer por recursividade

Na realidade existem pelo menos dois modos de nos referirmos ao conceito: um que privilegia a **estrutura dos objectos** e outra que se refere ao **processo** de resolução de um problema<sup>4</sup>.

- Quando um objecto de define em função dele próprio<sup>5</sup>
- Quando um processo se decompõe em sub processos semelhantes

Vamos agora mostrar vários exemplos de programas recursivos. Não escondemos que muitos deles são triviais, sendo fácil encontrar versões não recursivas, i.e., iterativas, para eles. A sua apresentação segue apenas princípios pedagógicos, ou seja, pretendemos através da apresentação dos exemplos deixar o leitor mais à vontade com o conceito. Com os exemplos também apresentaremos vários **tipos** de recursividade.

## 10.2 Exemplos

### 10.2.1 Números

Um exemplo clássico de recursividade é a função **factorial**. Por definição temos:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases} \quad (10.1)$$

A partir dela construímos o programa 10.3.

```

1 # factorial
2 def fact(n):
3 if n == 0:
4 return 1
5 else:
6 return n * fact(n -1)

```

Listagem 10.3: Factorial

Não pensamos haver muito mais a dizer sobre esta tradução da definição no programa que calcula o factorial de qualquer número natural. Também aqui admitimos que somos capazes de resolver o sub problema semelhante mais simples para o argumento  $(n - 1)$ .

---

<sup>4</sup>Estas duas visões estão inter-relacionados como adiante se verá.

<sup>5</sup>Estamos no domínio das chamadas definições indutivas.

No entanto, importa clarificar como e porque funciona. Os programas recursivos desdobram-se em **duas fases**. Vejamos com um exemplo concreto. Admitamos que queríamos saber a que era igual o factorial de 4. Chamada a função **fact**, com  $n = 4$ , e porque 4 é diferente de 0, o resultado vai ser o que for devolvido pela chamada recursiva de  $fact(n - 1) = fact(3)$  depois de multiplicado por 4. Este processo repete-se até que se chega ao caso de base, quando pretendemos saber o valor do factorial de 0 e este é dado directamente. Assim termina a primeira fase, denominada **desenrolar**<sup>6</sup>. A [Desenrolar, Enrolar](#)

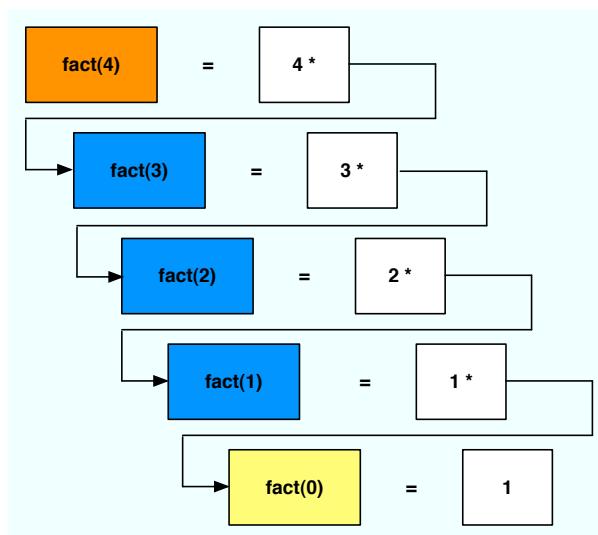


Figura 10.6: Factorial: fase de desenrolar

Como já foi referido, e a figura 10.6 ilustra, há um conjunto de cálculos que ficaram em suspenso e agora precisam ser concluídos. Entramos na segunda fase do processo, denominada **enrolar**<sup>7</sup>. Nesta fase vão sendo passados para *cima*, i.e., a quem pediu o resultado, e os valores calculados. Assim o valor do factorial de 0, calculado directamente, é transmitido à chamada  $fact(1)$  que efectua o produto desse valor por 1, que estava em suspenso. Esse resultado é por sua vez transmitido para  $fact(2)$  que estava à sua espera para multiplicar por 2, e assim sucessivamente. No final, obtemos o valor pretendido, ou seja 24. A figura 10.7, tenta ilustrar o processo.

Antes de prosseguir podemos já fazer uma pequena síntese. A **recursividade** caracteriza-se por:

[Síntese](#)

<sup>6</sup>Do inglês *unfold*.

<sup>7</sup>Do inglês *fold*.

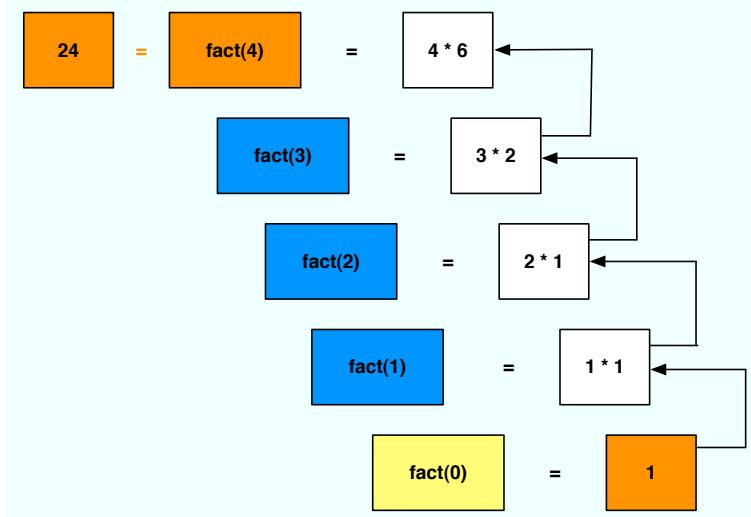


Figura 10.7: Factorial: fase de enrolar

- decompor o problema em sub problemas
- um (ou mais) dos sub problemas podem ser resolvidos directamente (caso(s) de base)
- um (ou mais) dos subproblemas são semelhantes ao problema inicial (caso(s) recursivo(s))
- os problemas semelhantes devem ser tais que nos façam aproximar do(s) caso(s) de base
- um processo de desenrolar, seguido por um de enrolar

Analisemos agora outros problemas com o mesmo grau de dificuldade e que podem ter uma solução recursiva simples. Os **números naturais** são definidos axiomaticamente, de modo indutivo, do seguinte modo:

- (1) 0 é um número natural
- (2) se  $n$  é um número natural então o seu sucessor,  $suc(n)$ , também é um número natural

A partir desta definição podemos construir a sequência de números naturais:  $0, suc(0), suc(suc(0)), \dots$ . Por conveniência designamos  $suc(0)$  por 1,  $suc(suc(0))$  por 2 e assim sucessivamente. Vejamos como podemos construir

uma hierarquia de funções sobre os números naturais.

```

1 def soma(m,n):
2 if n == 0:
3 return m
4 else:
5 return 1 + soma(m,n-1)

```

Listagem 10.4: 'Soma'

Soma

```

1 def prod(m,n):
2 if n == 0:
3 return 0
4 else:
5 return soma(m, prod(m,n-1))

```

Listagem 10.5: 'Produto'

Produto

```

1 def exp(m,n):
2 if n == 0:
3 return 1
4 else:
5 return prod(m, exp(m,n-1))

```

Listagem 10.6: 'Exponencial'

Exponencial

Todos estes casos, incluindo o de factorial, são exemplo de recursividade dita **linear**: por cada ramo da condicional só existe uma chamada recursiva. Mas há outro aspecto interessante nestes exemplos. Todos satisfazem o mesmo modelo abstracto que apresentamos na listagem 10.7<sup>8</sup>.

Recursividade Linear

```

1 def f(n):
2 if n == 0:
3 return <constante>
4 else:
5 return h(g(n),f(n-1))

```

Listagem 10.7: 'Modelo de recursividade linear'

---

<sup>8</sup>O caso da soma também pode ser apresentado na mesma forma alterando o modo como a chamada recursiva é feita, recorrendo à função sucessor Igualmente, o facto de o modelo apenas ter um argumento, contra dois das funções apresentadas é irrelevante.

Sem grandes surpresas o caso de base corresponde ao valor de 0. No caso recursivo aparece uma chamada para um sub problema mais simples (aqui  $n$  dá lugar a  $(n - 1)$ ). Um outro exemplo desta *família* pode ser posto em evidência quando consideramos somatórios como, por exemplo:

$$\sum_{i=0}^n i$$

Este somatório pode ser definido do seguinte modo:

$$\sum_{i=0}^n i = \begin{cases} 0 & n = 0 \\ n + \sum_{i=0}^{n-1} i & n > 0 \end{cases} \quad (10.2)$$

Daqui retiramos imediatamente o programa ilustrado na listagem 10.8.

```

1 def somat(n):
2 if n == 0:
3 return 0
4 else:
5 return n + somat(n-1)

```

Listagem 10.8: 'Somatório'

Existem muitos mais exemplos envolvendo números. Por exemplo, o Algoritmo de Euclides permite-nos calcular o **máximo divisor comum** de dois números naturais. O máximo divisor comum de dois números é o maior inteiro que os divide a ambos. Numa abordagem clássica uma solução simples seria:

1. Calcular os divisores dos dois números
2. Determinar os que são comuns.
3. Escolher o maior dos cumuns.

Um programa trivial seria o indicado na listagem 10.9.

```

1 def mdc(n,m):
2 div_n= divisores(n)
3 div_m= divisores(m)
4 inter=intersect(div_n,div_m)
5 return max(inter)
6
7 def divisores(num):
8 return [i for i in range(1,num+1) if (num % i) == 0]
9

```

```

10 def intersect(l1,l2):
11 return [i for i in l1 if i in l2]

```

Listagem 10.9: 'MDC: iterativo'

O que Euclides fez foi propor uma solução que consiste em ir subtraindo ao maior o menor até que um deles se reduza a 0. O algoritmo pode ser optimizado conforme se apresenta na listagem 10.10.

```

1 def mdc(m,n):
2 "Máximo Divisor Comum: algoritmo de Euclides"
3 if n == 0:
4 return m
5 else:
6 return mdc(n, m % n)

```

Listagem 10.10: 'Algoritmo de Euclides'

Por palavras dizemos que o máximo divisor comum de dois números é o primeiro se o segundo for zero ou então é igual ao máximo divisor comum entre o segundo e o resto da divisão inteira do primeiro pelo segundo. Como se pode ver a estrutura é a mesma: um caso de base ( $n$  igual a 0), e um caso recursivo. Neste último temos uma vez uma vez mais um sub problema semelhante mas de *menor* dimensão. Esta ideia de **redução** da dimensão do problema em cada etapa recursiva é fundamental. Mais, essa redução deve ser tal que nos conduza **sempre** ao caso de base. A não ser assim o programa nunca termina<sup>9</sup>. Este exemplo do **mdc** tem outra singularidade: o caso recursivo só contém a chamada da função não havendo mais nada para fazer além disso. Estes casos são apelidados de recursividade **terminal**<sup>10</sup>.

Recursividade  
Terminal

Actualmente os compiladores modernos são capazes de detectar uma definição recursiva terminal e gerar automaticamente o código iterativo correspondente.

Até agora todos os exemplos numéricos envolveram apenas uma chamada recursiva. À semelhança das Torres de Hanói podemos ter problemas com mais de uma chamada recursiva num mesmo ramo da condicional. Começemos com o célebre exemplo da sequência de Fibonacci<sup>11</sup>. Na sua origem está um problema que pode ser enunciado do seguinte modo. Admitamos

Fibonacci

<sup>9</sup>Na realidade termina por exaustação dos recursos da máquina!

<sup>10</sup>Saliente-se que o importante é que a última acção a executar seja a chamada recursiva. Existem casos de recursividade terminal em que há acções não recursivas antes da chamada terminal.

<sup>11</sup>O nome da sequência deve-se ao matemático italiano Leonardo de Pisa, também conhecido por filho de Bonacci, que viveu na Idade Média

que um casal de coelhos demora dois meses após o nascimento para estar em condições de se reproduzir. Consideremos ainda que quando atinge a idade de reprodução dá origem a um novo casal (macho e fêmea) de coelhos que obedece ao mesmo princípio. A questão é saber quantos casais de coelhos existem no início de cada mês. A figura 10.8 mostra a evolução da população de casais ao longo dos meses. A amarelo a indicação dos coelhos em condição de se reproduzirem.

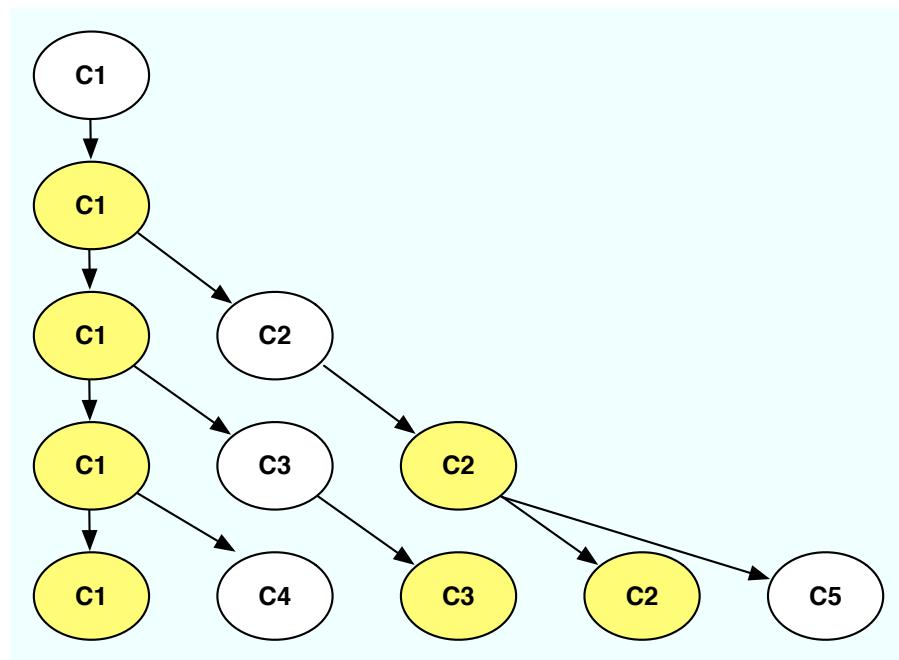


Figura 10.8: Fibonacci: coelhos e reprodução

O número de casais é dado pela sequência 1, 1, 2, 3, 5, ..., e uma observação atenta mostra que num dado mês o número de casais é igual à soma no número de casais existente nos dois meses anteriores. Daí a definição:

$$Fib(n) = \begin{cases} 1 & n = 1 \text{ ou } n = 2 \\ Fib(n - 1) + Fib(n - 2) & n > 2 \end{cases} \quad (10.3)$$

Esta definição tem uma tradução directa no programa da listagem 10.11.

```

1 def fib(n):
2 " Números de fibonacci recursivo"
3 if n == 1 or n == 2:
4 return 1

```

```

5 else:
6 return fib(n-1) + fib(n-2)

```

Listagem 10.11: 'Sequência de Fibonacci'

On números de Fibonacci estão presentes em diversas áreas da matemática (e.g., teoria de números) e na natureza (espirais de ananases, pinhas, girassóis, ...). Existe mesmo uma revista matemática, chamada *Fibonacci Quarterly*, que lhe é dedicada. Também na Internet existem vários sítios sobre os números de Fibonacci. É só googlar! Esta definição recursiva tem alguns aspectos interessantes. Desde logo existem dois valores possíveis para o caso de base. Em relação com esse facto as duas chamadas recursivas envolvem dois sub problemas de dimensão  $(n - 1)$  e  $(n - 2)$ .

Um tipo de recursividade semelhante à dos números de Fibonacci pode ser obtida a partir da definição dos coeficientes do **Binómio de Newton**:

[Binómio de Newton](#)

$$\binom{n}{k} = \begin{cases} 1 & k = 0, k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{caso contrário} \end{cases} \quad (10.4)$$

Daqui decorre trivialmente o programa da listagem 10.12.

```

1 def binomio(n,k):
2 "Coeficientes do binómio"
3 if k ==0 or k == n:
4 return 1
5 else:
6 return binomio(n-1,k) + binomio(n-1,k-1)

```

Listagem 10.12: 'Binómio de Newton'

Curiosamente existe uma relação entre os números de Fibonacci e os coeficientes de binómio. A figura 10.9 ilustra como se podem obter os primeiros à custa dos segundos, por soma dos valores nas diagonais da representação do Binómio de Newton como um triângulo (dito de Pascal).

Estamos perante dois exemplos de recursividade **não linear**: existe mais [Recursividade Não Linear](#)

Vamos agora ver como podemos conjugar definições de tal modo que a recursividade aparece indirectamente. Admitamos que queremos uma definição que nos permita determinar se um dado **número é ou não par**. Uma definição possível é a que nos diz que um número é par se o seu antecessor for ... ímpar. Isto significa que necessitamos de usar a definição de ímpar. Esta, por sua vez pode usar o facto de um número ser ímpar se o seu antecessor for ... par! E parece que estamos num círculo vicioso. Mas não como

[Par e Ímpar](#)

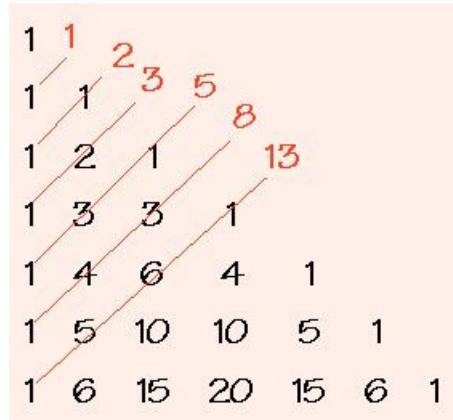


Figura 10.9: Números de Fibonacci e Binómio de Newton

## Recursividade Cruzada

já mostraremos. Para isso precisamos de introduzir o caso de base para as duas definições por forma a que o processo de chamada cruzada entre as duas definições termine. Se pensarmos um pouco chegaremos à seguinte solução:

```

1 # Par
2 def par(n):
3 if n < 0:
4 return False
5 elif n == 0:
6 return True
7 elif impar(n-1):
8 return True
9 else:
10 return False2
11
12 # Ímpar
13 def impar(n):
14 if n < 1:
15 return False
16 elif n == 1:
17 return True
18 elif par(n-1):
19 return True
20 else:
21 return False

```

Listagem 10.13: 'Par - Ímpar'

Chamamos a atenção para as condições de paragem nos dois casos. Podem ser simplificadas? Deixamos ao leitor a reflexão sobre esta questão.

Vejamos um outro exemplo de recursividade cruzada. O problema é o seguinte: dada uma sequência de números queremos determinar se todos os números ou são maiores que os seus vizinhos esquerdo e direito ou (exclusivo) são menores. Exemplo de sequência que satisfaz esta condição: **[0,9,3,7]**. Os elementos nas extremidades só testam uma das condições . Na prática é como se tivéssemos uma sequência de números em que cada um deles, alternadamente, sobe e desce em valor relativamente ao seu antecessor. A ideia para chegar à solução baseia-se na comparação dois a dois dos elementos da sequência. Uma sequência alterna se os seus primeiros dois números estiverem em ordem crescente (decrescente) e o resto da sequência sem o primeiro elemento alternar com início decrescente (crescente). O caso de base é trivial e resume-se à situação de existir apenas um elemento na sequência que manifestamente alterna. Depois destas considerações chegamos à solução apresentada na listagem 10.14.

```
1 def alterna_plus(lst):
2 if len(lst) == 1:
3 return True
4 elif lst[0] > lst[1]:
5 return alterna_minus(lst[1:])
6 else:
7 return False
8
9 def alterna_minus(lst):
10 if len(lst) == 1:
11 return True
12 elif lst[0] < lst[1]:
13 return alterna_plus(lst[1:])
14 else:
15 return False
16
17 def alterna(lst):
18 # a lista não pode ser vazia
19 if len(lst)==1:
20 return True
21 elif lst[0] > lst[1]:
22 return alterna_minus(lst[1:])
23 else:
```

24      **return alterna\_plus(lst[1:])**

Listagem 10.14: 'Sequência Alternada'

Chamamos a atenção para a necessidade de definir se no início a sequência cresce ou decresce. Note-se ainda os dois casos de base.

### 10.2.2 Sequências

#### Procura Simples

Sequências são colecções ordenadas de objectos. Por definição uma sequência pode não ter elementos, dizendo-se **vazia**. A sequência vazia é o equivalente ao 0 dos números naturais. Comecemos com um exemplo simples: saber se um elemento faz parte de uma sequência. A resposta que pretendemos é de tipo booleano: *True* ou *False*. Vamos usar o que sabemos sobre a estrutura das sequências para resolver o problema. Claro que vamos esquecer que o problema em Python se podia resolver facilmente com o teste **elem in seq**. Este exemplo tem apenas finalidade pedagógica. Aqui o caso de base é um pouco mais complexo. Se a sequência estiver vazia é óbvio que o elemento não está na sequência (*False*). Se tiver elementos e o primeiro for igual ao que procuramos então a resposta é afirmativa (*True*). Se nada disto for verdadeiro temos que continuar a procurar na sequência, mas agora sem o primeiro elemento, pois este já foi testado! Daí o programa da listagem 10.15

```
1 def procura_s(elem, seq):
2 if len(seq) == 0:
3 return False
4 elif seq[0] == elem:
5 return True
6 else:
7 return procura_s(elem, seq[1:])
```

Listagem 10.15: 'Procura Simples'

#### Capicua

Uma **capicua** (também chamada de palíndrome) é uma sequência que é igual lida da esquerda para a direita ou da direita para a esquerda. Há muitas formas de determinar se uma dada sequência é ou não capicua. Vamos pensar numa solução recursiva. A chave para encontrar rapidamente a solução está no modo como decomponemos o problema em sub problemas. Essa decomposição depende da estrutura do objecto e nas propriedades que decorrem da definição. Assim sabemos que uma sequência vazia é claramente uma capicua. O mesmo sucede se a sequência tiver apenas um elemento. Daqui retiramos o caso de base pois temos resposta directa para a questão. Mas, e se tiver mais do que um elemento? Bom neste caso será capicua se os elementos nos extremos da sequência forem iguais e se o resto da capicua sem

estes elementos for uma ... capicua!. A listagem 10.16 mostra o respectivo programa.

```

1 def capicua(seq):
2 if (len(seq)== 0) or (len(seq)== 1):
3 return True
4 elif seq[0] == seq [-1]:
5 return capicua(seq[1:-1])
6 else:
7 return False

```

Listagem 10.16: 'Capicua'

Todos sabemos que é fácil inverter uma sequência em Python. Basta fazer [Inversão seq\[::-1\]](#). Mas nem todas as linguagens são como Python! Então vamos ver como poderíamos resolver a questão de um modo geral. Vamos **pensar recursivo**: decompor o problema em sub problemas alguns semelhantes ao problema inicial mas de menor dimensão. Admitamos que *alguém* consegue resolver o problema de inverter a sequência **sem** o primeiro elemento. Então basta pegar nesta solução parcial e acrescentar no **final** o primeiro elemento. A figura 10.10 mostra a ideia.

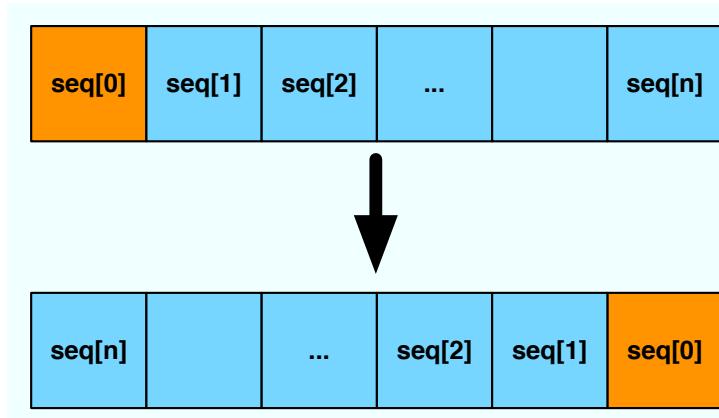


Figura 10.10: Inverter uma sequência

O programa correspondente é dado na listagem 10.17.

```

1 def inverte(seq):
2 if len(seq) == 0:
3 return seq
4 else:

```

```
5 return inverte(seq[1:]) + seq[0]
```

Listagem 10.17: 'Inverte'

Tal como está definido o programa apenas pode ser usado para cadeia de caracteres. Porquê? Que modificações são necessárias para que funcione para, por exemplo, listas?

### Decomposição

Já por várias vezes pusemos em relevo a necessidade de decompor um problema em sub problemas semelhantes. Mas haverá apenas uma maneira de o fazer? Consideremos o exemplo anterior da inversão de uma sequência. Posso considerar uma decomposição em que admito ser capaz de inverter a sequência do primeiro ao penúltimo elemento ficando depois apenas por colocar na posição apropriada o último elemento que, neste exemplo, passará a ser o primeiro. A figura 10.11 ilustra o processo e a listagem 10.18 apresenta o programa.

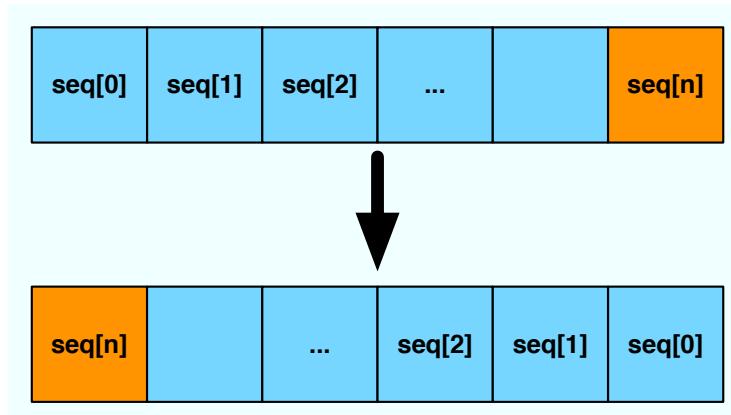


Figura 10.11: Inversão: alternativa

```
1 def inverte2(seq):
2 if len(seq) == 0:
3 return seq
4 else:
5 return seq[-1] + inverte2(seq[:-1])
```

Listagem 10.18: 'Inverte: alternativa'

Mas podemos pensar noutras decomposições. Por exemplo dividir a sequência ao meio, recursivamente inverter cada metade e depois juntar pela ordem correcta. O respectivo programa é apresentado na listagem 10.19. No-

tar a mudança na condição de paragem (caso de base). Consegue perceber o porquê da mudança?

```

1 def inverte3(seq):
2 if len(seq) == 1:
3 return seq
4 else:
5 meio= len(seq)//2
6 return inverte3(seq[meio:]) + inverte3(seq[:meio])

```

Listagem 10.19: 'Inverte: mais uma alternativa'

Havendo então várias possibilidades de decomposição por qual optar? Não há uma resposta simples para esta questão. Do ponto de vista da **correcção** do programa o que precisamos garantir é que:

- todos os casos de base são cobertos
- os sub problemas semelhantes aproximam-nos e convergem para o caso de base
- da articulação dos sub problemas semelhantes com os outros sub problemas obtém-se a solução final

No entanto, também devemos ter em atenção questões como a eficiência e a elegância.

Suponhamos que pretendemos um programa que dado um número  $n$  imprime a sequência descendente de  $n$  até 1 seguida da mesma sequência em modo ascendente. Por exemplo, se  $n = 7$  o resultado deverá ser  $[7, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 7]$ . Vamos então **pensar recursivamente**. Vamos admitir que, para um dado  $n$ , conseguimos mostrar a sequência ascendente e descendente para valores entre  $(n - 1)$  e 1. Então para completar o processo basta agora juntar no início e no fim o valor  $n$ . Qual o caso de base? Fácil, quando  $n = 1$ . Como vemos efectuamos uma decomposição com um sub problema semelhante e que se aproxima a cada chamada do caso de base. A figura 10.12 ilustra a ideia e o programa 10.20 mostra a solução.

```

1 def sobe_e_desce(n):
2 if n == 1:
3 return [1,1]
4 else:
5 return [n] + sobe_e_desce(n-1) + [n]

```

Listagem 10.20: 'Sobe e Desce'

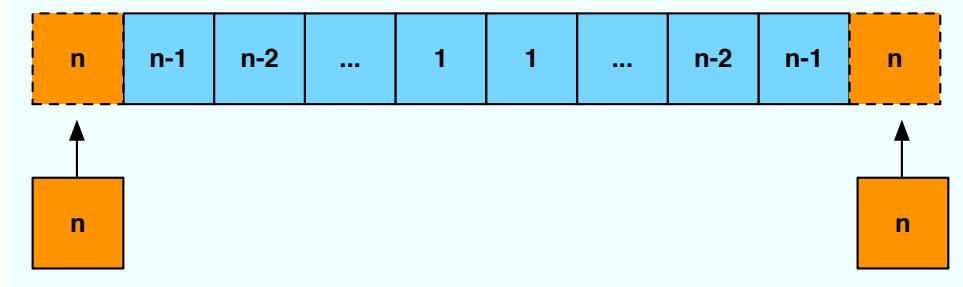


Figura 10.12: Sobe e Desce

Suponhamos que temos duas sequências de comprimento eventualmente [Intercalar](#) distinto. Admitamos que queremos criar uma nova sequência em que os seus elementos são obtidos intercalando os elementos das duas sequências primitivas. Se uma for maior do que a outra acrescentando no final os elementos em excesso. Suponhamos (pensamento recursivo!) que conseguimos intercalar as duas sequências sem os respectivos primeiros elementos. Trata-se de um problema semelhante e de menor dimensão. Sabemos resolver directamente algum caso? Sim! Quando uma das sequências não tiver elementos o resultado é simplesmente a outra sequência. Daí o programa que a listagem 10.21 apresenta.

```

1 def inter(l1,l2):
2 if l1 == []:
3 return l2
4 elif l2 == []:
5 return l1
6 else:
7 return [l1[0],l2[0]] + inter(l1[1:],l2[1:])

```

Listagem 10.21: Alternar

Acreditamos, com o leitor, que este não é um problema muito difícil. Afinal só nos era pedido para alternar os elementos começando sempre com um elemento de uma delas. Tentemos então complicar a situação. Queremos agora intercalar os elementos das duas sequências de todos os modos possíveis. A única restrição é que a ordem que eles têm nas sequências primitivas seja respeitada. Por exemplo:

```

1 >>> inter_all([1,2],[3,4])
2 [[1, 2, 3, 4], [1, 3, 2, 4], [1, 3, 4, 2], [3, 1, 2, 4], [3,
 1, 4, 2], [3, 4, 1, 2]]

```

Antes de olhar para a solução que apresentamos na listagem 10.22 pense no problema e tente encontrar a **sua** solução. Como nos pode ajudar aqui o pensamento recursivo? Comecemos por separar nas duas situações que resultam de começar pelo primeiro elemento de cada uma delas. Admitamos agora que conseguimos juntar todas as soluções para cada um dos dois casos enumeraçados, mas sem os respectivos primeiros elementos. Problema semelhante e menor! Como chegar à solução final? Bem, basta colocar o elemento em falta no seu sítio, isto é, no início, de **todas** as soluções parciais. Caso de base: quando uma das sequências estiver vazia.

```

1 def inter_all(l1,l2):
2 if l1 == []:
3 return [l2]
4 elif l2 == []:
5 return [l1]
6 else:
7 aux1= [[l1[0]] + temp for temp in inter_all(l1[1:], l2)]
8 aux2= [[l2[0]] + temp for temp in inter_all(l1, l2[1:])]
9 return aux1 + aux2

```

Listagem 10.22: Intercalar

Se não *gosta* de listas por compreensão<sup>12</sup>, a listagem 10.23 apresenta uma variante. Aqui usamos um terceiro argumento como auxiliar para guardar os resultados parciais.

```

1 def inter_all(l1,l2,aux=[]):
2 if l1 == []:
3 return [aux + l2]
4 if l2 == []:
5 return [aux + l1]
6 return inter_all(l1[1:], l2, aux+[l1[0]]) + inter_all(l1, l2
 [1:], aux + [l2[0]])

```

Listagem 10.23: Intercalar: variante

Quando uma sequência está ordenada (por exemplo de modo ascendente) existe um modo bastante eficiente de determinar se um elemento pertence ou não à sequência. A ideia é testar o elemento do *meio*. Se for idêntico ao que procuramos então o programa deve terminar. Caso seja diferente só existem duas hipóteses: ou o elemento que procuramos é maior, e nesse caso procuramos recursivamente na parte dos elementos maiores, ou o elemento é menor e então procuramos recursivamente na parte dos elementos menores.

[Procura Binária](#)

---

<sup>12</sup>Se não gosta faz mal ...

E qual é o caso de base? Quando, por exemplo a sequência se reduz a um elemento. Admitamos que o resultado do programa é o índice do elemento na sequência, caso exista, ou -1 caso contrário. A partir daqui obtemos o programa da listagem 10.24.

```

1 def procura_bin_rec(x,seq, inicio,fim):
2 """ Procura com base no facto dos elementos estarem
3 ordenados
4 """
5 if inicio == fim:
6 if seq[inicio] == x:
7 return inicio
8 else:
9 return -1
10 else:
11 meio =(inicio + fim)/2
12 if x == seq[meio]:
13 return meio
14 elif x < seq[meio]:
15 return procura_bin_rec(x,seq,inicio,meio-1)
16 else:
17 return procura_bin_rec(x,seq,meio+1,fim)

```

Listagem 10.24: 'Procura Binária'

## Árvores Binárias de Procura

As **árvores binárias de procura** são uma estrutura de dados importante em computação. São um caso especial de árvore binária (AB). Estas são estruturas que podemos definir indutivamente do seguinte modo<sup>13</sup>:

- (1) [] é uma árvore binária
- (2) se  $AB_1$  e  $AB_2$  são árvores binárias e  $elem$  é um elemento, apelidado de raiz, então  $[AB_1, elem, AB_2]$  é uma árvore binária.

Uma  $AB = [AB_1, elem, AB_2]$  diz-se de **procura** se e só se:

- 1) os elementos de  $AB_1$  forem todos inferiores a  $elem$  e  $AB_1$  for, ela também, uma AB de procura
- 2) os elementos de  $AB_2$  forem todos superiores a  $elem$  e  $AB_2$  for, ela também, uma AB de procura

O problema que nos propomos resolver recursivamente é o de dada uma lista de elementos construir a correspondente AB de procura. O pensamento

---

<sup>13</sup>Esta definição está feita de modo concreto e por referência a um conjunto de elementos genérico

recursivo torna esta questão muito simples. Se a lista estiver vazia o resultado será uma AB de procura vazia. Se não estiver vazia basta construir uma AB de procura tendo como raiz o primeiro elemento da lista e como AB de procura esquerda aquela que construirmos, recursivamente, com todos os elementos da lista menores do que a raiz, e como AB de procura direita aquela que construirmos, recursivamente, com os elementos da lista que são maiores do que a raiz. É fácil ver que introduzimos problemas semelhantes e menores. A listagem 10.25 mostra a solução.

```

1 def arv_bin_proc(lista):
2 if lista == []:
3 return lista
4 else:
5 pivot = lista[0]
6 esq = [elem for elem in lista if elem < pivot]
7 dir = [elem for elem in lista if elem > pivot]
8 return [arv_bin_proc(esq)] + [pivot] + [arv_bin_proc(dir)]

```

Listagem 10.25: Árvore Binária de Procura

### 10.2.3 Gráficos

Ao longo deste texto já recorremos várias vezes ao modulo **turtle**. Vejamos como a recursividade nos pode ajudar a fazer algumas figuras interessantes. Comecemos por uma ideia simples: desenhar um segmento e rodar a tartaruga de um dado ângulo. Repetir a operação para valores diferentes do segmento. A listagem 10.26 mostra o código e a figura 10.13 mostra o desenho que se obtém com a chamada **figura(30,45)**.

[Figuras Recursivas](#)

```

1 def figura(lado, angulo):
2 """Desenha figuras recursivamente a partir de um padrão de
3 base simples.
4 Admite que a tartaruga tem a caneta levantada.
5 """
6 pd()
7 if lado:
8 forward(lado)
9 right(angulo)
10 figura(lado-1, angulo)
11 ht()
12 return 0

```

Listagem 10.26: 'Uma figura simples'

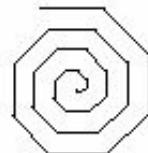


Figura 10.13: Uma figura simples

Qual é o caso de base? Quando o lado fica igual a zero, pois o **if** não é executado! Como é que sei que essa situação acontece? Porque a partir de um dado valor inicial do lado cada chamada recursiva retira uma unidade ao comprimento do lado! E se mudarmos o valor do ângulo? Obtemos logicamente uma figura diferente. Na figura 10.14 mostramos o resultado da chamada **figura(30,75)**.



Figura 10.14: Mudando o ângulo

Vamos alterar o programa por forma a controlar o valor do decremento do lado. O resultado pode ser observado na listagem 10.27. Notar como a condição de paragem foi alterada. Porquê?

```

1 def figura_inc_lado(lado,angulo,inc):
2 "Desenha recursivamente com o incremento como parâmetro"
3 pd()
4 if lado > 0:
5 forward(lado)
6 right(angulo)
7 figura_inc_lado(lado-inc,angulo,inc)

```

```

8 ht()
9 return 0

```

Listagem 10.27: 'Mais uma figura'

Podemos ainda controlar a variação do lado e do ângulo.

```

1 def figura_inc_lado_ang(lado,angulo,incl,inca):
2 "Desenha recursivamente com o incremento como parâmetro"
3 pd()
4 if lado > 0:
5 forward(lado)
6 right(angulo)
7 figura_inc_lado_ang(lado-incl,angulo-inca,incl,inca)
8 ht()
9 return 0

```

Listagem 10.28: 'Ainda outra figura'

Deixamos para o leitor a tarefa de simular o programa com diferentes valores e observar os resultados obtidos. A figura 10.15 ilustra o caso da chamada do programa com `figura_inc_lado_ang(100,120,5,3)`.

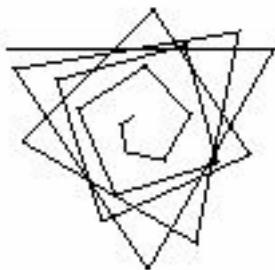


Figura 10.15: Variando o lado e o ângulo

Suponhamos que alguém nos diz que pretende um programa para deseñhar o objecto que vemos na figura 10.16.

A primeira questão que seguramente nos colocamos é a de saber se é **exactamente esta** pirâmide cujo desenho se pretende. Dir-nos-ão que não. A altura da pirâmide pode ser qualquer. Feita esta clarificação o nosso segundo pensamento deverá ser algo como: quais sãos os **dados** do problema? Qual o **resultado** pretendido? O que sei sobre o **domínio** que me possa ajudar a transformar os dados no resultado? É aqui que temos que começar

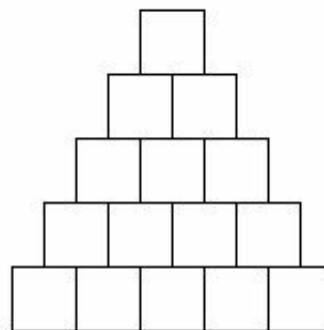


Figura 10.16: Que linda pirâmide

a alinhar umas ideias. Fazer um desenho pode ajudar. Assim a figura 10.17 tenta clarificar a questão.

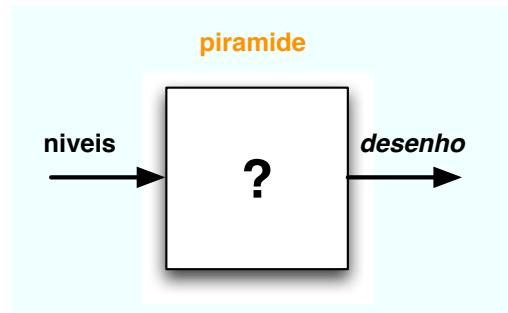


Figura 10.17: Um desenho diz mais do que mil palavras?

Vamos recorrer ao módulo **turtle** para efectuar o desenho. Antes disso há um aspecto importante a sublinhar desde já. O pai da linguística moderna Ferdinand de Saussure dizia que é *o ponto de vista que cria o objecto*. Neste caso podemos olhar para a nossa pirâmide como uma sucessão de linhas. É uma primeira abstração. Por seu turno, cada linha é uma sequência de quadrados. É uma boa prática de desenho separar estes aspectos. Finalmente, vamos **pensar recursivo** (ver figura 10.18)! Saberemos desenhar uma pirâmide com  $n$  níveis se nos facultarem o desenho da pirâmide com  $(n - 1)$  níveis? E se sim, qual o caso de base para a recursividade?

Desta discussão decorre uma solução simples que o programa 10.29 propõe.

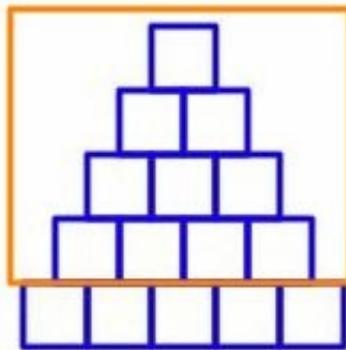


Figura 10.18: Pirâmide: pensar recursivo

```

1 def pir_rec(niveis):
2 if niveis == 1:
3 # desenha
4 linha_rec(1)
5 else:
6 pir_rec(niveis-1)
7 # nova posição
8 pu()
9 setx(xcor() - ((niveis - 2) * lado) - (lado/2))
10 sety(ycor()- lado)
11 pd()
12 linha_rec(niveis)

```

Listagem 10.29: 'As pirâmides'

O programa traduz a ideia de que se só temos uma linha para desenhar tratamos disso directamente, caso contrário desenhamos recursivamente uma pirâmide com  $(n - 1)$  níveis e depois terminamos acrescentando (desenhando) a última linha. O código incorpora adicionalmente os comandos necessários ao posicionamento correcto da tartaruga. A etapa seguinte é o da impressão das linhas. Como o nome da função indica tal pode ser feito também recursivamente. Afinal desenhar  $n$  quadrados é fácil se tivermos *alguém* que nos ajude desenhando  $(n - 1)$  quadrados. Do último nós tratamos directamente! Usando então o mesmo princípio de decomposição obtemos o programa da listagem 10.30.

```

1 def linha_rec(n):
2 if n == 1:

```

```

3 # desenha quadrado
4 quadrado()
5 else:
6 linha_rec(n-1)
7 # vai para o local certo
8 pu()
9 setx(xcor() + lado)
10 pd()
11 quadrado()

```

Listagem 10.30: 'As linhas'

Também aqui foi preciso incorporar comandos de controlo de posicionamento da tartaruga. Para concluir basta concretizar como se desenha um quadrado o que mostramos na listagem 10.31.

```

1 def quadrado():
2 # desenha
3 r=randint(0,255)
4 g=randint(0,255)
5 b=randint(0,255)
6 fillcolor(r,g,b)
7 fill(True)
8 for i in range(4):
9 fd(20)
10 rt(90)
11 fill(False)
12 return True

```

Listagem 10.31: 'Quadrados'

O código consiste essencialmente em repetir quatro vezes o desenho de um segmento e uma rotação de noventa graus. Na solução apresentada não resistimos, no entanto, a incorporar alguma cor! A figura 10.19 mostra um exemplo de resultado quando o lado vale 20 unidades<sup>14</sup>.

### 10.3 Exemplos Complementares

#### Anagramas e Permutações

Diz-se que duas palavras são anagramas quando são compostas exactamente pelos mesmos caracteres independentemente da ordem. Por exemplo, (**roma**, **amor**) ou (**lapa** , **pala**) são anagramas. Pretende-se um programa que dada uma palavra gera todos os anagramas possíveis dessa palavra. Como

<sup>14</sup>Este valor pode ser tornado variável, levando a uma pequena modificação no código.

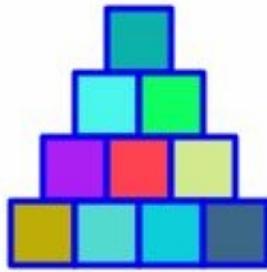


Figura 10.19: Uma pirâmide colorida

obter uma solução recursiva? Vamos decompor o problema nos seguintes sub problemas: gerar os anagramas para a palavra inicial sem o primeiro caractér (sub problema semelhante) para depois inserir o caractér inicial em **todas** as posições possíveis de **todos** os anagramas gerados. É esta solução que apresentamos na listagem 10.32.

```

1 def anagrama(cad):
2 if cad == '':
3 return [cad]
4 else:
5 resp= []
6 for perm in anagrama(cad[1:]):
7 for pos in range(len(perm) + 1):
8 resp.append(perm[:pos] + cad[0] + perm[pos:])
9
return resp

```

Listagem 10.32: 'Anagrama'

Os leitores que gostam de usar listas por compreensão podem chegar a outra solução com a que apresentamos na listagem 10.33.

```

1 def anag(cad):
2 if cad == '':
3 return [cad]
4 else:
5 return [perm[:pos] + cad[0] + perm[pos:] for perm in anag(
cad[1:])] for pos in range(len(perm)+1)]

```

Listagem 10.33: 'Anagrama: variante'

Ao olharmos para este problema facilmente concluímos que ele é na prática o mesmo de gerar todas as permutações para um conjunto de  $n$  elementos. Com as adaptações menores para funcionar com listas, apresentamos na listagem 10.34 o correspondente programa recursivo.

```

1 def permuta(lst):
2 if lst == []:
3 return []
4 else:
5 resp= []
6 for perm in permuta(lst[1:]):
7 for pos in range(len(perm) +1):
8 resp.append(perm[:pos] + [lst[0]] + perm[pos:])
9 return resp

```

Listagem 10.34: 'Permutações'

### Ordenamento

O problema do ordenamento de uma sequência é um problemas mais correntes em programação. Existem vários modos de o fazer e a análise das vantagens e inconvenientes das diferentes propostas revelou-se uma tarefa que além de importante foi estimulante do ponto de vista matemático. Neste momento vamos olhar para duas soluções recursivas para este problema.

### Ordenamento por Fusão

O ordenamento por fusão baseia-se num princípio semelhante ao da procura binária: dividir a sequência ao meio, ordenar cada uma das partes recursivamente e depois fundir as duas sub sequências de modo a obter a sequência original ordenada (ver figura 10.20). O caso de base ocorre quando as sequências têm tamanho unitário, situação em que estão forçosamente ordenadas. Notar o uso da função `deepcopy`.

```

1 def ordena_fusao(seq,esquerda,direita):
2 """Dividir a sequência ao meio. Ordenar cada uma das partes
 separadamente.
3 Depois fundir as duas partes"""
4 if esquerda == direita:
5 return [seq[esquerda]]
6 else:
7 seq=deepcopy(seq)
8 meio = (esquerda + direita)//2
9 seq_esq=ordena_fusao(seq,esquerda,meio)
10 seq_dir=ordena_fusao(seq,meio+1,direita)
11 return fusao(seq_esq,seq_dir)
12

```

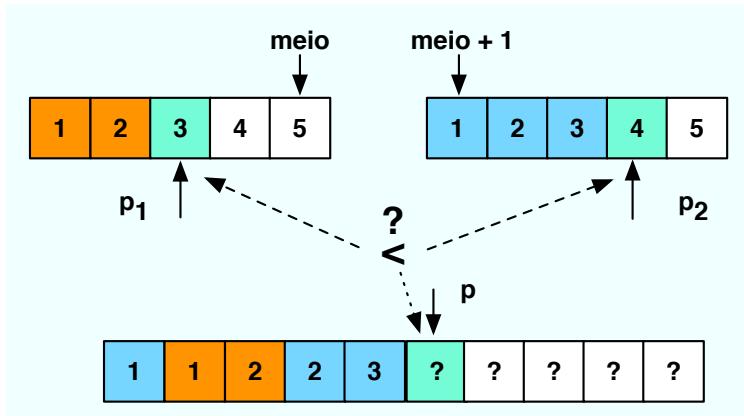


Figura 10.20: Ordenamento por Fusão

```

13 def fusao(seq1,seq2):
14 seq=[]
15 p_1=0
16 p_2=0
17 while (p_1 < len(seq1)) and (p_2 < len(seq2)):
18 if seq1[p_1] < seq2[p_2]:
19 seq.append(seq1[p_1])
20 p_1 = p_1 + 1
21 else:
22 seq.append(seq2[p_2])
23 p_2 = p_2 + 1
24 if p_1 == len(seq1):
25 seq.extend(seq2[p_2:])
26 else:
27 seq.extend(seq1[p_1:])
28 return seq

```

Listagem 10.35: 'Ordenamento por Fusão'

O ordenamento rápido<sup>15</sup> deve o seu nome a ser, em certas circunstâncias, o método de ordenamento mais rápido. O seu princípio é o seguinte. Escolhe-se um seu elemento para pivot e divide-se a sequência inicial em duas subsequências ficando numa, à direita, todos os elementos maiores do que o pivot e na outra, à esquerda todos os elementos menores do que o pivot. Este último vai ficar na sua posição definitiva. Este processo é repetido recursivamente sobre cada uma das duas subsequências (ver figura 10.21). Ao contrário do

[Ordenamento Rápido](#)

<sup>15</sup>Em inglês *quicksort*.

ordenamento por fusão aqui o ordenamento é **feito no lugar**, não sendo por isso necessário proceder a cópias da sequência.

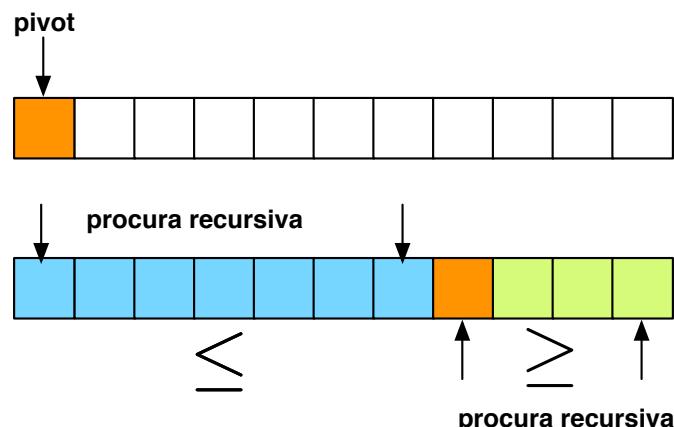


Figura 10.21: Ordenamento Rápido

```

1 def quicksort(seq,inicio,fim):
2 """Quick Sort"""
3 if inicio < fim:
4 divisor= particao(seq, inicio, fim)
5 quicksort(seq, inicio, divisor)
6 quicksort(seq, divisor+1, fim)
7 return seq
8
9
10 def particao(lista, esquerda, direita):
11 """Divide a lista em duas metades em torno de um elemento (pivot).
12 no final todos os elementos menores (maiores) ou iguais ao pivot estão à
13 esquerda (direita) da lista. Devolve o índice que separa a
14 parte esquerda da direita"""
15 pivot=lista[esquerda]
16 p_esq=esquerda
17 p_dir=direita
18 while True:
19 while lista[p_dir] > pivot:
20 p_dir = p_dir - 1

```

```

20 while lista[p_esq] < pivot:
21 p_esq = p_esq + 1
22 if p_esq < p_dir:
23 lista[p_esq],lista[p_dir] = lista[p_dir],lista[p_esq]
24 p_esq=p_esq + 1
25 p_dir=p_dir - 1
26 else:
27 return p_dir

```

Listagem 10.36: 'Ordenamento Rápido'

Fractais, teoria do caos, sistemas complexos são assuntos científicos de interesse crescente. No início deste texto referimos o *Sierpinski Gasket* e mostrámos a respectiva imagem(ver figura 10.1). Vamos mostrar como podemos construir um simulador recursivo usando o módulo **turtle**. Da imagem retiramos que a figura pode ser decomposta em três sub problemas semelhantes. O mais difícil parece ser gerir a **localização** de cada um dos três triângulos que resultam da decomposição. A recursividade trata disso para nós! O caso de base é por nós controlado através do parâmetro nível: quando for zero, termina. O caso recursivo é tratado dentro de um ciclo que é repetido três vezes. Se nos abstrairmos da chamada recursiva esse ciclo desenha um ... triângulo!

```

1 def sierpinsk(tamanho,nivel):
2 if nivel == 0:
3 return True
4 else:
5 for i in range(3):
6 sierpinsk(tamanho/2,nivel -1)
7 fd(tamanho)
8 rt(120)

```

Listagem 10.37: 'Sierpinski Gasket'

Na figura 10.22 pode ver-se o resultado de executar o programa com lado 200 e nível 4.

## 10.4 Quando usar?

A recursividade conduz a soluções simples, legíveis e elegantes. O seu uso depende fundamentalmente da estrutura dos objectos e da natureza do problema. Mas também temos que nos preocupar com os seus custos em espaço e em tempo. Com efeito cada vez que há uma chamada recursiva é necessário guardar o contexto do programa para mais tarde ser possível refazer

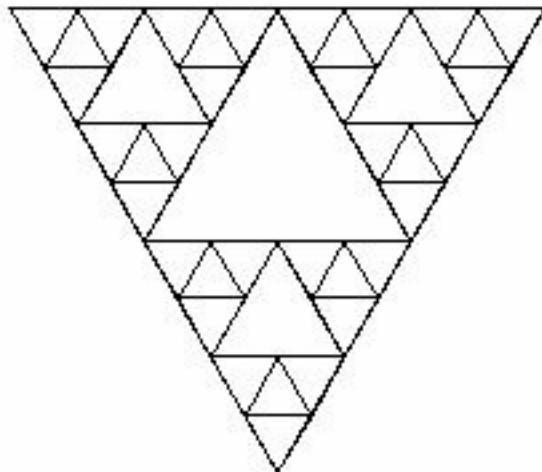


Figura 10.22: Sierpinskii: 200,4

os cálculos. Com objectos de grande dimensão isto pode tornar-se bastante pesado. Como critério de escolha diríamos, em primeiro lugar, a dificuldade em encontrar uma solução iterativa simples (veja-se por exemplo o caso das Torres de Hanói). Depois de encontrar uma solução recursiva a decisão seguinte passa por saber se se justifica a sua transformação ou não numa versão iterativa. Como já referimos certas versões recursivas (as terminais) são já transformadas automaticamente nas correspondentes versões iterativas.

### Memorização

Existem também comandos adicionais com que se podem anotar as soluções recursivas de modo a torná-las mais eficientes evitando o refazer de muitos cálculos. Por exemplo, quando calculamos o número de fibonacci de ordem 4, os cálculos recursivos obrigam a computar  $\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$ . Por seu turno  $\text{fib}(3)$  determina o cálculo de  $\text{fib}(2)$  cujo terá que ser recalculado na segunda chamada recursiva de  $\text{fib}(4)$ . A figura 10.23 ilustra o problema.

Para nos ajudar a resolver esta questão vamos usar uma técnica conhecida por memorização<sup>16</sup>. A ideia consiste em guardar os cálculos já efectuados e cada vez que precisamos de calcular um valor consultamos primeiro a nossa memória antes de efectuar explicitamente os cálculos. O acesso à memória deve ser eficiente sob pena de, a não ser assim, se perder no acesso o que se ganha por não ter que (re)fazer os cálculos. Para tal vamos recorrer a um dicionário: a chave será  $n$  e o valor  $\text{fib}(n)$ . Vejamos como fica a nossa nova versão de Fibonacci.

---

<sup>16</sup>Esta ideia é usada de modo geral em programação dinâmica

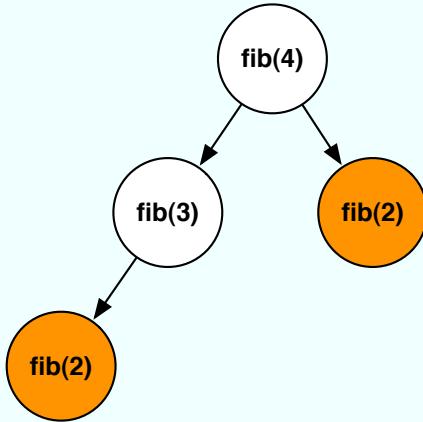


Figura 10.23: Fibonacci: cálculos duplicados

```

1 def fibonacci(n, res={}):
2 if n == 1 or n == 2:
3 return 1
4 else:
5 if n in res:
6 return res[n]
7 else:
8 fib_n = fibonacci(n-1, res) + fibonacci(n-2, res)
9 res[n] = fib_n
10 return fib_n

```

Finalmente, podemos tentar fazer a transformação de modo **manual**. Está fora do alcance deste texto introdutório apresentar a teoria que valida estas transformações. Assim iremos apresentar um exemplo simples que mostra como nos podemos substituir ao compilador. A ideia central passa pela introdução de um mecanismo de **pilha**<sup>17</sup>. A pilha será usada para guardar os diferentes contextos das chamadas recursivas (fase de enrolar), para mais tarde os podermos ir aí buscar e efectuar os cálculos pendentes (fase de desenrolar).

[Transformação Recursivo Iterativo](#)

Retomemos então o exemplo do cálculo do factorial (ver listagem 10.38).

```

1 def fact_rec(n):

```

---

<sup>17</sup>No capítulo ... falaremos com mais rigor sobre pilhas e outros tipos de dado. Por agora basta que o leitor perceba que uma pilha é um mecanismo que pode ser implementado através de uma lista na qual os elementos são introduzidos e retirados da mesma *extremidade*.

```

2 if n == 0:
3 return 1
4 else:
5 return x * fact_rec(n - 1)

```

Listagem 10.38: De novo o factorial

Como referimos, para remover a recursividade precisamos implementar um mecanismo de pilha. Antecipando o que trataremos mais à frente, vamos definir uma **classe** pilha, conforme ilustra a listagem 10.39.

```

1 class Stack:
2
3 # Construtor
4 def __init__(self):
5 self.stack = []
6
7 def push(self,object):
8 self.stack.append(object)
9
10 def pop(self):
11 if len(self.stack) == 0:
12 raise 'Error', 'stack is empty'
13 obj = self.stack[-1]
14 del self.stack[-1]
15 return obj
16
17 def isempty(self):
18 if len(self.stack) == 0:
19 return True
20 else:
21 return False
22
23 def top(self):
24 return self.stack[-1]
25
26 def show(self):
27 print self.stack

```

Listagem 10.39: Tipo de Dados Pilha

A fazermos o comando `pilha=Stack()` associamos ao nome *pilha* um objecto do tipo *Stack*<sup>18</sup>. A construção da solução passa por substituir cada

<sup>18</sup>Em termos concretos criamos um objecto de valor lista vazia.

chamada recursiva pela salvaguarda na pilha do contexto (variáveis locais, parâmetros,...) e pela actualizações das variáveis. Numa segunda fase os cálculos em suspenso são realizados por consulta do contexto guardado na pilha. A listagem 10.40 mostra o resultado do processo.

```

1 def fact_it(n):
2 stack=Stack()
3 factorial = 1;
4 # desenrolar
5 while n > 0:
6 stack.push(n)
7 n = n -1
8 # enrolar
9 while not stack.isEmpty():
10 factorial = factorial * stack.pop()
11
12 return factorial

```

Listagem 10.40: Factorial Iterativo

Para concluir podemos ainda dizer que há também uma questão de estilo pessoal na opção pelo uso da recursividade. Ao leitor a decisão final, que se espera seja fundamentada!

## Sumário

Neste capítulo introduzimos através de vários exemplos o conceito de definições recursivas. Um programa diz-se recursivo se, directa ou indirectamente, se chama a si próprio. As situações recursivas aparecem quando estamos a decompor um problema em sub problemas e damos origem a alguns sub problemas semelhantes ao original. Para poder funcionar as soluções recursivas recorrem a casos de base que são resolvidos directamente, e a casos recursivos. Estes últimos devem ser tais que façam convergir as sucessivas chamadas recursivas para os casos de base. Existem vários tipos de recursividade designadas por terminal, linear ou cruzada. Existe um custo computacional inerente às definições recursivas sendo no entanto nalguns casos a melhor solução possível para um problema. Existem situações recursivas que podem ser optimizadas seja pelo compilador seja por anotações no próprio código.

## Teste os seus conhecimentos

Procure determinar o seu de conhecimento dos seguintes conceitos.

- O que é uma definição recursiva.
- Que tipos de recursividade conhece.
- Quando se deve recorrer à recursividade.
- Como se pode transformar um programa recursivo.
- Em que consiste a técnica da memorização.

## Exercícios

**Exercício 10.1** Todos sabemos que as linguagens de programação têm um operador que permite calcular o resto da divisão inteira de dois números. Admitindo que os números são positivos ou nulos implemente um versão recursiva para o problema.

**Exercício 10.2** Existe um modo alternativo de calcular uma exponencial que se baseia na identidade:

$$x^n = \begin{cases} x^{n/2} \times x^{n/2} & \text{se } x \text{ for par} \\ x^{n/2} \times x^{n/2} \times x & \text{se } x \text{ for ímpar} \end{cases}$$

Com base nesta identidade defina uma solução recursiva para o cálculo da exponencial. Tenha em atenção problemas de eficiência computacional evitando a duplicação de cálculos.

**Exercício 10.3** Escreva um programa que permite eliminar de uma cadeia de caracteres os casos de caracteres repetidos em posições consecutivas. Por exemplo:

```

1 >>> print removedup('aabccda')
2 abcda

```

**Exercício 10.4** Escreva um programa recursivo que dado um elemento e uma árvore binária de procura indique se o elemento pertence ou não à árvore. Altere a sua solução por forma a também devolver a sub-árvore que tem esse elemento por raiz.

**Exercício 10.5** Escreva um programa recursivo que permita inserir um elemento numa árvore binária de procura. No final a árvore deve continuar

a ser uma AB de procura. Caso o elemento já exista nada deve feito.

**Exercício 10.6** Os elementos de uma AB de procura podem ser listados de três modos distintos:

- em ordem: quando se listam primeiro os elementos da sub-árvore esquerda, seguido da raíz e terminando com os elementos da sub-árvore direita;
- pré-ordem: primeiro a raíz, depois a sub-árvore esquerda e finalmente a sub-árvore direita
- pós-ordem: primeiro a sub-árvore esquerda, seguida da sub-árvore direita e terminando com a raiz.

Desenvolva os respectivos programas de travessia na versão recursiva.

**Exercício 10.7** Retome o exemplo da listagem 10.28 e faça variar os próprios incrementos. Simule para alguns valores.

**Exercício 10.8** Queremos usar o módulo **turtle** para desenhar uma árvore simples como a indicada na figura 10.24.

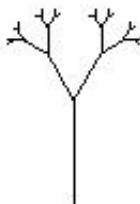


Figura 10.24: Uma árvore recursiva

Desenvolva o respectivo programa recursivo. A figura 10.25 dá uma ideia do processo gerativo.

**Exercício 10.9**

A árvore do exercício 10.4 não é muito *natural*. Uma maneira de desenhar árvores mais interessantes consiste em desequilibrar a sub árvore direita e a esquerda fazendo, por exemplo os ramos de uma maiores do que a outra. A

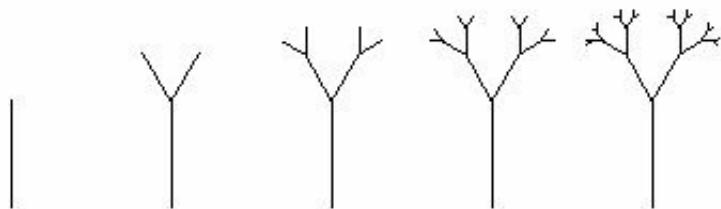


Figura 10.25: O processo

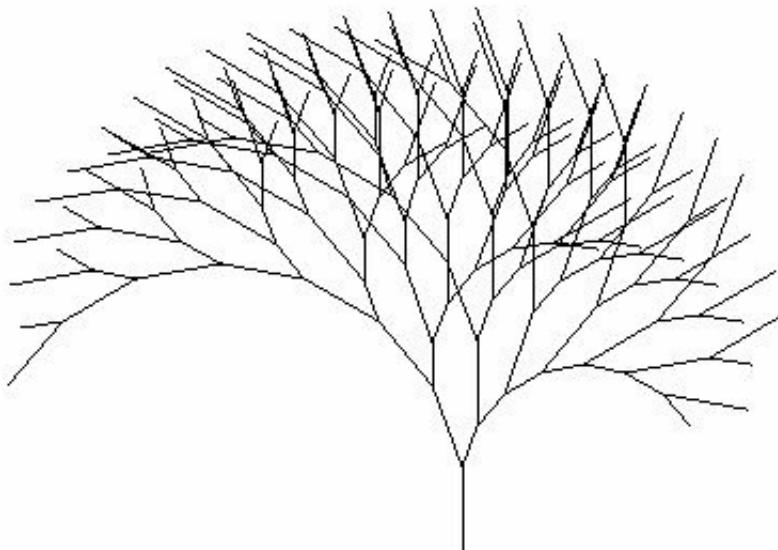


Figura 10.26: Uma árvore mais realista

figura 10.26 ilustra um resultado quando o lado esquerdo é duplo do lado direito. Tente criar o programa recursivo que permite obter estes desenhos.

**Exercício 10.10** Como sabemos uma lista pode ter como elementos listas. Um problema interessante é o de transformar uma lista genérica numa lista plana, isto é, uma lista formada apenas pela sequência dos seus elementos. Exemplo:

```
1 >>> print aplana([[1,2],[[3]],4, [5,[6],7]])
2 [1, 2, 3, 4, 5, 6, 7]
```

Desenvolva o respectivo programa recursivo.

**Exercício 10.11** Escreva um programa recursivo que permita determinar se um dado padrão ocorre ou não num dado texto. O que se lhe oferece dizer sobre os custos computacionais da sua solução?

### Exercício 10.12

Suponhamos que temos uma sequência de  $n$  vectores  $(V_1, V_2, \dots, V_n)$ , todos do mesmo cumprimento. Pretende-se um programa que forneça todos os vectores possíveis de comprimento  $n$  formados combinando ordenadamente os elementos dos vectores  $V_i$ . Exemplo:

```
1 >>> print prod_vectores([[1,2,3],['a','b','c']])
2 [[1, 'a'], [1, 'b'], [1, 'c'], [2, 'a'], [2, 'b'], [2, 'c'],
 [3, 'a'], [3, 'b'], [3, 'c']]
```

**Exercício 10.13** A multiplicação de matrizes é um processo fundamental em muitas áreas da computação. O seu custo computacional, medido em termos do número de multiplicações e adições dos seus elementos, é bastante elevado. Existe um método recursivo de multiplicação de matrizes, conhecido por método de **Strassen**, bastante eficiente. Se tivermos duas matrizes  $n \times n$ ,  $X$  e  $Y$ , e quisermos calcular  $Z = X \times Y$  sabemos que o número de multiplicações necessárias é da ordem de  $\mathcal{O}(n^3)$ . O método de Strassen consiste em dividir **recursivamente** cada uma das duas matrizes em quatro matrizes de dimensão  $n/2 \times n/2$ . Quando temos matrizes  $2 \times 2$  Strassen encontrou um conjunto de formulas que apenas necessitam de 7 multiplicações e 18 somas e/ou adições. Com este algoritmo a complexidade baiuxa para  $\mathcal{O}(n^{2.81})$ . Vejamos como. Consideremos as matrizes  $X$  e  $Y$  e o seu produto  $Z$ .

$$\begin{pmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \times \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix}$$

As fórmulas de Strassen são as seguintes:

$$\begin{aligned} p_1 &= (x_{11} + x_{22}) \times (y_{11} + y_{22}) \\ p_2 &= (x_{21} + x_{22}) \times y_{11} \\ p_3 &= x_{11} \times (y_{12} - y_{22}) \\ p_4 &= x_{22} \times (y_{21} + y_{11}) \\ p_5 &= (x_{11} + x_{12}) \times y_{22} \\ p_6 &= (x_{21} - x_{11}) \times (y_{11} + y_{12}) \\ p_7 &= (x_{12} - x_{22}) \times (y_{21} + y_{22}) \end{aligned}$$

A partir delas podemos computar os elementos da matriz  $Z$ :

$$\begin{aligned} z_{11} &= p_1 + p_4 - p_5 + p_7 \\ z_{12} &= p_3 + p_5 \\ z_{21} &= p_2 + p_4 \\ z_{22} &= p_1 + p_3 - p_2 + p_6 \end{aligned}$$

Para simplificar admita que  $n$  é uma potência de 2, isto é  $n = 2^k$  e implemente o algoritmo.

**Exercício 10.14** Um autómato finito é uma máquina de estados usada em várias aplicações informáticas, como por exemplo para implementar um analisador lexical de um compilador. Um **Autómato finito** pode ser definido matematicamente pelo tuplo:

$$\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$$

no qual,  $Q$  é o conjunto de estados do autómato,  $\Sigma$  é o alfabeto de entrada,  $\delta$  é a função de transição entre estados determinada pela leitura de um símbolo do alfabeto de entrada,  $q_0$  é o estado inicial do autómato e  $F$  é o conjunto de estados finais de  $(M)$ . Notar que  $F \subseteq Q$ . Quando usado como reconhecedor de sequências de caracteres do alfabeto de entrada a máquina é colocada no seu estado inicial e vai transitando entre estados à medida que consome os símbolos da sequência. Se quando consumir todos os símbolos de entrada a máquina se encontrar num dos seus estados finais diz-se que reconheceu a sequência. O que se pretende é uma implementação de um simulador **recursivo** de um autómato finito. O simulador deve ser genérico e não depender de um autómato em particular. A figura 10.27 mostra graficamente um autómato finito que reconhece cadeias de 1 e 0 em que o número de uns é par.

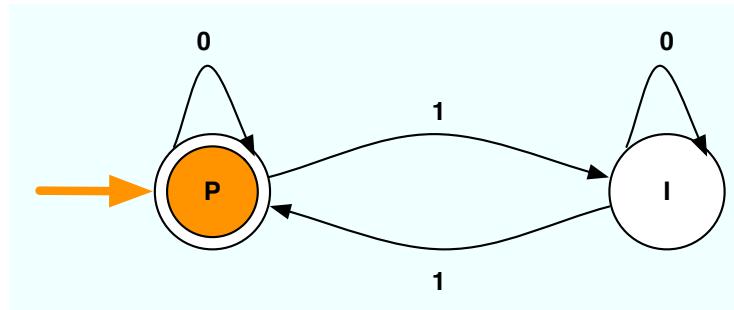


Figura 10.27: Detector de Paridade Par

Para lhe simplificar a vida na listagem 10.41 mostramos como se pode **representar** o autómato indicando explicitamente  $\delta$  (por recurso a um dicionário), o estado inicial  $q_0$  e o conjunto dos estados finais  $F$ .

```

1 transit={'P':{'0':'P','1':'I'}, 'I':{'0':'I','1':'P'}}
2 inicial= 'P'
3 final= ['P']

```

Listagem 10.41: 'Detector de Paridade Par'



## O módulo `turtle`

### A.1 Introdução

O módulo `turtle` permite efectuar desenhos. Baseia-se na ideia de uma **tartaruga** que se passeia sob nosso controlo numa **janela**, podendo deixar um rastro visível. Para isso tem **uma caneta**. Sempre que a caneta está em baixo e a tartaruga caminha, lá aparece o rastro. Existem comandos sobre a tartaruga, comandos sobre a caneta e comandos sobre a janela. As operações estão implementadas em duplicado, num caso, numa lógica de programação procedural (i.e., funções) e, noutro caso, numa lógica orientada aos objectos (i.e., métodos). Sempre que quisermos usar mais do que uma tartaruga temos que recorrer ao modo orientado aos objectos. Quando instala o Python fica com a possibilidade de importar de imediato o módulo `turtle`, pois actualmente o módulo faz parte dos módulos pré-definidos. Vamos descrever os comandos essenciais do módulo, na versão procedural<sup>1</sup>. Uma descrição completa pode ser consultada no manual da linguagem.

### A.2 Tartaruga

#### A.2.1 Movimento e Orientação

Os movimentos básicos podem ser feitos em função da direcção actual da tartaruga, ou em função do ponto de chegada.

---

<sup>1</sup>no caso dos métodos equivalentes apenas tem que acrescentar o parâmetro *self*.

| Nome                                               | Descrição                                                                     |
|----------------------------------------------------|-------------------------------------------------------------------------------|
| <b>forward(unidades)</b><br>ou <b>fd(unidades)</b> | Avança na direcção em que se encontra tantas unidades                         |
| <b>back(unidades)</b> ou<br><b>bk(unidades)</b>    | Recua na direcção em que se encontra tantas unidades                          |
| <b>goto(pos,y=None)</b>                            | Movimenta a tartaruga para a posição indicada                                 |
| <b>setx(x)</b>                                     | Movimenta a tartaruga para a posição de coordenada x. y mantém-se.            |
| <b>sety(y)</b>                                     | Movimenta a tartaruga para a posição de coordenada y. x mantém-se.            |
| <b>speed(veloc=None)</b>                           | Sem argumento, devolve. Com argumento, altera velocidade. Valor entre 0 e 10. |
| <b>delay(atraso = None)</b>                        | Sem nada devolve, com valor altera. Em milisegundos.                          |

Tabela A.1: Movimentos

| Exemplo Movimento |                                                                                                                                     |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------|
|                   | <pre> 1 turtle.forward(50) 2 turtle.goto((50,50)) 3 turtle.back(25) 4 turtle.goto(25,75) 5 turtle.setx(75) 6 turtle.sety(25) </pre> |

| Nome                      | Descrição                      |
|---------------------------|--------------------------------|
| <b>right(angulo)</b>      | Roda para a direita o ângulo.  |
| <b>left(ângulo)</b>       | Roda para a esquerda o ângulo. |
| <b>setheading(ângulo)</b> | Orientação absoluta            |

Tabela A.2: Orientação

**Exemplo Orientação**

```
1 >>> import turtle
2 >>> turtle.heading()
3 0.0
4 >>> turtle.right(45)
5 >>> turtle.heading()
6 315.0
7 >>>
```

### A.2.2 Desenho

| Nome                                 | Descrição                                                               |
|--------------------------------------|-------------------------------------------------------------------------|
| <code>circle(raio,amplitude)</code>  | Desenha um círculo ou um arco de círculo.                               |
| <code>dot(tamanho=None, *cor)</code> | Desenha um ponto na posição em que se encontra.                         |
| <code>stamp()</code>                 | Desenha a forma da tartaruga na posição em que se encontra. Devolve id. |
| <code>clearstamp(stampid)</code>     | Limpa a forma da tartaruga na posição stampid.                          |

Tabela A.3: Desenho

|                             |                                                                                                                   |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------|
| <b>argumentos</b>           | número                                                                                                            |
| <b>argumentos opcionais</b> | número, inteiro                                                                                                   |
| <b>chamada</b>              | <code>circle(número) # círculo completo</code>                                                                    |
|                             | <code>circle(número, número) # arco</code>                                                                        |
|                             | <code>circle(número, número, inteiro)</code>                                                                      |
| <b>Exemplo Círculo</b>      | <pre> 1 &gt;&gt;&gt; turtle.circle(50) 2 &gt;&gt;&gt; turtle.circle(120,180) # semi-círculo 3 &gt;&gt;&gt; </pre> |

### A.2.3 Estado

| Nome                        | Descrição                                                                                   |
|-----------------------------|---------------------------------------------------------------------------------------------|
| <b>position()</b>           | Devolve a posição actual da tartaruga na forma de um tuplo (x,y).                           |
| <b>towards(pos,y=None)</b>  | Devolve o ângulo entre a orientação actual da tartaruga e a posição passada como argumento. |
| <b>xcor()</b>               | Devolve a posição da tartaruga relativa ao eixo dos x.                                      |
| <b>ycor()</b>               | Devolve a posição da tartaruga relativa ao eixo dos y.                                      |
| <b>heading()</b>            | Devolve a orientação actual da tartaruga.                                                   |
| <b>distance(pos,y=None)</b> | Devolve a distância entre a posição actual da tartaruga e a posição passada como parâmetro. |

Tabela A.4: Estado da tartaruga

#### A.2.4 Visibilidade

| Nome                        | Descrição                           |
|-----------------------------|-------------------------------------|
| <b>hideturtle() ou ht()</b> | Esconde a tartaruga.                |
| <b>showturtle()</b>         | Indica se a tartaruga está visível. |
| <b>isvisible()</b>          | Indica se a tartaruga está visível. |

Tabela A.5: Visibilidade

#### A.2.5 Aspecto

| Nome                      | Descrição                                     |
|---------------------------|-----------------------------------------------|
| <b>shape(nome=None)</b>   | Define a forma da tartaruga.                  |
| <b>resizemode(mode)</b>   | Permite alterar dimensões de acordo com mode. |
| <b>shapesize(f1,f2,o)</b> | Altera a forma. depende de resizemode(mode).  |

Tabela A.6: Aspecto da tartaruga

#### A.2.6 Medidas

| Nome                            | Descrição             |
|---------------------------------|-----------------------|
| <b>degrees(circulo = 360.0)</b> | Altera para graus.    |
| <b>radians()</b>                | Altera para radianos. |

Tabela A.7: Medidas

## A.3 Caneta

### A.3.1 Estado

| Nome                  | Descrição                                 |
|-----------------------|-------------------------------------------|
| <b>down() ou pd()</b> | Coloca a caneta em baixo.                 |
| <b>up() ou pu()</b>   | Coloca a caneta no ar.                    |
| <b>pen()</b>          | Devolve ou define os atributos da caneta. |
| <b>pensize(n)</b>     | Define a dimensão da caneta.              |
| <b>isdown()</b>       | diz se a caneta está em baixo.            |

Tabela A.8: Estado da caneta

### A.3.2 Controlo da cor

|                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>argumentos</b>           | diversos tipos                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <b>argumentos opcionais</b> | número, inteiro                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>chamada</b>              | <pre>pen() # consulta</pre> <p>pen(fillcolor='black', pencolor='red', pensize=10) # define alguns atributos para todas as canetas</p> <p>tarta.color('yellow') # Define atributo para uma caneta específica</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Exemplo</b>              | <pre> 1 &gt;&gt;&gt; turtle.pen(fillcolor="black", pencolor="red", pensize=10) 2 &gt;&gt;&gt; sorted(turtle.pen().items()) 3 [('fillcolor', 'black'), ('outline', 1), ('pencolor', 'red'), 4 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'), 5 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)] 6 &gt;&gt;&gt; penstate=turtle.pen() 7 &gt;&gt;&gt; turtle.color("yellow", "") 8 &gt;&gt;&gt; turtle.penup() 9 &gt;&gt;&gt; sorted(turtle.pen().items()) 10 [('fillcolor', ''), ('outline', 1), ('pencolor', 'yellow'), 11 ('pendown', False), ('pensize', 10), ('resizemode', 'noresize'), 12 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)] 13 &gt;&gt;&gt; turtle.pen(penstate, fillcolor="green") 14 &gt;&gt;&gt; sorted(turtle.pen().items()) 15 [('fillcolor', 'green'), ('outline', 1), ('pencolor', 'red'), 16 ('pendown', True), ('pensize', 10), ('resizemode', 'noresize'), 17 ('shown', True), ('speed', 9), ('stretchfactor', (1, 1)), ('tilt', 0)]</pre> |

| Nome             | Descrição                                                     |
|------------------|---------------------------------------------------------------|
| pencolor(*args)  | Altera a cor da caneta.                                       |
| fillcolor(*args) | Define a cor de preenchimento.                                |
| color(*args)     | Sem parâmetros devolve as cores da caneta e de preenchimento. |

Tabela A.9: Cor

### A.3.3 Preenchimento

| Nome                           | Descrição                               |
|--------------------------------|-----------------------------------------|
| begin_fill()                   | A chamar para activar preenchimento.    |
| end_fill()                     | A chamar para desactivar preenchimento. |
| filling()                      | Devolve se está a preencher ou não.     |
| write(text, move, align, font) | Escreve text na posição da tartaruga.   |

Tabela A.10: Preenchimento

| Exemplo | Preenchimento                                                                                                                                                                                    |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|         | <pre> 1 &gt;&gt;&gt; turtle.begin_fill() 2 &gt;&gt;&gt; for i in range(4): 3 ...     turtle.forward(100) 4 ...     turtle.right(90) 5 ... 6 &gt;&gt;&gt; turtle.end_fill() 7 &gt;&gt;&gt; </pre> |

## A.4 Janela

| Nome                             | Descrição                               |
|----------------------------------|-----------------------------------------|
| <code>bye()</code>               | Fecha janela.                           |
| <code>exitonclick()</code>       | Fecha janela controlado pelo rato.      |
| <code>setup(L, A, IX, IY)</code> | Define tamanho e posição da janela      |
| <code>title(cadeia)</code>       | Define título da janela igual a cadeia. |

Tabela A.11: Específico da Janela

| Nome                                                   | Descrição                         |
|--------------------------------------------------------|-----------------------------------|
| <code>bgcolor(args)</code>                             | Define a cor de fundo.            |
| <code>bmpic(args)</code>                               | Define uma imagem de fundo.       |
| <code>clearscreen()</code>                             | Limpia a janela.                  |
| <code>screensize(l,a,cor)</code>                       | Define tamanho e cor de fundo     |
| <code>setworldcoordinates<br/>(llx,lly,urx,ury)</code> | Altera o sistemas de coordenadas. |

Tabela A.12: Controlo da janela

## A.5 Eventos

| Nome                                    | Descrição                        |
|-----------------------------------------|----------------------------------|
| <code>onscreenclick(fun,btw,add)</code> | Reage ao rato dentro da janela.  |
| <code>onkeypress(fun,arg)</code>        | Reage a carregar numa tecla.     |
| <code>onkeyrelease(fun,arg)</code>      | Reage a libertar uma tecla.      |
| <code>listen(x,y)</code>                | Dá o foco de atenção ao teclado. |

Tabela A.13: Eventos



## Bibliografia

- [1] Mark Lutz. *Learning Python (4th edition)*. O'Reilly, 2009.
- [2] M. Stefk, D. Bobrow, and D. Winter. Object-oriented programming: themes and variations, 1986.
- [3] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and applications*, 1987.