
Introdução à Programação e Resolução de Problemas

com Python

Ernesto Costa

Versão 0.9
12 de Setembro de 2014

A ti para que não penses que o dedico a outro/a.

Conteúdo

1	Introdução	1
1.1	Computadores	1
1.2	Linguagens	5
1.3	A linguagem Python	17
1.4	Programas	19
1.5	O meu primeiro programa	21
1.6	Fazer escolhas	26
1.7	Repetir	27
1.8	Intermezzo	29
1.9	Módulos	30
1.10	Adivinhar	32
1.11	Modo não interactivo	36
	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	58
	Sumário	64
	Teste os seus conhecimentos	64
	Exercícios	64
3	Objectos (I)	71
3.1	Generalidades	71
3.2	Números	76

3.3 Booleanos	88
3.4 Cadeia de Caracteres	91
3.5 Range	107
3.6 Tuplos	110
3.7 Intermezzo	120
3.8 Imutabilidade	122
Sumário	128
Teste os seus conhecimentos	128
Exercícios	128
4 Instruções Destrutivas	135
4.1 Generalidades	135
4.2 Atribuição	138
4.3 Leitura	147
4.4 Escrita	149
Sumário	153
Teste os seus conhecimentos	153
Exercícios	153
5 Instruções de Controlo	161
5.1 Introdução	161
5.2 Sequências	165
5.3 Condicionais	167
5.4 Ciclos	174
5.5 Intermezzo	179
5.6 Qual o valor de π ?	182
5.7 Outras Instruções de controlo	190
5.8 Excepções	194
Sumário	199
Teste os seus conhecimentos	199
Exercícios	200
6 Objectos (II)	207
6.1 Introdução	207
6.2 Listas	207
6.3 Dicionários	226
6.4 Mais exemplos	243
Sumário	245
Teste os seus conhecimentos	245
Exercícios	252

7 Ficheiros	253
7.1 Generalidades	253
7.2 Leitura	256
7.3 Escrita	261
7.4 Navegar	262
7.5 Intermezzo	267
7.6 Exemplo	268
7.7 De um ficheiro de palavras a um dicionário de frequências	277
7.8 Outros Tipos de Ficheiros	279
Sumário	286
Teste os seus conhecimentos	286
Exercícios	288
8 Visões (II)	295
8.1 Introdução	295
8.2 Representação de imagens	297
8.3 O módulo cImage	301
8.4 Exemplos Básicos	303
8.5 Manipulações simples	309
8.6 Intermezzo: abstracção	314
8.7 Exemplos complementares	316
8.8 Filtros	325
8.9 O formato de imagem PPM	334
Sumário	337
Teste os seus conhecimentos	338
Exercícios	338
9 Recursividade	347
9.1 Conceitos	347
9.2 Exemplos	352
9.3 Exemplos Complementares	374
9.4 Quando usar?	379
Sumário	383
Teste os seus conhecimentos	383
Exercícios	384
10 Complementos	393
10.1 Introdução	393
10.2 Ambiente e Alcance das Variáveis	396
10.3 Módulos	415
10.4 Definições e Argumentos	421

10.5 Iteradores e Geradores	436
10.6 Funções de Ordem Superior	449
Sumário	460
Teste os seus conhecimentos	460
Exercícios	461

Listas de Tabelas

1.1	Modelo PCAP	16
2.1	Nomes	58
3.1	Literais Numéricos	76
3.2	Operações básicas sobre números	79
3.3	Operadores de conversão de tipos	82
3.4	Operadores ao nível do Bit	86
3.5	Operadores Booleanos	88
3.6	Operadores de Comparação	89
3.7	Literais Para Cadeias de Caracteres	92
3.8	Caracteres de controlo em cadeias de caracteres	93
3.9	Operações básicas para cadeias de caracteres	95
3.10	Operações adicionais para cadeias de caracteres	99
3.11	Métodos das cadeias de caracteres	100
3.12	Operações para tuplos	112
4.1	Palavras Reservadas	137
5.1	Representações de Falso	165
5.2	O que escolher? Preços em euros.	201
6.1	Literais para Listas	209
6.2	Operações sobre Listas	209
6.3	Métodos Pré-Definidos para Listas	220
6.4	Literais para dicionários	231
6.5	Operações sobre Dicionários	233
6.6	Métodos Pré-Definidos para Dicionários	235
7.1	Modos de abertura de ficheiros de texto	255

7.2	Operações de leitura com ficheiros	256
7.3	Operações de escrita com ficheiros	261
7.4	Operadores de navegação	263
7.5	Ficheiro de Dados: entrada	290
7.6	Ficheiro transformado	290
8.1	Imagens e tamanhos	297
10.1	Tipos de argumentos	435

Listas de Figuras

1.1	Arquitectura de um computador	2
1.2	Compilação	4
1.3	Interpretação	4
1.4	Interpretação: Python (Adaptado de [3])	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	17
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	60
2.8	Seno e coseno	63
2.9	Uma estrela	65
2.10	Espiral de Estrelas	65

2.11	Sem rei nem roque	66
2.12	Deixar rasto	67
2.13	Tanto quadrado...	67
2.14	Parece um nautilus...	68
2.15	O símbolo dos Jogos Olímpicos	69
2.16	Radioactividade	69
2.17	Serenidade e Equilíbrio	70
3.1	Ambiente, nomes e objectos	73
3.2	Tipos básicos	77
3.3	Conversão automática de tipos	83
3.4	Cadeia de caracteres: indexação	96
3.5	Organização dos tuplos na memória	115
3.6	Partilha da memória	115
3.7	Embricamento	116
3.8	Partilha da memória	123
3.9	Mutabilidade (I): antes (A) e depois (B)	126
3.10	Mutabilidade (II): antes (A) e depois (B)	127
4.1	Ligaçāo Nomes - Objectos	139
4.2	Tipagem Dinâmica	141
4.3	Mudança de identidade	142
4.4	Entrada e Saída de dados	147
5.1	Os blocos de um programa	162
5.2	Sequência	166
5.3	A condicional if-then	167
5.4	A condicional if-then-else	169
5.5	A condicional if-elif-else	170
5.6	Ciclo for	177
5.7	O ciclo while	178
5.8	O Padrão Acumulador	180
5.9	Simulação de monte Carlo: o caso de π	185
5.10	Monte Carlo animado	188
5.11	A função e^{-x^2}	189
5.12	Hierarquia de excepções (visão parcial)	197
5.13	Calcular probabilidades	203
5.14	Desenho de uma grelha	205
5.15	Passeio aleatório	206
6.1	A representação de uma lista simples	212
6.2	A lista alterada	213

6.3	Uma lista mais complexa	213
6.4	<i>Aliasing</i>	214
6.5	Alterar sem efeitos não desejados	215
6.6	Tudo se complica...	216
6.7	Sem problema...	218
6.8	Ambientes, objectos e nomes	220
6.9	As Baleias dia a dia	227
6.10	Código Genético	228
6.11	Expressão Genética	229
6.12	Códificação das Bases	238
6.13	Passeando na cidade	249
7.1	Representação em memória	255
7.2	Um ficheiro depois de aberto	257
7.3	Situação depois de lidos os primeiros 8 bytes	260
7.4	Temperaturas das cidades de Portugal	273
7.5	Gráfico com legenda	274
7.6	Temperatura e pluviosidade	277
7.7	Coeficiente de Correlção de Pearson: exemplos	283
7.8	A Apple e a Coca-cola	287
8.1	Mapeamento RGB cores	296
8.2	Os tipos de cImage	302
8.3	Um janela simples	304
8.4	Fundo vermelho	304
8.5	Uma imagem em branco ... é preta!	305
8.6	Lidar com a cor e a posição	306
8.7	Reproduzir uma imagem	308
8.8	Negativo de uma imagem	310
8.9	Escala de cinzentos	312
8.10	Mais cinzento	313
8.11	Sepia	314
8.12	Alterando o brilho	317
8.13	O chão da cozinha	318
8.14	O problema do posicionamento	319
8.15	Distorcer uma imagem	321
8.16	Clonar um pixel	322
8.17	Espelho vertical	323
8.18	Onde colocar os pixeis?	324
8.19	Tratamento da pixelização	325
8.20	Vizinhança	326

8.21 Aplicar um filtro a uma imagem	329
8.22 Filtro gaussiano	330
8.23 Aplicando um filtro gaussiano	330
8.24 Operadores de Sobel	332
8.25 Operadores de Sobel	334
8.26 Acrescentar uma moldura	340
8.27 Corte de imagem	340
8.28 A preto e branco	341
8.29 Redução de uma imagem	342
8.30 Espelho horizontal	343
8.31 Redução de cores	343
8.32 Suavizar sem moldura	344
8.33 Gandstein	344
8.34 Encriptar uma imagem	345
9.1 <i>Sierpinski Gasket</i>	348
9.2 Sierpinski: construção	348
9.3 Torres de Hanói	349
9.4 Resolução de um sub problema semelhante	350
9.5 Torres de Hanói: solução final	350
9.6 Factorial: fase de desenrolar	353
9.7 Factorial: fase de enrolar	354
9.8 Fibonacci: coelhos e reprodução	358
9.9 Números de Fibonacci e Binómio de Newton	360
9.10 Inverter uma sequência	363
9.11 Inversão: alternativa	364
9.12 Sobe e Desce	366
9.13 Uma figura simples	369
9.14 Mudando o ângulo	369
9.15 Variando o lado e o ângulo	371
9.16 Que linda pirâmide	371
9.17 Um desenho diz mais do que mil palavras?	371
9.18 Pirâmide: pensar recursivo	372
9.19 Uma pirâmide colorida	374
9.20 Ordenamento por Fusão	376
9.21 Ordenamento Rápido	377
9.22 Sierpinski: 200,4	379
9.23 Fibonacci: cálculos duplicados	380
9.24 Ovais no plano. Caso de $n = 3$	385
9.25 Uma árvore recursiva	386
9.26 O processo	386

9.27 Uma árvore mais realista	387
9.28 Detector de Paridade Par	389
9.29 Floco de Neve	390
9.30 Curvas de Hilbert	391
10.1 Um programa	394
10.2 A hierarquia dos espaços de nomes	396
10.3 Ambiente inicial. Existem dois espaços de nomes, sendo que <u>main</u> está ligado a <u>builtins</u> (seta a tracejado). Sob a forma de nuvens aparecem os (descritores dos) objectos.	399
10.4 O ambiente com os vários espaços de nomes. O espaço de nomes main é o espaço activo e através dele accedemos aos diferentes objectos. Não é possível aceder directamente ao módulo cmath que foi importado pelo módulo raizes . A tracejado fino indicam-se as ligações dos descritores dos módulos e das definições e os seus espaços de nomes. Para não sobrecarregar a imagem, e sempre que tal não comprometa a compreensão da figura passaremos a omitir algumas destas ligações.	401
10.5 Criação de um espaço de nomes local à definição	402
10.6 A partir do módulo main accedemos ao módulo raizes , e a partir deste ao módulo cmath . É então possível calcular qual o objecto que vai ser associado com o nome comum . Os espaços criados pelas chamadas das definições são locais ao módulo onde foram definidas. A figura ilustra o momento da devolução do resultado do cálculo da raiz.	403
10.7 Alcance dos nomes (de variáveis).	404
10.8 Os nomes e os respectivos espaços.	405
10.9 Calculando um valor	406
10.10 Alcance das variáveis	412
10.11 O seu a seu dono	413
10.12 Como se chega ao resultado	414
10.13 Importação simples	416
10.14 Importação selectiva	418
10.15 Definições e memória	424
10.16 Antes	426
10.17 No início: caso 1	426
10.18 No início: caso 2	427
10.19 Antes da chamada	428
10.20 Chamada: associação dos parâmetros	429
10.21 Durante a execução os objectos são alterados	429
10.22 Após a execução	430

10.23copy versus deepcopy	431
-------------------------------------	-----

Listagens de Código

1.1	Arranque do interpretador	20
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	60
2.4	Seno e cosseno	61
3.1	Operações com números	78
3.2	Uso das plicas	92
3.3	Marcas em cadeias de caracteres	93
3.4	Caracteres: operações básicas	95
3.5	Mais operações	99
3.6	Partilha	122
5.1	Blocos	162
5.2	Operadores relacionais	163
5.3	Outros operadores	164
5.4	Sequência	165
5.5	Condicional normal	168
5.6	Condicional Geral: exemplo	171
5.7	Raízes: solução trivial	171
5.8	Raízes múltiplas	172
5.9	Testa raízes reais	172
5.10	Raízes: solução geral	173
5.11	Desenhar por repetição	174
5.12	De novo o quadrado	176
5.13	Formas simples	176
5.14	Ciclo for	176
5.15	Introdução protegida de dados	178

5.16	π segundo Leibniz	183
5.17	Método de Monte Carlo animado	186
5.18	Calcular uma área	189
5.19	Break: exemplo de uso	190
5.20	Ciclos <i>potencialmente</i> infinitos	191
5.21	Break: quadrados perfeitos	191
5.22	Continue: ímpares	192
5.23	Continue: entra código	192
5.24	Else: números primos	192
5.25	Tudo junto	193
5.26	Excepções: divisão por zero	194
5.27	Try: raízes reais apenas	195
5.28	Excepções: definição pelo utilizador	196
5.29	try: uso de finally	198
5.30	Teste para abortar programa	199
5.31	Ciclo while	201
6.1	Nota	208
6.2	Baleias	208
6.3	Mutabilidade e Referências	213
6.4	Mutabilidade e Referências (II)	214
6.5	Média	225
6.6	Média 2	225
6.7	Mediana	225
6.8	Desvio Padrão	226
6.9	Visualização	226
6.10	código genético	230
6.11	Genealogia I	245
6.12	Genealogia II	245
6.13	Genealogia III	245
6.14	Pares e Ímpares	247
6.15	alterna	247
6.16	contar menores	247
6.17	Índices das ocorrências	249
7.1	Abertura de um ficheiro	254
7.2	Leitura completa de um ficheiro	258
7.3	Leitura de ficheiros	258
7.4	Operações de entrada	259
	261
7.5	Escrita num ficheiro	261
7.6	'Navegar num ficheiro'	263
7.7	Ler e mostrar temperaturas	269

7.8 Ler ficheiro: alternativas	270
7.9 Todas as temperaturas	270
7.10 Temperaturas	271
7.11 Gráfico com legendas	272
7.12 Compara cotações	285
7.13 'Desemprego'	291
7.14 'Crescimento do PIB'	291
9.1 'Torres de Hanói'	351
9.2 'Exemplo de sessão'	351
9.3 Factorial	352
9.4 'Soma'	355
9.5 'Produto'	355
9.6 'Exponencial'	355
9.7 'Modelo de recursividade linear'	355
9.8 'Somatório'	356
9.9 'MDC: iterativo'	356
9.10 'Algoritmo de Euclides'	357
9.11 'Sequência de Fibonacci'	358
9.12 'Binómio de Newton'	359
9.13 'Par - Ímpar'	360
9.14 'Sequência Alternada'	361
9.15 'Procura Simples'	362
9.16 'Capicua'	363
9.17 'Inverte'	363
9.18 'Inverte: alternativa'	364
9.19 'Inverte: mais uma alternativa'	365
9.20 'Sobe e Desce'	365
9.21 Alternar	366
9.22 Intercalar	367
9.23 Intercalar: variante	367
9.24 'Procura Binária'	368
9.25 'Uma figura simples'	368
9.26 'Mais uma figura'	370
9.27 'Ainda outra figura'	370
9.28 'As pirâmides'	372
9.29 'As linhas'	373
9.30 'Quadrados'	373
9.31 'Anagrama'	374
9.32 'Anagrama: variante'	375
9.33 'Permutações'	375
9.34 'Ordenamento por Fusão'	376

9.35 'Ordenamento Rápido'	377
9.36 'Sierpinski Gasket'	378
9.37 De novo o factorial	381
9.38 Tipo de Dados Pilha	382
9.39 Factorial Iterativo	382
9.40 'Detector de Paridade Par'	389

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 9. 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

Todo o começo é difícil – isso
vale em qualquer ciência.

Karl Marx, *in* Prefácio à
primeira edição alemã de “O
Capital”

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 secretaria 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 defenda (e.g., Herbert Simon) que o ser humano e os computadores são apenas duas instâncias de algo mais abstracto 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.

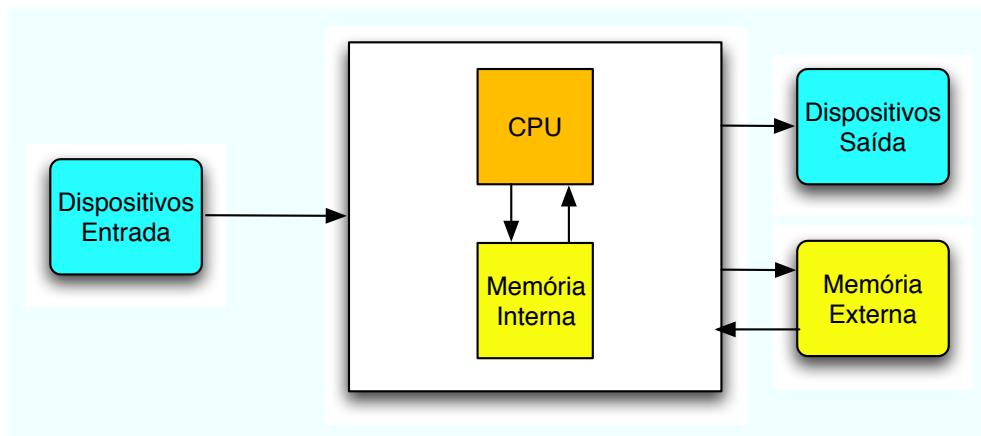


Figura 1.1: Arquitectura de um computador

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.

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 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.

[Arquitectura
de Von
Neumann](#)

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.

[Compiladores e
Interpretadores](#)

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.

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.

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

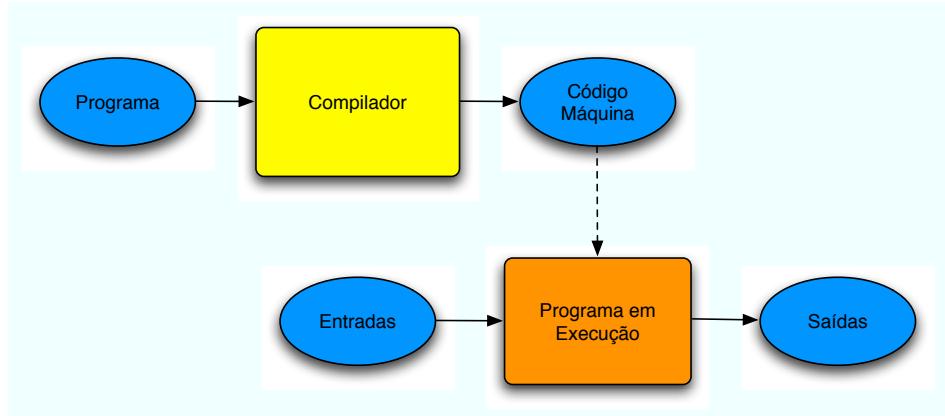


Figura 1.2: Compilação



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).

⁷ As 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.

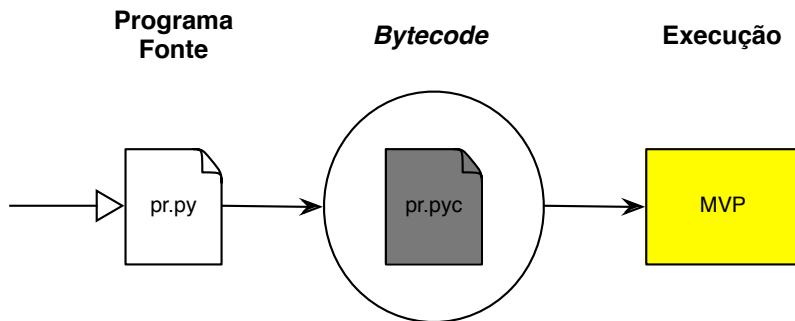


Figura 1.4: Interpretação: Python (Adaptado de [3])

1.2 Linguagens

Em qualquer sociedade a comunicação entre os seus membros é mediatizada 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 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

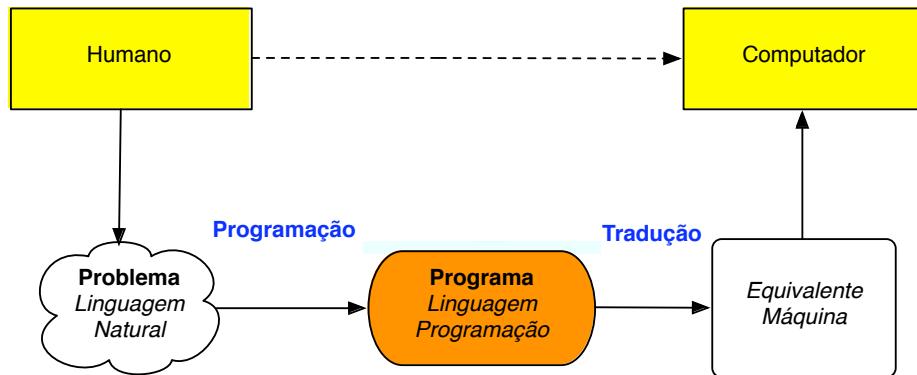


Figura 1.5: O processo de comunicação humano - computador

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,
- 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

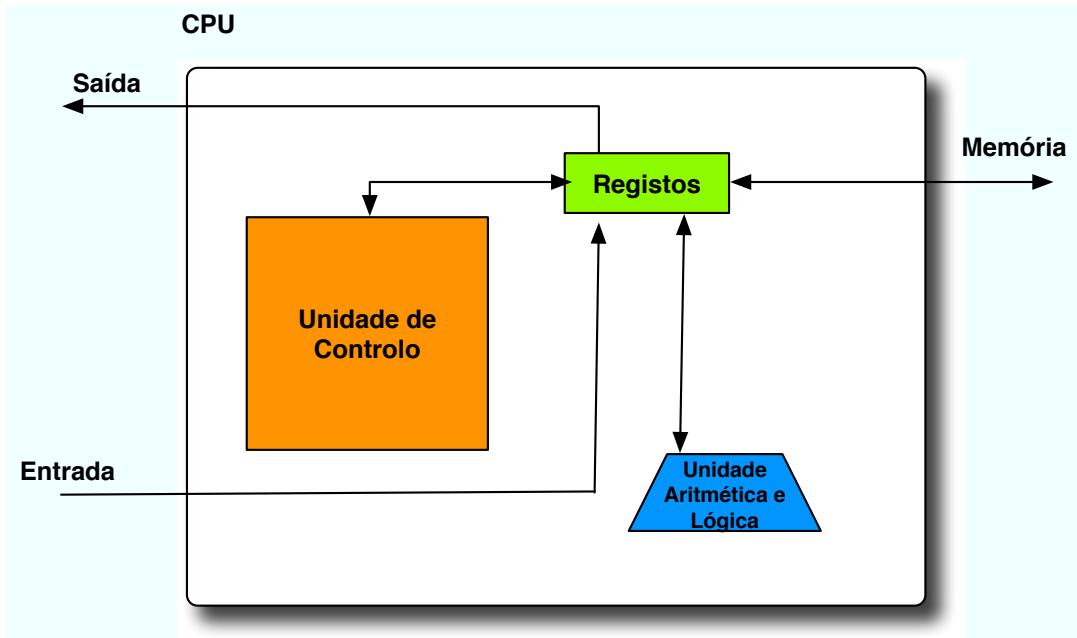


Figura 1.6: A unidade de processamento central

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 pudesse 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,

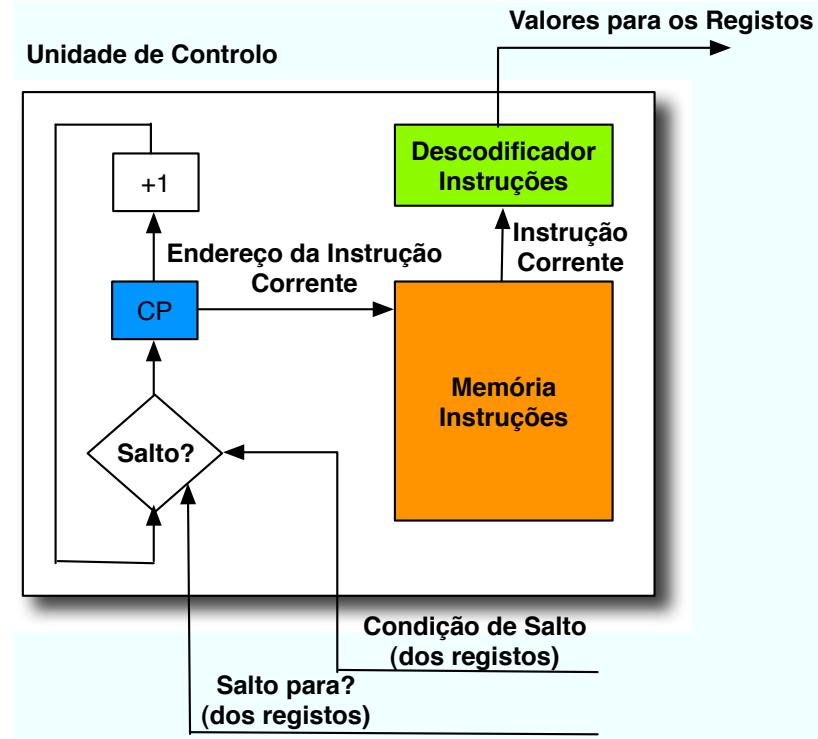


Figura 1.7: A unidade de controlo

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.

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

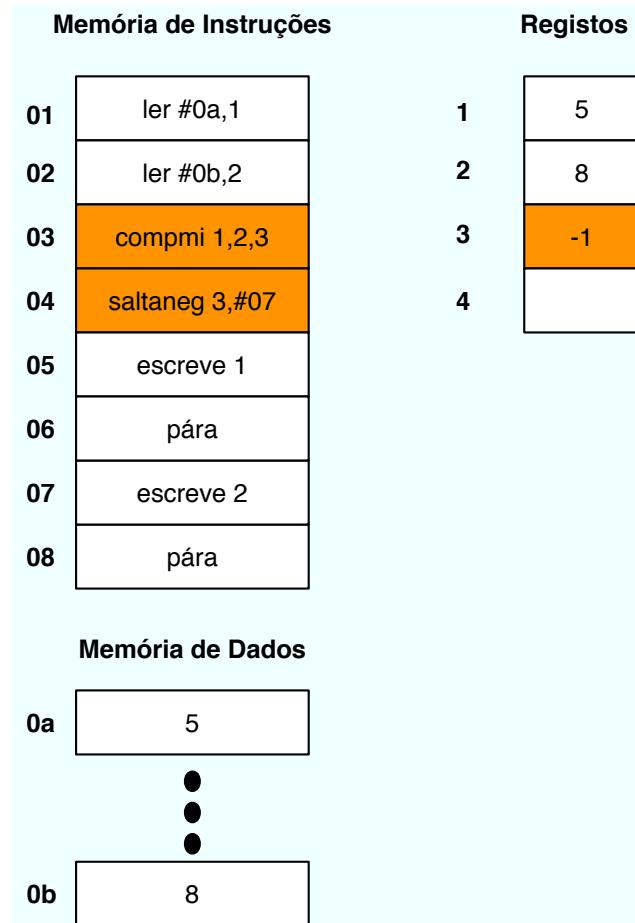


Figura 1.8: Programa em Assembly

Programa = Dados + 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.

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/saída). 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 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 ([4])

"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

```

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

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).*).

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 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 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 }
```

¹⁰Mais adiante, no capítulo 9, trataremos com mais cuidado do problema da recursividade.

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 ([5]):

$$\text{Orientação aos Objectos} = \text{Objectos} + \text{Classes} + \text{Herança}$$

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 }
```

¹¹No sentido do inglês *template*.

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*.

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).

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

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>.

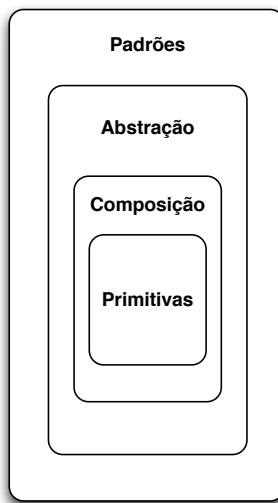


Figura 1.9: Modelo PCAP

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

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!

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 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 empresas 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.

```

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/>.

- 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!

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 [3]):

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 como 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 como que se ilustra na listagem 1.1.¹⁶

¹⁶O que efectivamente aparece varia de máquina para máquina e de versão do interpretador.

```

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 >>>

```

Listagem 1.1: Arranque do interpretador

O símbolo `>>>` é o **caractér de pronto**¹⁷ 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**¹⁸ (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

¹⁷Do inglês *prompt character*.

¹⁸Do inglês *Read - Eval - Print*.

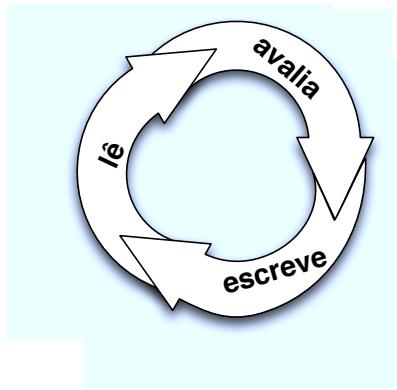


Figura 1.10: O ciclo lê - avalia - escreve.

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).

```

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 decimal¹⁹. 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 (=)²⁰. Vejamos para o homem como pode ser feita a associação.

Variável

¹⁹O 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.

²⁰Para 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 $x = x + 5$?

```

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
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.

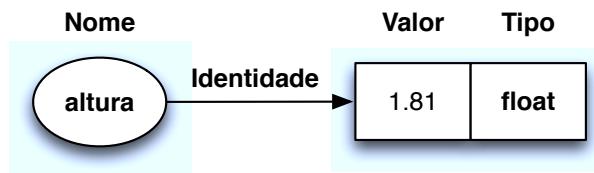


Figura 1.11: Objectos e seus atributos

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 ser 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 expressão²¹. Os três tipos de instruções referidos, atribuição, entrada e saída, formam um grupo importante

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

Cadeias de Caracteres

de instruções do paradigma procedural, 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.

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).

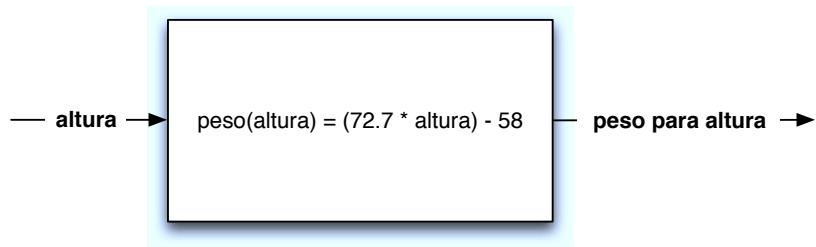


Figura 1.12: A função peso: conhecido o valor de `altura` podemos calcular o peso.

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 10.15 retrata a situação.

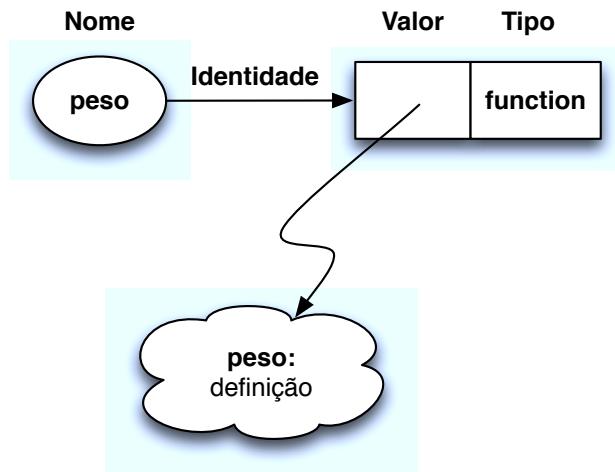


Figura 1.13: Associação entre um nome e uma definição

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 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 fazer²².

```

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 igual²³. 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:

```

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

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

Instruções de Controlo

```

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 >>>

```

Veja-se como usámos números para codificar o género²⁴. 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 lin-

²⁴Quando falarmos de outros tipos de objectos veremos que podemos usar, com mais a propósito, cadeias de caracteres.

Ciclos

guagens 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 genera == 1:
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 >>>

```

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)²⁵, 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.).

²⁵Existem mais tipos de números como mais tarde se verá.

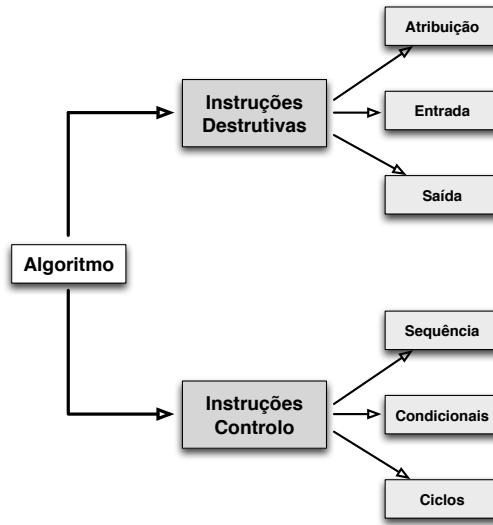


Figura 1.14: Instruções

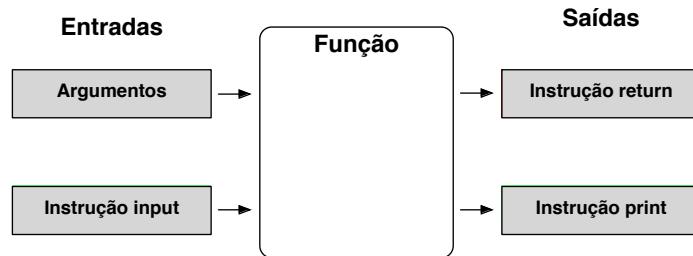


Figura 1.15: Entrada e saída de objectos

1.9 Módulos

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**.

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 aparenta ter nenhuma dificuldade. Envolve, no entanto, cálculos com a constante π . 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 π 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 π 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. Mas isso coloca a questão de como introduzir os diferentes valores. 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 ...

```

[Métodos](#)

```

9  >>> adivinha()
10 O seu palpito: 33
11 False
12 >>> adivinha()
13 O seu palpito: 25
14 False
15 >>>

```

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 0 seu palpite: 33
11 0 seu palpite: 25
12 0 seu palpite: 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("0 seu palpite 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 0 seu palpite sff: 33
19 Muito pequeno...
20 0 seu palpite sff: 55
21 Muito pequeno...
22 0 seu palpite 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 perde-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 indutiva 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 na listagem 1.3.

```

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

```

Listagem 1.3: Raiz quadrada

O código tem duas partes: a primeira (linhas 1 a 7), consiste na definição da função que permite o cálculo da raiz quadrada; a segunda (linhas 9 e

10), permite o uso da definição²⁶. A listagem abaixo ilustra uma forma de 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 sobre o que faz o programa e a sua colocação naquela zona é importante para efeitos de documentação do programa²⁷.

[Comentários](#)

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 9 e 10 da listagem 1.3

```
if __name__ == '__main__':
    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.

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

²⁷Incluindo a documentação automática.

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,

- 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 9.459×10^{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 8×6 . Admitindo que quer cobrir o chão com tijoleira de 2×2 , 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 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 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 1.16.

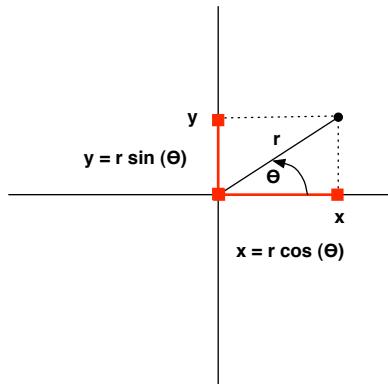


Figura 1.16: De cartesianas a polares

Exercício 1.18 Módulo math M Johannes Kepler²⁸ 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.

²⁸http://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).

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 booleanos¹. 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

¹Claro 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.

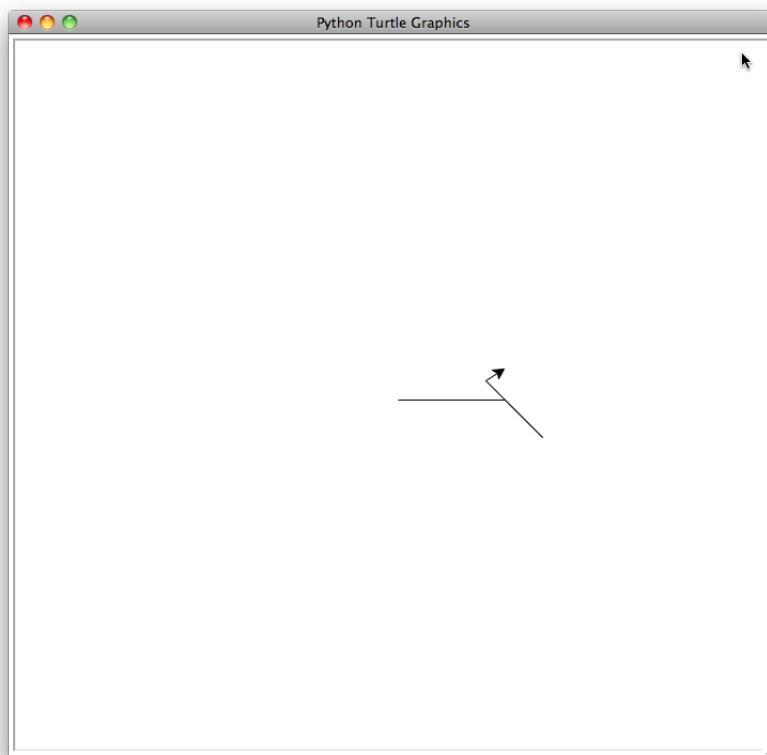


Figura 2.1: O passeio (curto) da tartaruga

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 é, 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 >>> turtle.setx(200)
4 >>> turtle.ycor()
5 100
6 >>> turtle.xcor()
7 200
8 >>> turtle.setheading(225)
9 >>>
```

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 ?? 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° , desenhar o segundo lado, rodar 90° , 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 2.2).

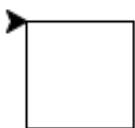


Figura 2.2: Mas que lindo quadrado

Notar que, ao efectuar uma quarta rotação, a direcção da tartaruga é 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, x_cor, y_cor, angulo):
4     """
5         Desenha um quadrado com o comprimento de lado, o vértice
6             inferior esquerdo em (x_cor, y_cor) e direcção inicial angulo.
7     """
8     # Preparação
9     turtle.goto(x_cor, y_cor)
10    turtle.setheading(angulo)
11    # Desenha
12    for conta in range(4):
13        turtle.forward(lado)
14        turtle.right(90)
15    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.

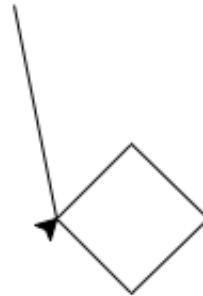


Figura 2.3: Um quadrado defeituoso

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, x_cor, y_cor, angulo):
4     """
5         Desenha um quadrado com o comprimento de lado, o vértice
6             inferior
7             esquerdo em (x_cor, y_cor) e direcção inicial angulo.
8     """
9     # Preparação
10    turtle.penup()
11    turtle.goto(x_cor, y_cor)
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 código². 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 argumento, tenta 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

²Generalizar é outro modo de dizer abstrair.

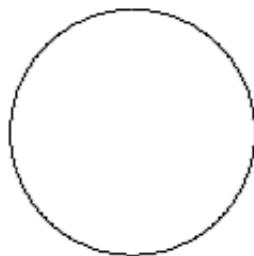


Figura 2.4: Uma bela circunferência

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 operações³. 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()
```

³Tecnicamente, 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**⁴.

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

⁴(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.penup()
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.

Formas Como vimos a tartaruga pode assumir diversas formas de entre um conjunto pré-definido: *array*, *turtle*, *circle*, *square*, *triangle* e *classic*. Esta última é a que é utilizada por defeito. O utilizador pode, no entanto, definir ele próprio a forma da tartaruga. O modo mais directo consiste na construção de um polígonos que depois é registado como uma nova forma. Vejamos como se processa.

```

1  turtle.penup()
2  turtle.begin_poly()
3  for i in range(5):
4      turtle.forward(10)
5      turtle.left(72)
6  turtle.end_poly()
7  turtle.pendown()
8
9  pentagono = turtle.get_poly()
10 turtle.register_shape('pentagono',pentagono)
11
12 turtle.shape('pentagono')
13 turtle.forward(100)
14 turtle.left(45)
15 turtle.forward(50)

```

Como se pode ver definimos uma sequência de segmentos ligados, especificando que se trata de um polígono (linhas 1 a 7). Mais tarde, recuperamos o polígono, damos-lhe um nome (linha 9) e registamos a respectiva forma (linha 10). A partir desse momento só temos que definir a nova forma como a forma da nossa tartaruga (linha 12) e começar a movimentá-la (linhas 13 - 14).

Podemos definir ainda novas formas recorrendo ao construtor de formas pré-definido **Shape**. Existem três formas alternativas.

```
1 # 1: polígono
2 forma_1 = turtle.Shape('polygon',((0,0),(-5,0),(-5,10),(0,5),
3   ,(5,10),(5,0)))
4 turtle.register_shape('dente',forma_1)
5
6 turtle.shape('dente')
7 turtle.forward(100)
8 turtle.left(45)
9 turtle.forward(50)
10
11 # 2: imagem
12 forma_2 = turtle.Shape('image', 'play.gif')
13 turtle.register_shape('play',forma_2)
14
15 turtle.shape('play')
16 turtle.forward(100)
17 turtle.left(45)
18 turtle.forward(50)
19
20 # 3: Forma composta
21 forma_3 = turtle.Shape('compound')
22 forma_3.addcomponent(((0,0),(-5,0),(-5,10),(0,5),(5,10),(5,0))
23   , 'red','black')
24 forma_3.addcomponent(((5,0),(0,5),(5,0)), 'black','red')
25
26 turtle.register_shape('oh_my',forma_3)
27
28 turtle.shape('oh_my')
29 turtle.forward(100)
30 turtle.left(45)
31 turtle.forward(50)
```

A primeira possibilidade consiste em definir, como no caso anterior, um polígono, indicando o posicionamento dos seus vértices. Faz-se o registo da forma e usa-se (linhas 2 a 8). A segunda possibilidade, baseia-se no recurso a uma imagem que é registada como sendo uma forma (linhas 11 a 17). Finalmente, a terceira oportunidade, recorre à definição de componentes que depois são acrescentadas à forma da tartaruga (linhas 20 a 28).

Nomes longos e curtos Vários dos métodos do módulo **turtle** têm diferentes maneiras de ser nomeadas. Isso é verdade para os métodos, por exemplo, que envolvem o movimento, a posição, a orientação, a visibilidade e a escrita.

Tipo	Longo	Curto
Movimento	forward()	fd()
Movimento	backward()	bk()
Posição	setposition()	setpos()
Orientação	setheading()	seth()
Visibilidade	hideturtle()	ht()
Escrita	pendown()	pd() ou down()
Escrita	penup()	pu() ou up()

Tabela 2.1: Nomes

Estas formas, longas e curtas, podem ser misturadas. O leitor deve consultar o manual da linguagem para uma listagem completa. Ao longo do texto usaremos as diversas formas.

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 π .

¹ `import turtle`

```

2 import math
3
4 def grafico(tartaruga, funcao, inicio, fim, n):
5     """ Desenha o gráfico da função f entre inicio e fim
6         usando n segmentos."""
7     # Tamanho dos segmentos
8     tam_seg = (fim - inicio)/n
9     # Posiciona-se
10    x = inicio
11    tartaruga.up()
12    tartaruga.goto(x, funcao(x))
13    tartaruga.down()
14    # Desenha
15    for conta in range(n):
16        x = x + tam_seg
17        tartaruga.goto(x, funcao(x))
18
19 if __name__ == '__main__':
20     turtle.setworldcoordinates(-math.pi, -2, math.pi, 2)
21     toto = turtle.Turtle()
22     toto.pen(pensize=5, pencolor='gray')
23     grafico(toto, math.sin, -math.pi, math.pi, 50)
24     toto.hideturtle()
25     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 rastro (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 a 16). 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

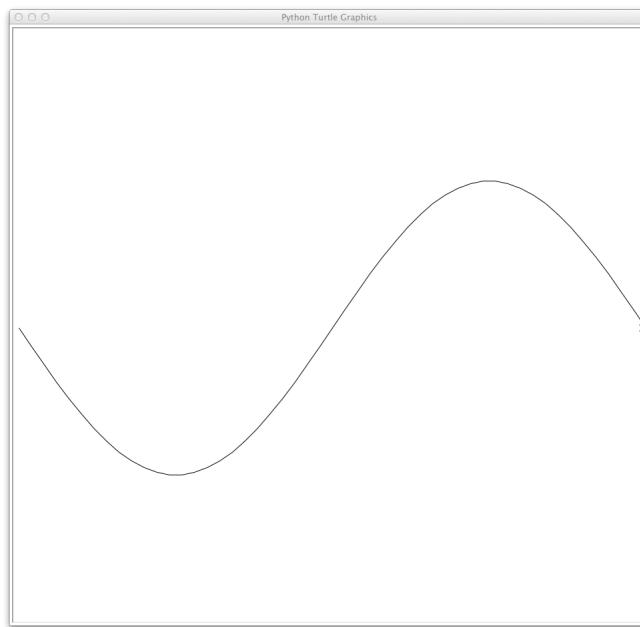


Figura 2.7: A função seno

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()
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
```

```

18     # Posiciona-se
19     tartaruga.up()
20     tartaruga.goto(x, funcao(x))
21     tartaruga.down()
22     # Desenha
23     for conta in range(n):
24         x = x + tam_seg
25         tartaruga.goto(x, funcao(x))
26
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

```

```

17     tam_seg = (fim - inicio)/n
18     x = inicio
19     # Posiciona-se
20     tartaruga.up()
21     tartaruga.goto(x, funcao(x))
22     tartaruga.dot(6)
23     tartaruga.down()
24     # Desenha
25     for conta in range(n):
26         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.

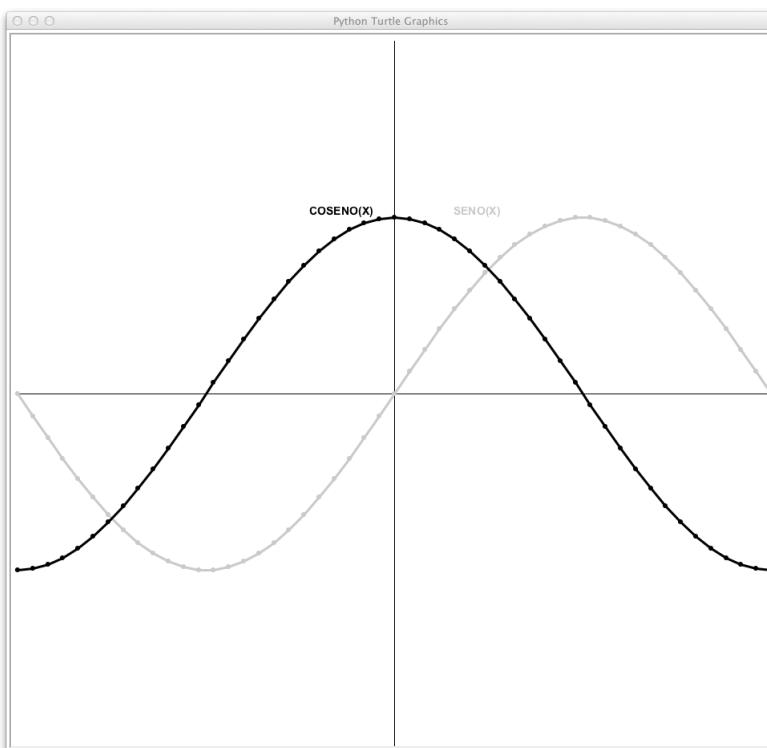


Figura 2.8: Seno e cosseno

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.

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.

Exercício 2.2 F

Adapte a ideia do exercício anterior para desenhar uma figura semelhante a 2.10. Procure implementar uma solução que possibilite variantes da figura apresentada.

Exercício 2.3 F

Faça um programa que lhe permita simular um passeio aleatório (*random walk*). A figura 2.11 ilustra o que se pretende. Procure alterar a cor de cada segmento de modo aleatório.



Figura 2.9: Uma estrela

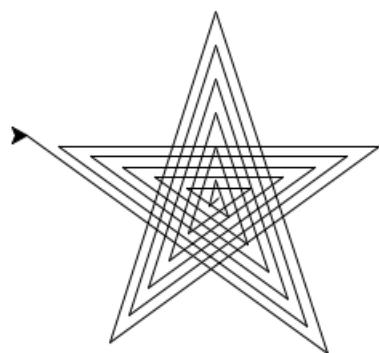


Figura 2.10: Espiral de Estrelas

Exercício 2.4 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 conta o valor do raio.

Exercício 2.5 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

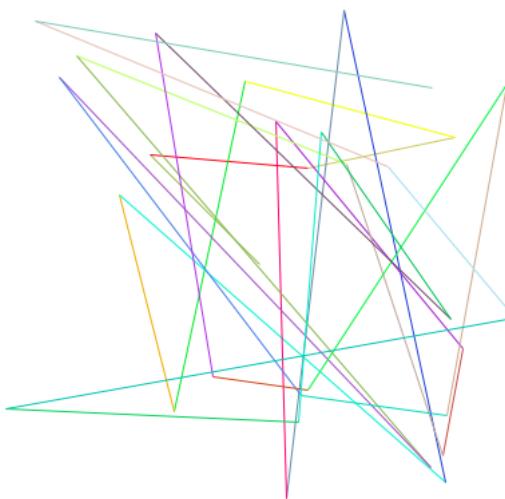


Figura 2.11: Sem rei nem roque

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 2.6 F

Usando o método descrito no capítulo construa diferentes formas para a tartaruga e use-as.

Exercício 2.7 M

O método `forward` permite movimentar uma tartaruga no sentido da sua orientação actual. Queremos alterar o seu funcionamento por forma que a tartaruga nunca se afaste mais do que um certo valor, definido pelo utilizador, do seu ponto de partida. Caso chegue ao limite do espaço que pode visitar a tartaruga pode, em alternativa:

1. alterar a sua orientação ficando apontada para a posição de partida e continuar a movimentar-se;
2. como no caso anterior mas permitindo uma pequena variação aleatória na nova orientação da tartaruga.

Socorra-se dos métodos `distance`, `towards` e `setheading` do módulo `turtle` para implementar dois programas que resolvam as duas situações

Exercício 2.8 M

Existe um método no módulo `turtle` que permite à tartaruga deixar um rastro dos lugares por onde andou. Esse comando chama-se `stamp`. Veja no manual da linguagem como funciona e escreva um programa que lhe permita desenhar o que a figura 2.12 ilustra. Procure que a sua solução seja genérica.

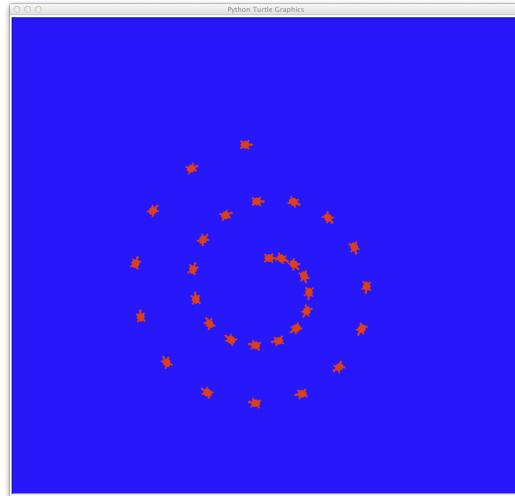


Figura 2.12: Deixar rastro

Exercício 2.9 D

Desenvolva um programa que lhe permita desenhar quadrados concêntricos como, por exemplo, indicado na figura 2.13.

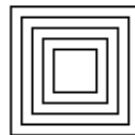


Figura 2.13: Tanto quadrado...

Exercício 2.10 D

Desenvolva um programa que lhe permita desenhar a figura 2.14. Se reparar bem a figura é formada por quadrados cujos lados têm dimensão diferentes e os ângulos iniciais têm orientações diferentes.

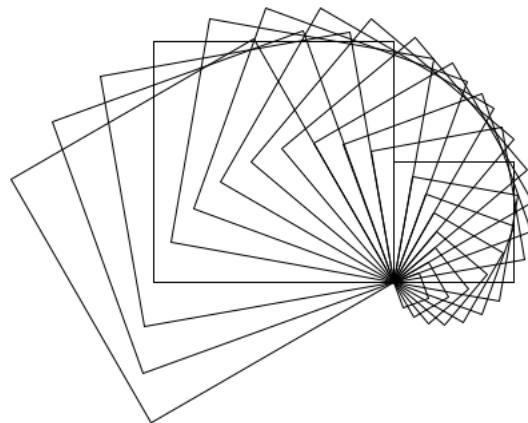


Figura 2.14: Parece um náutilus...

Exercício 2.11 M

Todos conhecemos o símbolo dos Jogos Olímpicos. A figura 2.15 mostra um exemplo.

Socorrendo-se do módulo **turtle** implemente um programa que permita desenhar o símbolo. Procure que a sua solução seja modular por forma a poder eventualmente aproveitar partes do seu programa para resolver outros problemas.

Exercício 2.12 D

Existem vários sinais universais, alguns que nos ajudam a evitar lugares indesejáveis. É o caso do sinal de presença de radioactividade que a figura 2.16 ilustra.

Socorrendo-se do módulo **turtle** implemente um programa que permita desenhar o símbolo. Procure que a sua solução seja modular por forma a poder eventualmente aproveitar partes do seu programa para resolver outros

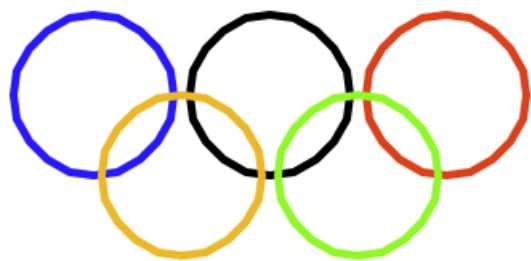


Figura 2.15: O símbolo dos Jogos Olímpicos

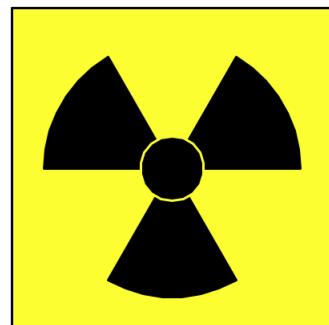


Figura 2.16: Radioactividade

problemas.

Exercício 2.13 D

Existem símbolos ligados a formas de ver e viver a vida. Um dos mais conhecidos é designado por **YinYang** que apresentamos na figura 2.17.



Figura 2.17: Serenidade e Equilíbrio

Socorrendo-se do módulo **turtle** implemente um programa que permita desenhar o símbolo. Procure que a sua solução seja modular por forma a poder eventualmente aproveitar partes do seu programa para resolver outros problemas.

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 podendo 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 tipo¹ 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
```

¹Não 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 obtém exactamente os mesmos resultados. Podemos visualizar (ver figura 3.1) a situação da memória que resulta da sessão anterior.

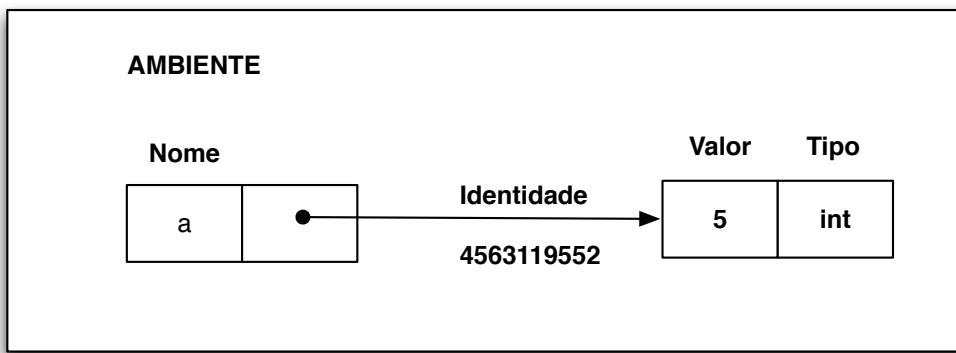


Figura 3.1: Ambiente, nomes e objectos

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 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  ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
   'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
   'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
   'FutureWarning', 'GeneratorExit', '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 mapeamentos². 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)³.

3.2 Números

Literais

Os objectos primitivos mais simples são os números. Existem fundamentalmente três tipos de números: inteiros, reais (ou vírgula flutuante)⁴ 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 numéricos⁵.

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.

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

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

⁴Do inglês *float*.

⁵Os inteiros também podem ser representados em notação binária, octal e hexadecimal.

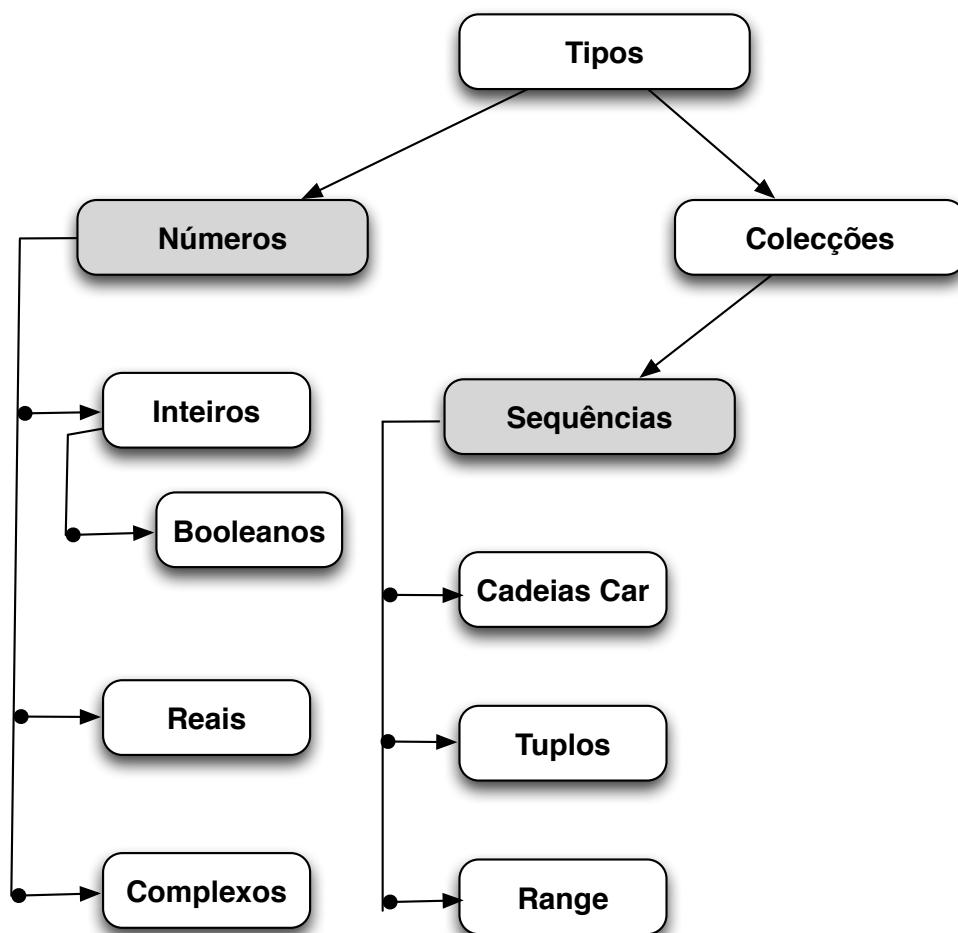


Figura 3.2: Tipos básicos

```

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 "ecoados" 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)
abs	Valor absoluto
conjugate	o conjugado de um complexo
divmod	Divisão truncada e resto
pow	Exponenciação
**	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 subconjunto¹. 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. Vejamos um exemplo simples.

```

1 >>> 0.1 + 0.2
2 0.3000000000000004

```

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* (ϵ) mais pequeno para o qual se verifica $1 + \epsilon > 1$.

```

1 >>> import sys
2 >>> sys.float_info
3 sys.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^a podemos retirar, entre outras coisas, os valores máximo e mínimo que se pode representar bem como o valor de *epsilon*. Ficamos a saber que tem uma precisão de 16 dígitos.

^aOs valores listados dependem da máquina em que está a correr o interpretador.

Na tabela 6.2 afirmamos que // representa a divisão truncada. Na rea-

lidade 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 temos 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 truncagem 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(<i>obj</i>, <i>base</i>=10)</code>	Converte para inteiro na base <i>base</i>
<code>float(<i>obj</i>)</code>	Converte para float
<code>complex(<i>real</i>, <i>imag</i>=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')

```

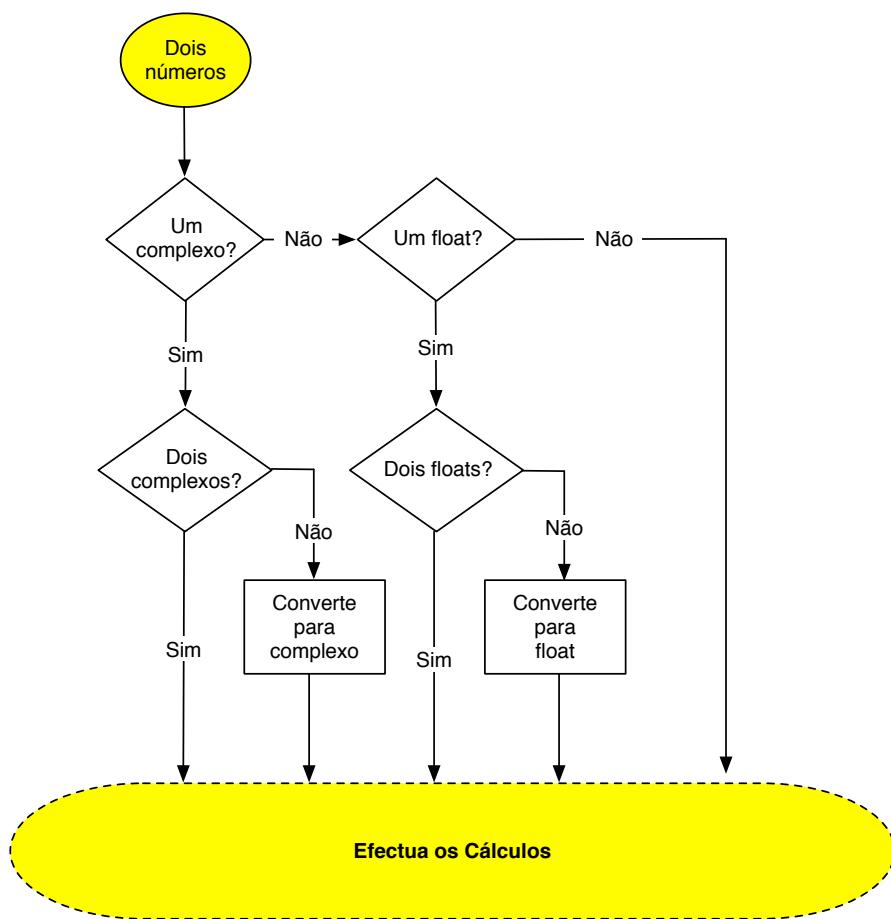


Figura 3.3: Conversão automática de tipos

```

6 123
7 >>> float('123')
8 123.0
9 >>> float('123e10')
10 1230000000000.0
11 >>>
12 >>> float(34)
13 34.0
14 >>> complex(5,3)
15 (5+3j)
16 >>> complex('4+5j')
17 (4+5j)
18 >>> complex(5)
19 (5+0j)
20 >>> float(4+5j)
21 Traceback (most recent call last):
22   File "<stdin>", line 1, in <module>
23 TypeError: can't convert complex to float; use abs(z)
24 >>> abs(4+5j)
25 6.4031242374328485

```

Estes exemplos mostram que nem tudo é possível fazer. Por exemplo, não podemos converter um número complexo num real⁶.

Construtores

As funções indicadas na tabela 3.3 são conhecidas como **construtores** do tipo⁷. 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 em obter separadamente cada uma destas partes. No caso dos complexos,

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

⁷Não é por acaso que têm o nome do tipo.

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 prioridade⁸. 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.

⁸Podemos 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 polinómio⁹.

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

```

⁹Na 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`¹⁰. Com argumentos genéricos (linhas

¹⁰Como 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 sequências¹¹, de caracteres¹², homogéneas (todos os seus elementos são do mesmo tipo, no caso de 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

¹¹Existem outros tipos de sequências como o *range*, os tuplos e as listas de que falaremos mais adiante.

¹²Em **Python** ao contrário de outras linguagens não existe o tipo caracter. Um caractere não é mais do que uma cadeia com um único elemento.

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
r'\t\nernesto\n costa'	Cadeia bruta

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 exemplo.

Caracteres de controlo¹³

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

¹³Do 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 3.8.

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 código¹⁴. 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](#)

¹⁴Existem 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. Vejamos alguns exemplos.

```

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 >>>

Operadores

Existe um conjunto de operações comuns aos vários tipos de sequências, logo que podem ser usadas com cadeias de caracteres. A tabela 3.9 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 **sobrecarregados**. 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 `[]`.

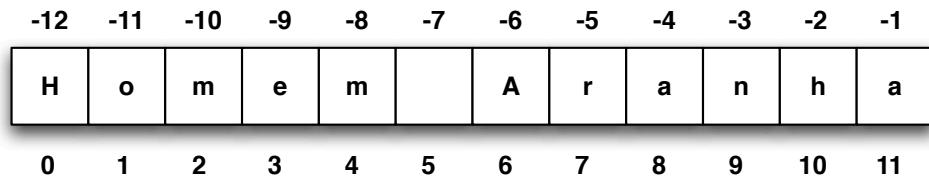


Figura 3.4: Cadeia de caracteres: indexação

```
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 'HmmAah'
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 parênteses¹⁵. 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 >>>

```

¹⁵Tecnicamente, 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 terminar¹⁶.

```

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 (%s). Aparece também um caractere de escape especial(\\n) 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>cad in s</code>	Determina se cad é sub cadeia de s
<code>max(cad)</code>	Qual o maior elemento da cadeia
<code>min(cad)</code>	Qual o menor elemento da cadeia
<code>s.index(cad)</code>	O índice da primeira ocorrência de cad em s
<code>s.count(cad)</code>	O número de ocorrências de cad em s

Nos dois últimos exemplos estamos perante métodos que se podem aplicar aos diferentes tipos de sequências. A listagem 3.5 ilustra algumas aplicações das operações da tabela 3.10 sendo evidente as diferenças de notação entre os métodos e as restantes operações.

```

1 >>> cadeia = 'Homem Aranha'
2 >>> 'ma' in cadeia
3 False
4 >>> max(cadeia)

```

¹⁶O leitor deve consultar o material para ver todas as possibilidades oferecidas pela linguagem.

```

5   'r'
6 >>> min(cadeia)
7   ,
8 >>> cadeia.count('a')
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 específicos

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(cad)</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
<code>s.endswith(cad)</code>	Verifica se s termina em cad

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 'UACGAUGGGUCAAUGUCGAU'
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 pesquisar¹⁷. 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)

```

¹⁷A procura é feita entre o primeiro valor inclusive e o último exclusive. Se este não for indicado procura-se até ao final da cadeia.

```

6   '123'
7   >>> str('abc')
8   'abc'
9   >>> str(12e3)
10  '12000.0'
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     """Transforma 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
3         de ARN."""
4     return arn.find('AUG')
5
6 def gene_pos_b(arn, pos):
7     """Indica a posição do início do primeiro gene na cadeia
8         de ARN,
9         a partir da posição pos."""
10    return arn.find('AUG', pos)
```

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 = ""

```

```

5     car_impares = ""
6     car_conta = 0
7     for car in texto_normal:
8         if car_conta % 2 == 0: # par ou ímpar?
9             car_pares = car_pares + car
10            else:
11                car_impares = car_impares + car
12                car_conta = car_conta + 1
13    texto_encriptado = car_impares + 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. No capítulo ?? voltaremos a falar sobre metodologia da programação.

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 '

```

```

6     texto_encriptado = ''
7     for car in texto_normal:
8         indice = alfabeto.find(car)
9         texto_encriptado = texto_encriptado + chave[indice]
10    return texto_encriptado

```

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 correspondente.

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 hiper-facilitada 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). Os **range** são mais um exemplo de sequência. 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
17     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 = ""

```

```

7   for indice in range(comp):
8       if indice % 2 == 0: # par ou ímpar?
9           car_pares = car_pares + texto_normal[indice]
10      else:
11          car_impar = car_impar + 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 eficiência¹⁸.

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 profundamente na secção 3.6.

```

1 >>> tuple(range(10))

```

¹⁸Existem outros aspectos a ter em conta, como por exemplo, a possibilidade de reutilizar o código.

```

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)
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     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). Os tuplos juntam-se assim às cadeias de caracteres e aos range

como exemplos de sequências. 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         ....
4
5     return math.sqrt((ponto_2[0] - ponto_1[0])** 2 + (ponto_2
6         [1] - ponto_1[1])** 2)

```

Cada ponto terá agora que ser representado por um tuplo¹⁹. 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 e que reproduzimos de novo na tabela 3.12

Alguns exemplos de utilização.

```

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

```

¹⁹Tuplos formados apenas por sequências de números são um modo natural de representar **vectores**.

Tabela 3.12: Operações para tuplos

Literal	Interpretação
<code>+</code>	Concatenação de cadeias de caracteres
<code>*</code>	Cópias de superfície de uma cadeia
<code>len</code>	Comprimento da cadeia
<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
<code>[i:j:k]</code>	Fatiamento

```

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 empacotamento²⁰. 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 ()

```

²⁰Do inglês *packing*.

```

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

```

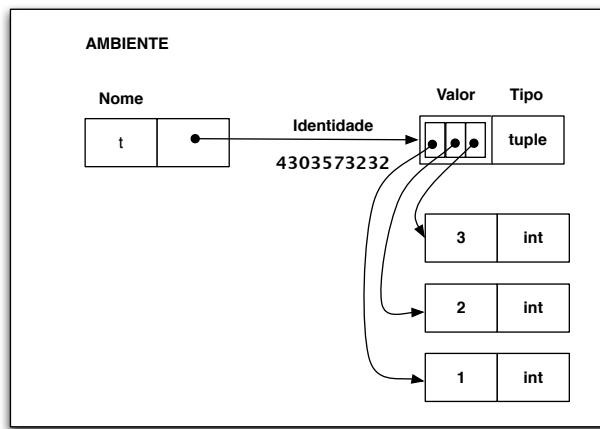


Figura 3.5: Organização dos tuplos na memória

```

5 4299455840
6 >>> id(t[1])
7 4299455840
8 >>> a = 2
9 >>> id(a)
10 4299455840
11 >>>

```

A figura 3.6 ilustra a situação.

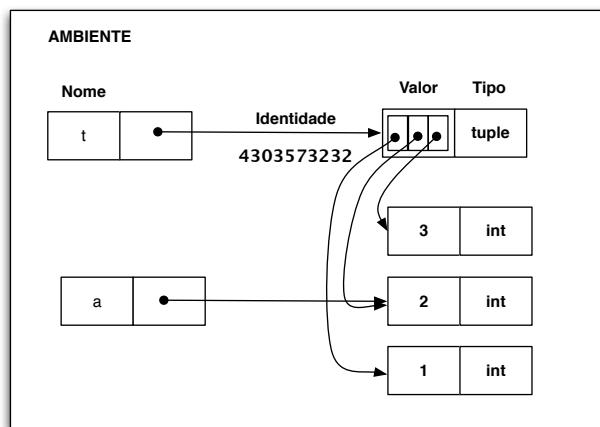


Figura 3.6: Partilha da memória

Embricamento

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)`.

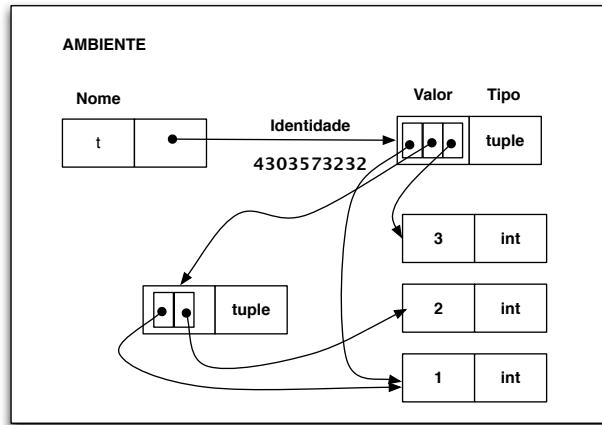


Figura 3.7: Embricamento

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, long=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 z', verbose=True)
2  from builtins import property as _property, tuple as _tuple
3  from operator import itemgetter as _itemgetter
4  from collections import OrderedDict
5
6  class OutroPonto(tuple):
7      'OutroPonto(x, y, z)'
8
9      __slots__ = ()
10
11     _fields = ('x', 'y', 'z')
12
13     def __new__(_cls, x, y, z):
14         'Create new instance of OutroPonto(x, y, z)'
15         return _tuple.__new__(_cls, (x, y, z))
16
17     @classmethod
18     def _make(cls, iterable, new=tuple.__new__, len=len):
19         'Make a new OutroPonto object from a sequence or
20             iterable'
21         result = new(cls, iterable)
22         if len(result) != 3:
23             raise TypeError('Expected 3 arguments, got %d
24                             % len(result)')
25         return result
26
27     def __repr__(self):
28         'Return a nicely formatted representation string'
29         return self.__class__.__name__ + '(x=%r, y=%r, z=%r)' % self
30
31     def _asdict(self):
32         'Return a new OrderedDict which maps field names
33             to their values'
34         return OrderedDict(zip(self._fields, self))
35
36     __dict__ = property(_asdict)
37
38     def _replace(_self, **kwds):
39         'Return a new OutroPonto object replacing
          specified fields with new values'
        result = _self._make(map(kwds.pop, ('x', 'y', 'z'),
                               _self))
        if kwds:
            raise ValueError('Got unexpected field names: %s' % kwds)
```

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 tem pelo menos um elemento.

```

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.

Expressões Geradoras

Por estranho que pareça, para um problema tão simples existem outras soluções para este problema.

```

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

```

Esta solução depende de conhecimento mais 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.

```

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
14            melhor_elem = elem
15            melhor_valor = valor
16
17    return melhor_elem, melhor_valor

```

```

10      # actualiza
11      melhor_valor = valor
12      melhor_elem = elem
13      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 Imutabilidade

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_3`, `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 simplificado²¹, 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

```

²¹Sempre que não causar ambiguidade apresentaremos os desenhos simplificados.

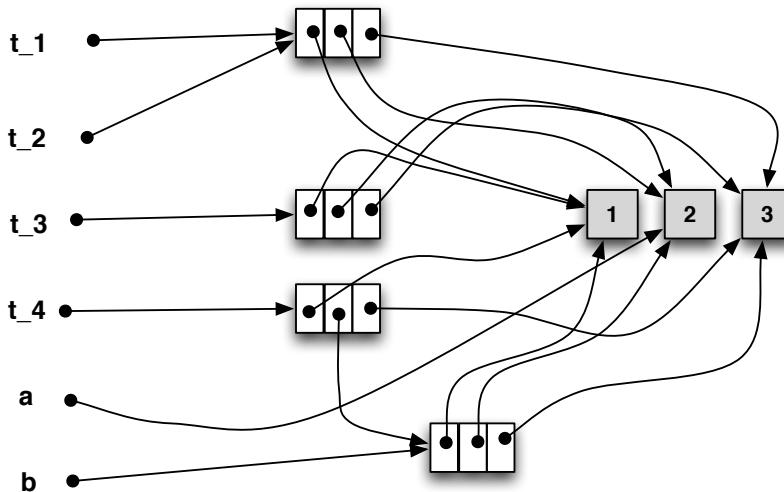


Figura 3.8: Partilha da memória

```

5 >>> id(t_3)
6 4303574512
7 >>> id(t_4)
8 4303574352
9 >>> id(a)
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 como 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 10.19).

Mas se só alterarmos uma parte, o que acontece?

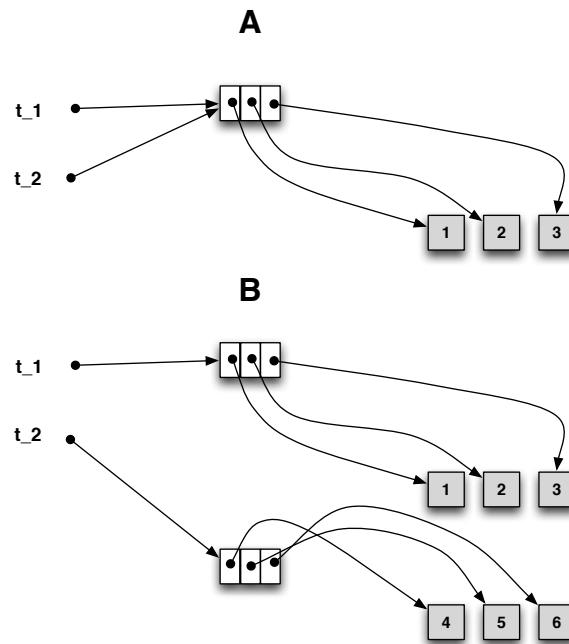


Figura 3.9: Mutabilidade (I): antes (A) e depois (B)

```

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

NoneType

Quando falámos do tipo booleano acentuámos o facto de ser um tipo com apenas dois valores. Existe um outro tipo, **NoneType** que apenas tem **um valor**, designado pela constante **None**. Na realidade **None** designa a ausência

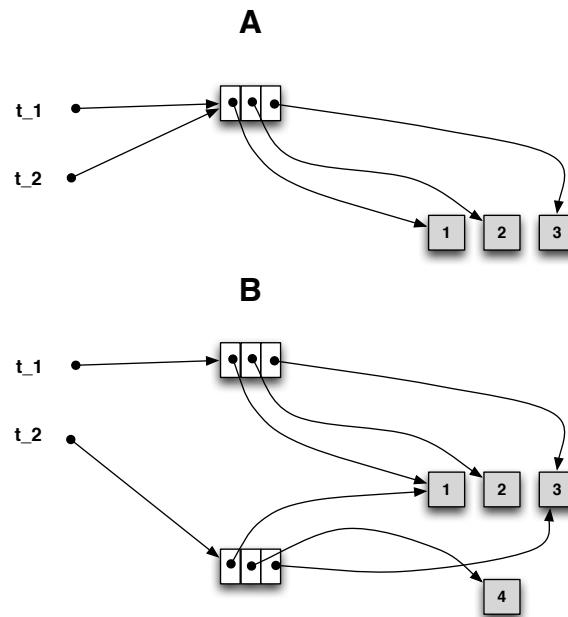


Figura 3.10: Mutabilidade (II): antes (A) e depois (B)

de valor.

```

1 >>> type(None)
2 <class 'NoneType'>
3 >>> id(None)
4 4417401440
5 None
6 >>>
```

`None` pode ser usado com utilidade em várias situações, por exemplo para designar falso :

```

1 if None:
2     print('None')
3 else:
4     print('Hei!')
```

Ao executar este código é impresso 'Hei!'.

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 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 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:

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

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

Escreva um programa que dada uma cadeia de caracteres nos permite obter todos os **prefixos** da cadeia. O exemplo abaixo mostra a saída gerada no caso em que a cadeia é a frase 'Monty Python'.

```
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**. Faça também uma versão genérica que funcione para qualquer cadeia de caracteres.

```
1 n
2 on
3 hon
```

```

4 thon
5 ython
6 Python
7 Python
8 y Python
9 ty Python
10 nty Python
11 onty Python
12 Monty Python

```

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

4.1 Generalidades

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.

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

[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`¹

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 nomes². 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 >>>

```

¹Existem 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

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

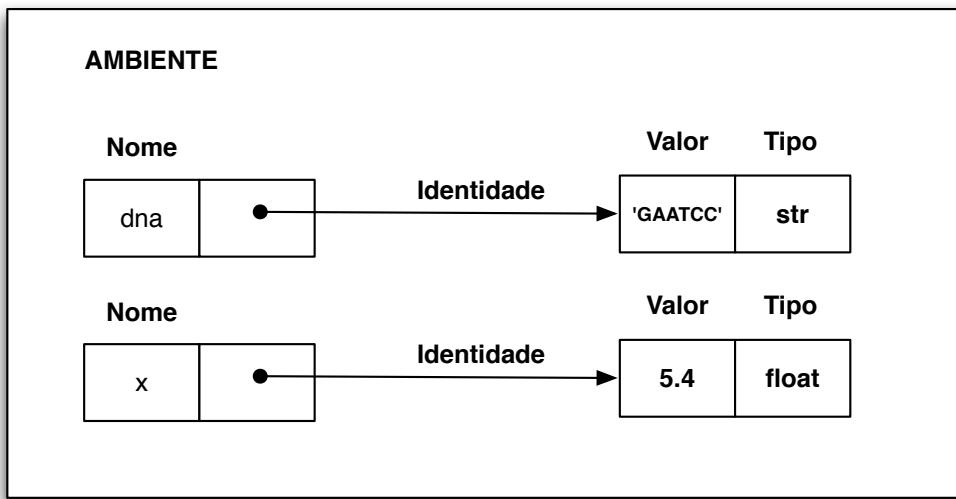


Figura 4.1: Ligação Nomes - Objectos

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 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 objecto³.

```

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>>>

```

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

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.

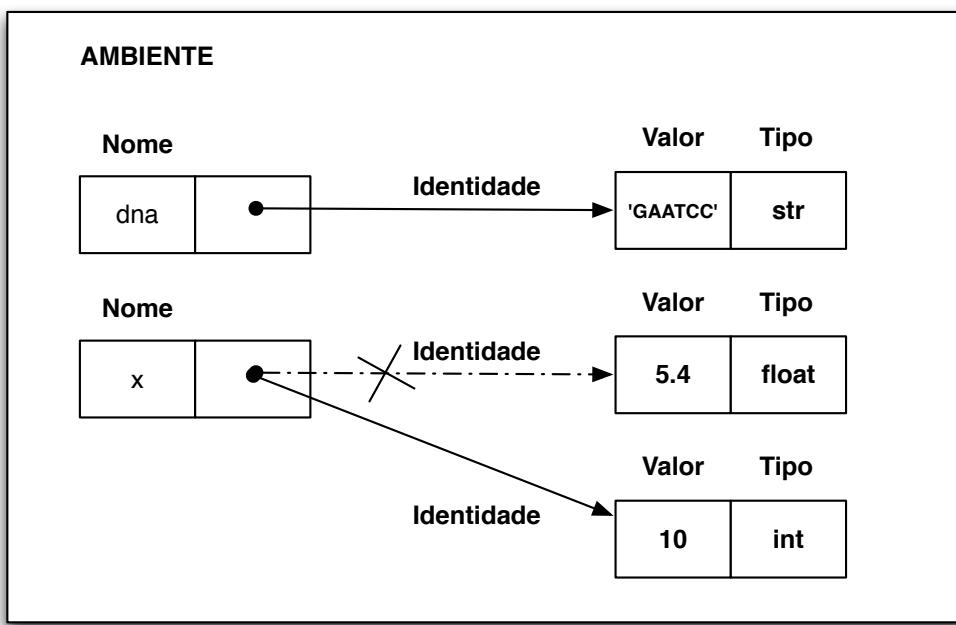


Figura 4.2: Tipagem Dinâmica

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 carácterística⁴.

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)

```

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

```

7 False
8 >>> x
9 10
10 >>> y
11 11
12 >>>

```

A figura 4.3 ilustra o que aconteceu.

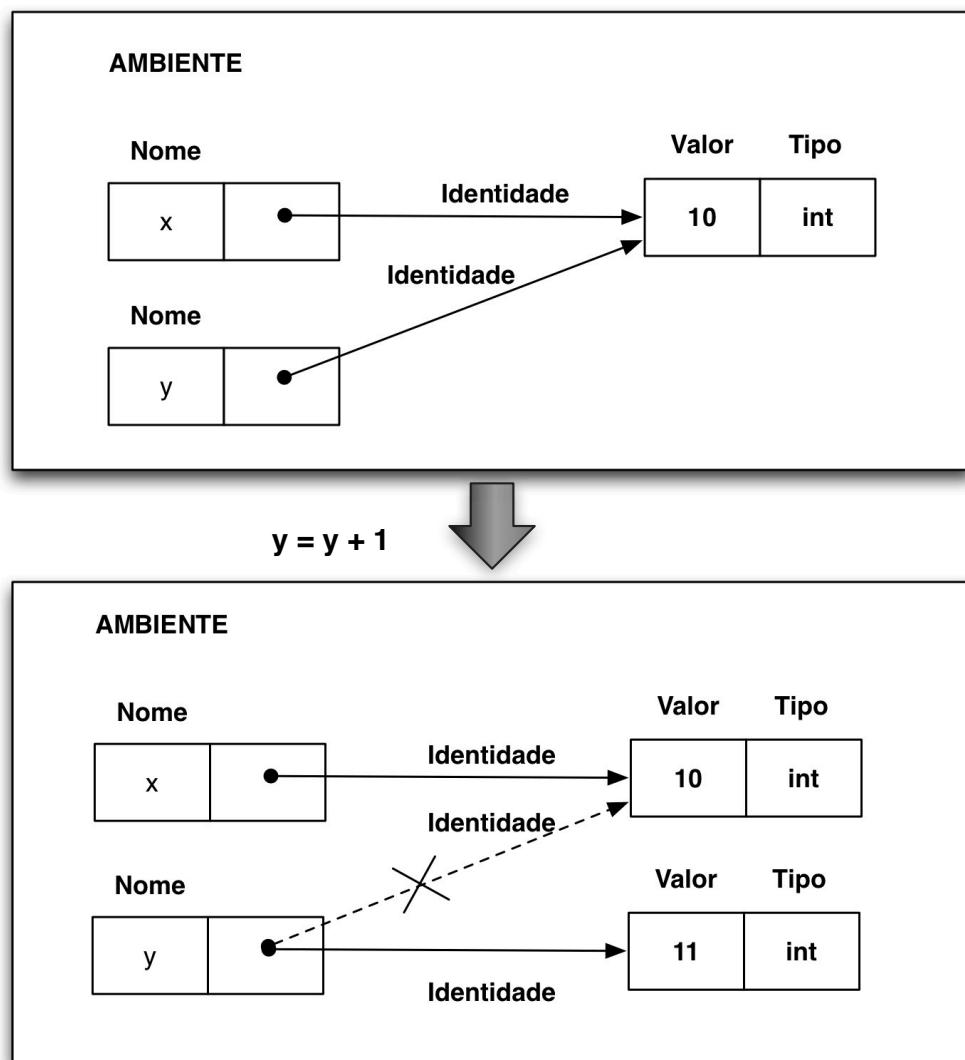


Figura 4.3: Mudança de identidade

Atribuição Aumentada

Existem operações de atribuição em que os objectos aparecem à esquerda e à direita do sinal de atribuição, como se ilustra a seguir.

```

1 >>> x = 4
2 >>> x = x + 1
3 >>> y = 10
4 >>> y = y ** 2
5 >>> x
6 5
7 >>> y
8 100
9 >>>

```

À esquerda do sinal de atribuição o nome remete para o objecto através da sua **identidade**, enquanto que o seu aparecimento à direita do sinal refere-se ao **valor** do objecto. Sendo estas situações muito frequentes a linguagem Python tem uma facilidade sintática que se designa por operador de atribuição aumentado. A sessão anterior podia ser replicada como se segue.

```

1 >>> x = 4
2 >>> x += 1
3 >>> y = 10
4 >>> y **= 2
5 >>> x
6 5
7 >>> y
8 100
9 >>>

```

No efeito final as duas formas são equivalentes. No entanto, esta segunda forma de proceder é mais económica visto objecto ser apenas avaliado uma vez e porque, sempre que possível, não há lugar à criação de um novo objecto. Existem operações de atribuição aumentada para os operadores aritméticos convencionais e ainda para operadores que actuam ao nível do bit, e.g., deslocamento à esquerda e à direita, **e**,**ou**, **xor**. Alguns exemplos.

```

1 >>> x = 4
2 >>> x >>= 1
3 >>> x
4 2
5 >>> x <=> 2
6 >>> x

```

```

7 8
8 >>> x |= 1
9 >>> x
10 9
11 >>> x &= 0
12 >>> x
13 0

```

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
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.

 **Nomes com prefixo ***

Podemos ligar por atribuição os elementos de uma estrutura mas temos que tomar algumas precauções.

```

1 >>> x,y = (5,7)
2 >>> x
3 5
4 >>> y
5 7
6 >>> x,y = (5,7,9)
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 ValueError: too many values to unpack (expected 2)
10 >>>
```

Esta situação pode ser ultrapassada se o nome estiver prefixado por asterisco.

```

1 >>> x,*y = (5,7,9)
2 >>> x
3 5
4 >>> y
5 [7, 9]
6 >>> *x,y = (5,7,9)
7 >>> x
8 [5, 7]
9 >>> y
10 9
11 >>> x,*y,z = (3,5,7,9)
12 >>> x
13 3
14 >>> y
15 [5, 7]
16 >>> z
17 9
18 >>> x,*y,z = (5,7)
19 >>> x
20 5
21 >>> y
22 []
23 >>> z
24 7
25 >>>
```

Os valores prefixados com asterisco ficam associados com um objecto do tipo **lista** de que falaremos no capítulo 6. No último caso **y** fica associado à lista vazia.

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).

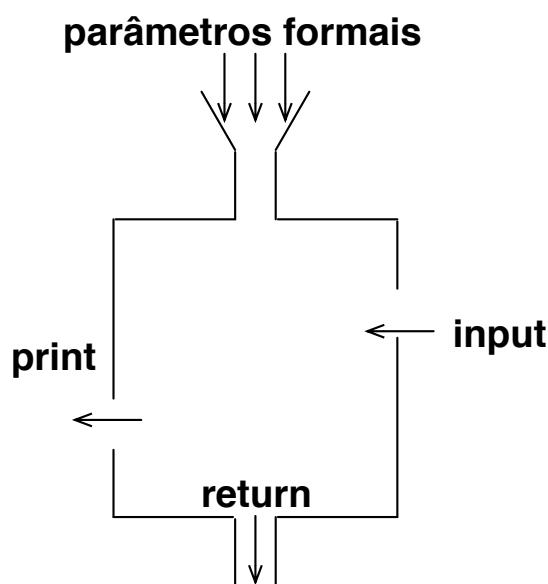


Figura 4.4: Entrada e Saída de dados

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 >>>
```



Chamada de funções

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
...
8 >>> entrada = eval(input('Introduza uma expressão >>> '))
9 Introduza uma expressão >>> duplo(4)
10 >>> entrada
11 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 expressões⁵.

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

```

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

```

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)
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 caracter 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 sobreescarregado `%` 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' % (''
2     'Ernesto', 'Costa'))
3 O primeiro nome é: Ernesto, e o último é: Costa
>>>

```

A cadeia de caracteres à esquerda do operador % pode conter várias marcas de conversão que são precedidas de %⁶.

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

```

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: bandeiras⁷, 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.

⁶Admitimos que este uso múltiplo do sinal % é potencialmente factor de confusão.
⁷do inglês *flags*.

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
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, em-

bora 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)
- ____1
- __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 α, β, γ . Sugestão: obtenha os códigos **unicode** de cada caracter.

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 >>>
```

Inspeccione 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.

	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:

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

E é a energy (em Joules), t_i e t_f as temperaturas inicial e final (em graus Celsius) 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 (Celsius):** 10
- 2** **Temperatura final (Celsius):** 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 (Fahrenheit [-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 tabulações¹. A figura 5.1 ilustra essa situação e a listagem 5.1 mostra um exemplo concreto.

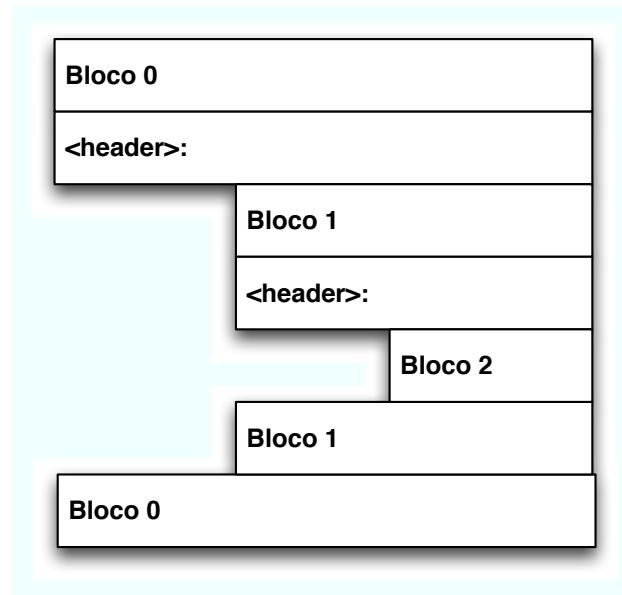


Figura 5.1: Os blocos de um programa

```

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

¹Não 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, o **==** (ou **!=**) que permite determinar se dois objectos têm o mesmo valor (valor diferente), ou ainda o **is** que permite saber se dois nomes estão associados ao mesmo objecto, isto é se têm a mesma identidade (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 >>> y = 10
10 >>> z is w
11 True
12 >>> z is y
13 True
14 >>> x == z
15 False

```

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 5.1².

Tabela 5.1: Representações de Falso

Objecto	Descrição
None	Nada
0	Zero
"	Cadeia de Caracteres vazia
()	Tuplo vazio
[]	Lista vazia
{}	Dicionário vazio

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.

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**.

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

```

1 >>> 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/\ \
8 ... cod.txt')
9 >>> fich
10 <open file '/tempo/cod.txt', mode 'r' at 0x5b4a0>

```

²Listas e dicionários serão tratados mais à frente.

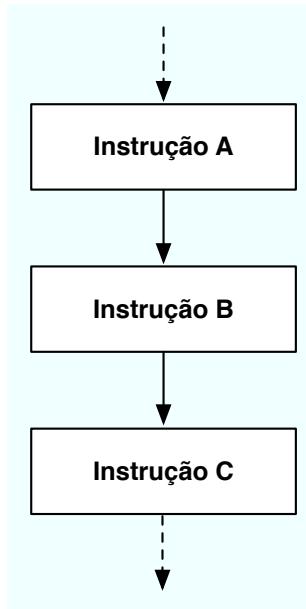


Figura 5.2: Sequência

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 escolhas, 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.

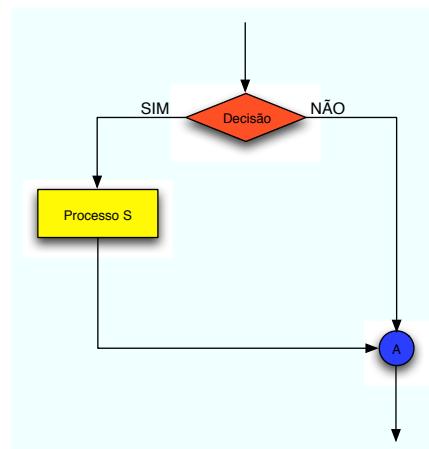


Figura 5.3: A condicional if-then

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= eval(input('temperatura sff:'))
  
```

```

3   if t > 38.5:
4     print('ATENÇÃO: febre!')

```

Outro exemplo:

```

1 def exe2():
2   nome=input('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,
       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 frequente de termos duas alternativas mas ambas com acções a realizar. A figura 5.4 ilustra a ideia.

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 lista-gem 5.5.

```

1 # --- par
2 def exe5(n):

```

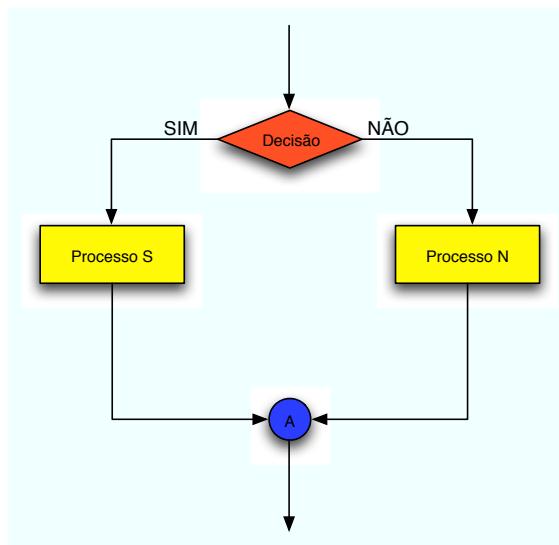


Figura 5.4: A condicional if-then-else

```

3   if (n % 2) == 0:
4       print( "O número %d é par" % n)
5   else:
6       print( "O número %d é ímpar" % n)
7   # -- desconhecido
8 def exe6():
9     nome=input('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     CTTTTCATAGAGCTTGAGAACTGCTGACCTAGGGTTT'
21     comp_primer=len(primer)
22     percent_GC=primer.count('G') + primer.count('C')/comp_primer
23     if percent_GC > 0.50:
24         if comp_primer > 20:
25             programa_PCR=1

```

```

26   else:
27     programa_PCR=2

```

Listagem 5.5: Condicional normal

Condicional Geral Vamos agora generalizar a condicional permitindo um número variável de vias de decisão (ver figura 5.5).

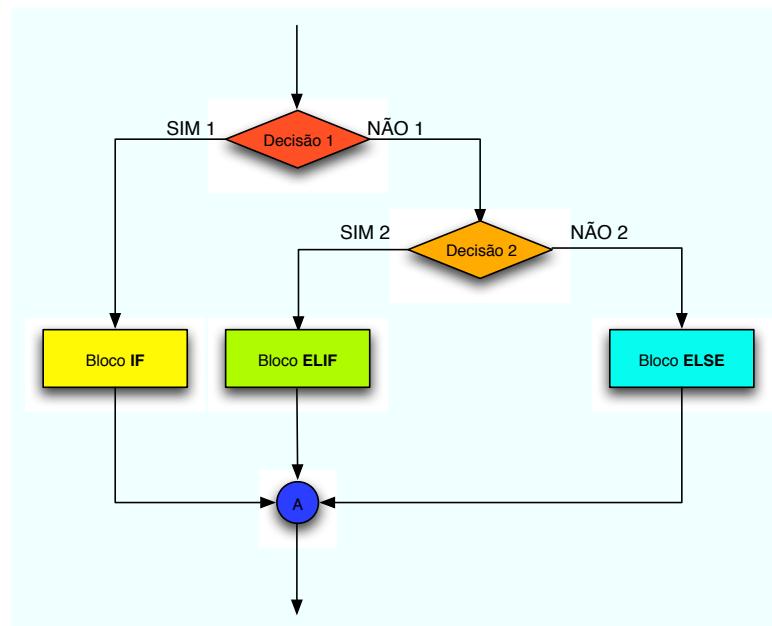


Figura 5.5: A condicional if-elif-else

Genericamente a sintaxe é a seguinte:

```

1  if <condição1>:
2    <instruções1>
3  elif <condição2>:
4    <instruções2>
5  elif <condição3>:
6    <instruções3>
7  ...
8  else:
9    <instruções>

```

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='vATGCAAnATG'
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 5.6: 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 5.7).

```

1 import math
2
3 def main1():
4     """ Calculo das raízes reais de um polinómio."""
5     a,b,c = eval(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=(-b + raiz_discrim) / (2 * a)
16   raiz2=(-b - raiz_discrim) / (2 * a)
17   return raiz1,raiz2

```

Listagem 5.7: 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 5.8).

```

1 def main2():
2     """ Calculo das raízes reais de um polinómio."""
3     a,b,c = eval(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 5.8: 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 5.9).

```

1 def main3():
2     """ Calculo das raízes reais de um polinómio."""
3     a,b,c = eval(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,
r1))

```

```

10  else:
11      print ("As raízes do polinómio de coeficientes\
12          a=%d b=%d c= %d são r1=%3.2f r2=%3.2f" % (a,b,c,r1,r2))
13
14 def raizes3(a,b,c):
15     """ Calcula raízes reais."""
16     discriminante= pow(b,2) - 4 * a * c
17     if discriminante < 0:
18         return None,None
19     elif discriminante == 0:
20         raiz1 = raiz2 = -b/ (2 * a)
21         return raiz1, raiz2
22     else:
23         raiz_discrim = math.sqrt(discriminante)
24         raiz1=(-b + raiz_discrim) / (2 * a)
25         raiz2=(-b - raiz_discrim) / (2 * a)
26         return raiz1,raiz2

```

Listagem 5.9: 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 socorrermos-nos do módulo **cmath** (ver listagem 5.10).

```

1 import cmath
2
3 def main4():
4     """ Cálculo das raízes de um polinómio."""
5     a,b,c = eval(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=(-b + raiz_discrim) / (2 * a)
25         raiz2=(-b - raiz_discrim) / (2 * a)
26         return raiz1,raiz2
27     elif discriminante == 0:
28         raiz1 = raiz2 = -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 5.10: 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 5.11). Para este tipo de problemas as linguagens de programação disponibilizam estruturas de controlo repetitivas, também conhecidas por ciclos.

A listagem 5.11 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 5.11: 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**³.

for

Em aplicações simples os ciclos **for** são executados um número fixo e pré-

[Ciclos for](#)

³Existe uma outra forma de repetição designada por **recursividade** e que será discu-

definido de vezes. Por exemplo, o exemplo da listagem 5.11 pode ser reescrito conforme mostramos na listagem 5.12. 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 5.12: 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 5.13..

```

1 def exec2(lado,angulo,vezes):
2     pd()
3     for i in range(vezes):
4         fd(lado)
5         rt(angulo)
6     ht()
```

Listagem 5.13: 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 <iterável>:
2     <instrução>
```

O iterável é qualquer objecto composto. Por exemplo, pode ser uma cadeia de caracteres ou um tuplo⁴. Antes de executar a <instrução> o <nome> assume o primeiro elemento do iterável. Depois de executada a <instrução> toma o seu segundo valor, e assim sucessivamente. A figura 5.6 ilustra o conceito.

Um exemplo simples do uso do ciclo **for** é ilustrado na listagem 5.14.

```

1 # No módulo
```

tida no capítulo 9.

⁴Com a descoberta de novos objectos nos capítulos seguintes veremos que há uma grande variedade de objectos que pode assumir o papel de objecto iterável.

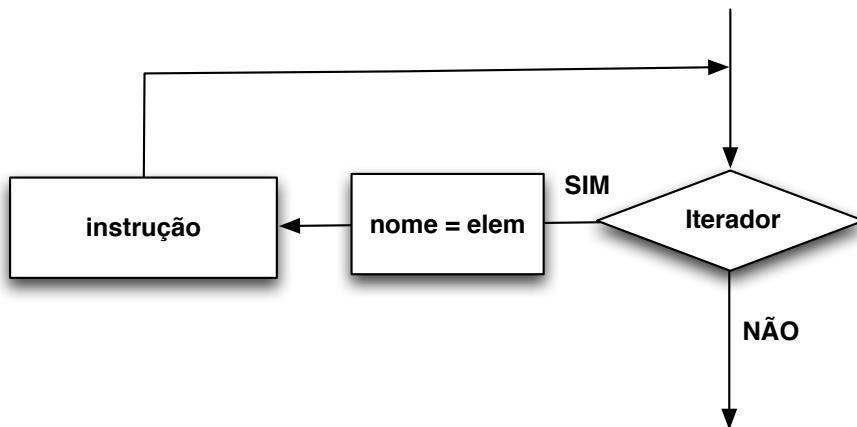


Figura 5.6: Ciclo for

```

2 def exec4(seq):
3     for ch in seq:
4         print(ch)
5
6 # no interpretador
7 >>> exec4('abc')
8 a
9 b
10 c
11 >>>
  
```

Listagem 5.14: 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`.

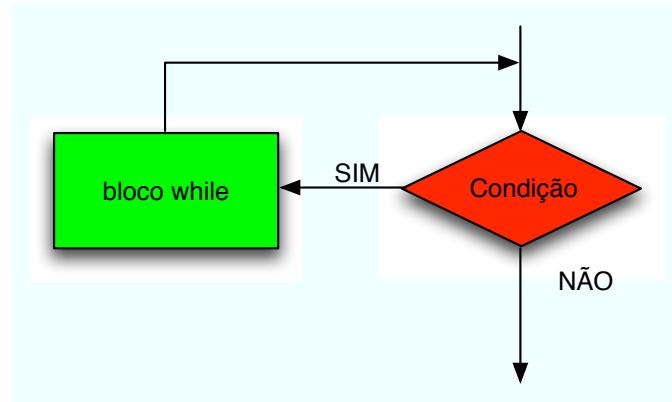


Figura 5.7: O ciclo `while`

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 5.15 mostra um possível programa⁵.

```

1 def main():
2     numero=eval(input("Número por favor:\t"))
3     while numero <=0:
4         print ("\nO número tem que ser positivo!")
5         numero=eval(input("Número por favor:\t"))
6     print (" Número é igual a %d" % numero)

```

Listagem 5.15: 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=input("Entre a palavra sff:\t")

```

⁵Não 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!

```

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=eval(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 quantas vezes a sequência **TATA**⁶ 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 se soluções tipo, e estão na origem do aparecimento dos chamados **padrões de desenho**⁷. Suponhamos que nos pedem

Padrões de Desenho

⁶Conhecida por *TATA box* na literatura inglesa.

para calcular a soma dos primeiros n números inteiros positivos, isto é:

$$\sum_{i=1}^n i$$

Uma maneira de resolver o problema é dispor dos números em sequência e ir efectuando as sucessivas adições: somamos 1 com 2, o resultado é somado com o número seguinte, 3, o resultado é adicionado 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.

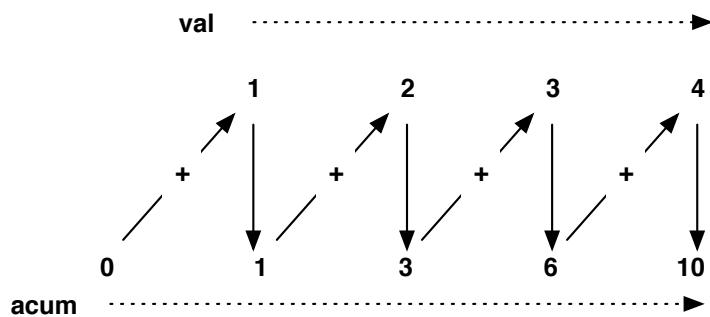


Figura 5.8: O Padrão Acumulador

Posto isto o programa que resolve o nosso problema é trivial:

```

1 def padrao_1(n):
2     acum = 0
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

⁷Do inglês *Design Patterns*.

noutro tipo de problemas essa contagem seja explícita e, nalguns casos, essa contagem seja o valor que queremos saber.

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 **invariantes de ciclo**. Veremos num capítulo mais à frente como os invariantes de ciclo nos podem ajudar a mostrar a textbfcorrecção 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 π ?

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 aproxima-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 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 π . O número π 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 métodos⁸. Mas antes de discutir a solução informática importa dizer que desde há longos séculos se procurou encontrar um valor aproximado para π . A listagem abaixo mostra algumas das tentativas.

```

1 >>> # Babilónia
2 >>> 3 + 1/8
3 3.125
4 >>> # Antigo Egípto
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

```

⁸Não 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.

```

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 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 5.16: π 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 / (i-1)
8         direita = i / (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 * ((2 * i) ** 2) / (((2 * i) ** 2 - 1))
8     return 2 * acum

```

Comparando esta versão com a última baseada na fórmula de Leibniz fica patente a semelhança dos dois **programas** que se baseiam no mesmo padrão que recorre a um acumulador..

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).

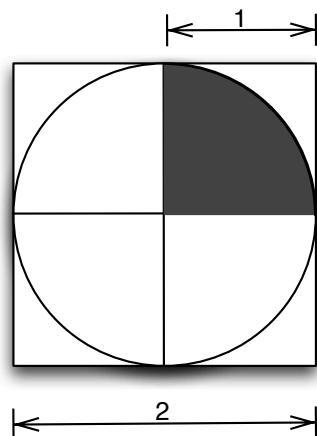


Figura 5.9: Simulação de monte Carlo: o caso de π

Por outro lado sabemos que a área da circunferência é igual a $\pi \times r^2 = \pi^9$. Logo, a área da parte da circunferência que cobre cada quadrado pequeno vale $\frac{\pi}{4}$. O método de Monte Carlo¹⁰, é um método estocástico inicialmente proposto por Stanislaw Ulam e Nicholas Metropolis, que pode ser usado para calcular o valor aproximado de π . Neste caso, consiste em simular o lançamento de dardos na direção 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 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 1¹¹ 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.

⁹Pois o raio é 1.

¹⁰O 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.

¹¹Porque o raio da circunferência é igual a 1!

```

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 5.17.

```

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 5.17: Método de Monte Carlo animado

Como se pode ver, alterámos ligeiramente o programa que calcula o valor de π , 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.

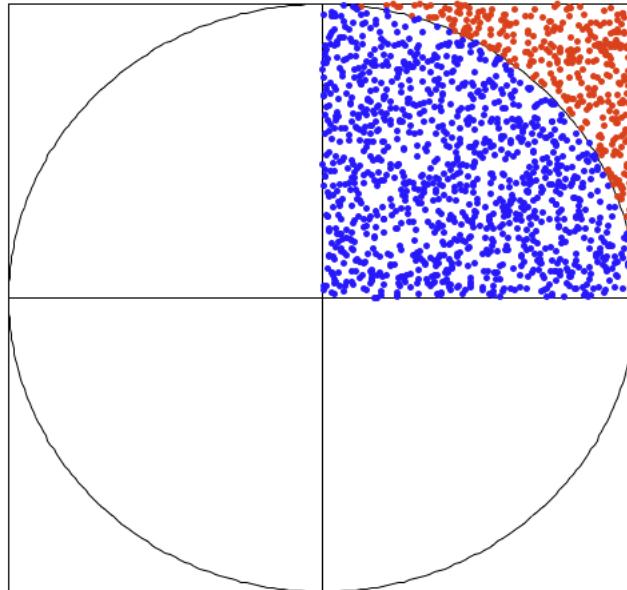


Figura 5.10: Monte Carlo animado

Simulações

O método de Monte Carlo pode ser usado em diferentes situações. Suponhamos que queremos calcular a área sob um curva, entre dois pontos determinados¹². Para ser mais concreto, admitamos que se trata da função e^{-x^2} cujo gráfico se apresenta na figura 5.11.

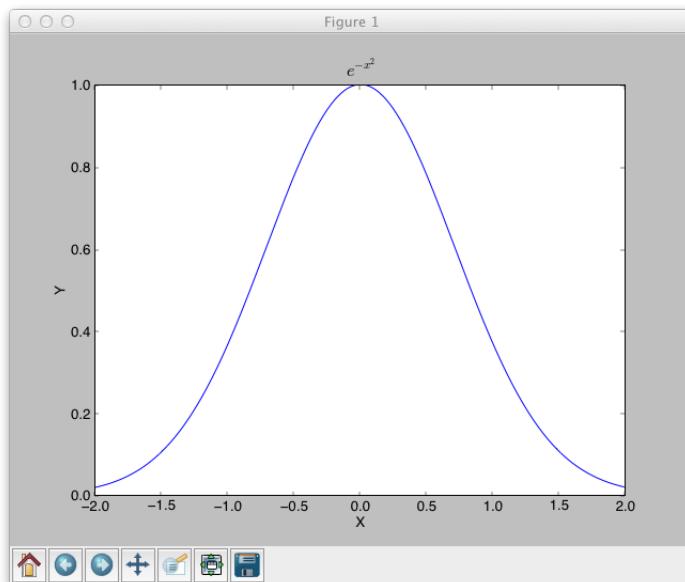


Figura 5.11: A função e^{-x^2}

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):
9         x = random.uniform(a,b)
```

¹²O leitor reconhecerá que se pretende calcular o integral da função entre dois pontos.

```

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 5.18: 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 ciclo¹³.

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 5.19.

```

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 5.19: Break: exemplo de uso

¹³Este 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*.

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 5.20 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,
9             idade))

```

Listagem 5.20: Ciclos *potencialmente* infinitos

Num último exemplo, reproduzido na listagem 5.21, procuramos o maior número a partir de um certo limite que é um quadrado perfeito.

```

1 import math
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=math.sqrt(num)
7         if raiz == int(raiz):
8             print ("Maior quadrado perfeito menor do que %d é %d" %
9                 (n,num))
10            break

```

Listagem 5.21: 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 5.22 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 5.22: Continue: ímpares

O exemplo seguinte (listagem 5.23) 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 5.23: 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 5.19 podemos ver como se pode usar essa possibilidade para encontrar números primos, como ilustra a listagem 5.24.

```

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 5.24: Else: números primos

Vejamos mais um exemplo de introdução controlada de um código. No exemplo da listagem 5.25 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 5.25: 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 decido...
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 5.26 ilustra o caso simples em que isolamos a situação de uma divisão por zero.

```

1 def try_1(x):
```

```

2   try:
3       y=eval(input("\nDenominador: "))
4       print(x / y)
5   except ZeroDivisionError:
6       print("\nCuidado: o denominador não pode ser zero!")

```

Listagem 5.26: 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 proteção de dados de entrada.

```

1 def try_2():
2     """ Exemplo de uso de pass.
3     Espera interrupção pelo keyboard.
4     """
5
6     while True:
7         try:
8             x=int(input("Um numero sff:\t"))
9             break
10        except ValueError:
11            pass
12    print "\nFinito!"

```

Regressando ao nosso exemplo das raízes de uma equação do segundo grau vejamos (listagem 5.27) como podemos proteger o programa que apenas prevê raízes reais.

```

1 def try_3():
2     """ Cálculo das raízes reais de um polinómio.
3     """
4
5     try:
6         a,b,c = eval(input("Os coeficientes sff (a,b,c):\t"))
7         discriminante=pow(b,2)- 4 * a * c
8         raiz_discrim = math.sqrt(discriminante)
9         raiz1= (-b + raiz_discrim) / (2 * a)
10        raiz2 = (-b - raiz_discrim) / (2 * a)
11        if raiz1 == raiz2:
12            print ("O polinómio de coeficientes\
13                a=%d b=%d c= %d tem raízes multiplas raiz1= raiz2=%3.2f
14                " % (a,b,c,raiz1))
15        else:
16            print( "As raízes do polinómio de coeficientes\

```

```

15     a=%d b=%d c= %d sao raiz1=%3.2f raiz2=%3.2f" % (a,b,c,
16             raiz1,raiz2))
17 except ValueError:
    print ("\n Não tem raízes reais!")

```

Listagem 5.27: Try: raízes reais apenas

Um exemplo clássico do uso de excepções 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.

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 5.28.

```

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 5.28: Excepções: 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

```

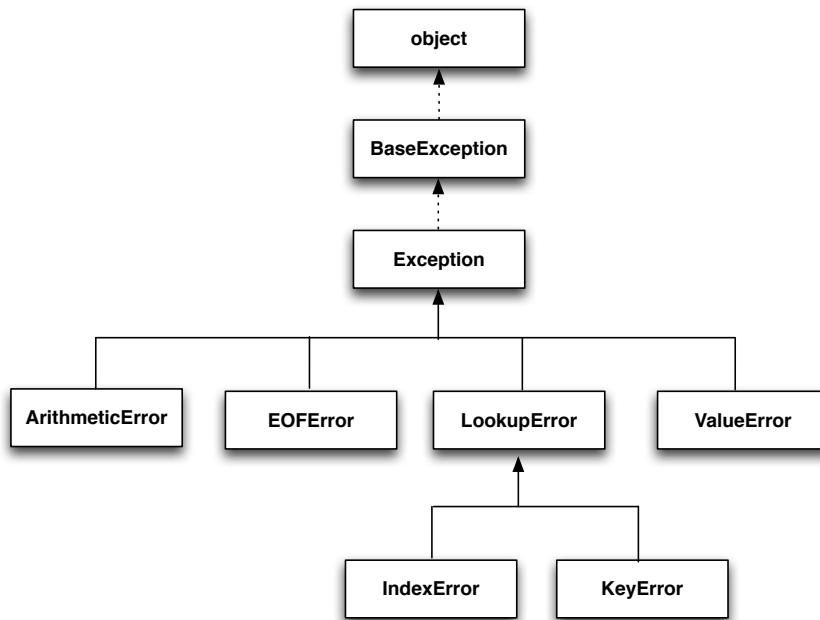


Figura 5.12: Hierarquia de exceções (visão parcial)

```

2
3 try:
4     f = open('myfile.txt')
5     s = f.readline()
6     i = int(s.strip())
7 except IOError as erro:
8     print("I/O error: %s" % erro)
9 except ValueError:
10    print("Os dados não puderam ser convertidos para inteiros
11    .")
11 except:
12     print("Erro inesperado:", sys.exc_info()[0])
13     raise
  
```

Neste exemplo chamamos a atenção para o último **except** onde não existe nenhuma exceção definida associada e é levantada (**raise**) uma geral. Pode ainda aparecer uma cláusula **else**, sempre a seguir às exceções, e que será executada se nada de anormal se passar com o **try**. No exemplo abaixo estaremos 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 5.29

```

1  >>> def divide(x, y):
2      ...
3          try:
4              result = x / y
5          except ZeroDivisionError:
6              print ("divisão por zero!")
7          else:
8              print( "o resultado é", result)
9          finally:
10             print( "a executar a cláusula finally")
11 ...
12
13 >>> divide(2, 1)
14 o resultado é 2
15 a executar a cláusula finally
16 >>> divide(2, 0)
17 divisão por zero!
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 5.29: 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 5.30.

```
1 if <condição>:  
2     <aborta_programa>
```

Listagem 5.30: Teste para abortar programa

É mais correcto e genérico usar o comando **assert**.

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 AssertionError: 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 subtracção 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 5.31: 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(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 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 Módulo random 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.

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:

$$\text{fact}(n) = n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

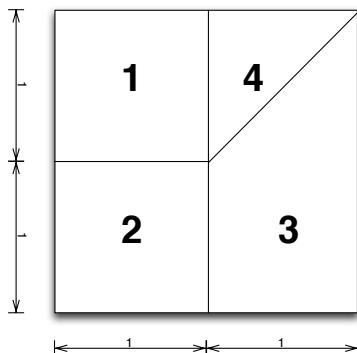


Figura 5.13: Calcular probabilidades

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 harmônico 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 harmônicos 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 harmônicos 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, excluindo ele próprio. 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.

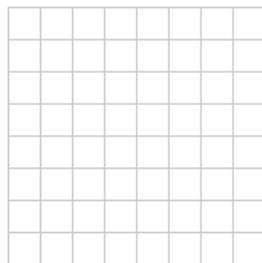


Figura 5.14: Desenho de uma grelha

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).

Exercício 5.21 D

A 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

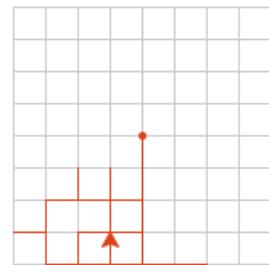


Figura 5.15: Passeio aleatório

sequência é igual à soma dos dois elementos imediatamente anteriores. Eis os primeiros números da sequência:

$^1 \quad 1, 1, 2, 3, 5, 8, 13, 21, \dots$

Escreva um programa que dado um número **verifica** se ele pertence ou não à sequência de Fibonacci.

Objectos (II)

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.

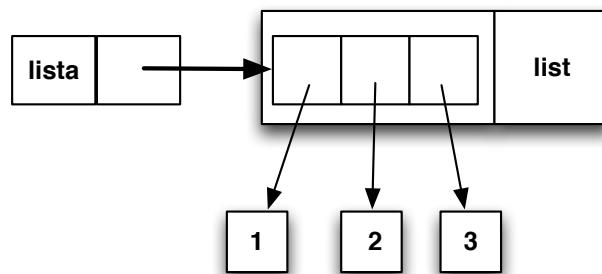


Figura 6.1: A representação de uma lista simples

Se agora alterar o segundo elemento de 2 para 4 a nova situação é retratada na figura 6.2.

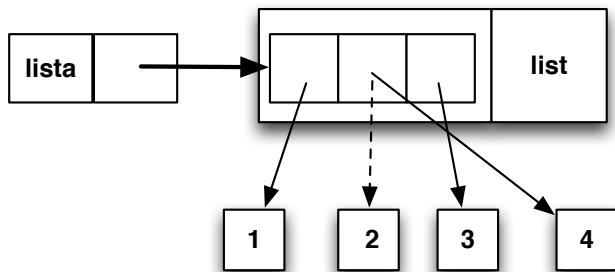


Figura 6.2: A lista alterada

Como se vê pela imagem a referência de toda a lista, isto é a sua identidade, mantém-se! É óbvio que listas mais complexas têm representações ... mais complexas. A figura 6.3 mostra de modo simplificado¹ o caso da lista `nova_lista = [1, [2,3,4], [5,6]]`.

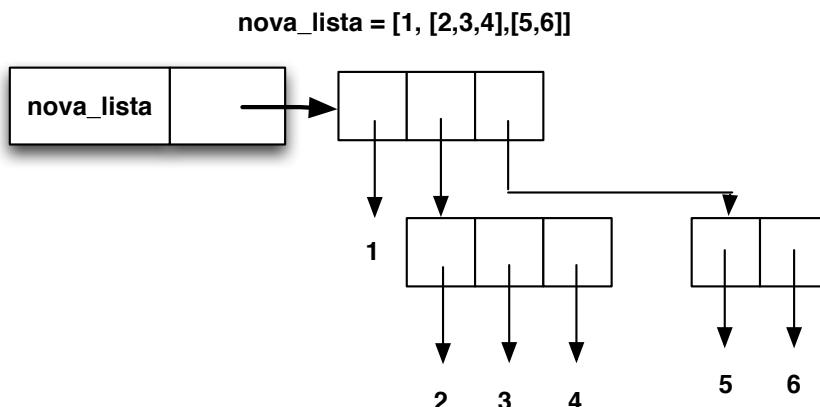


Figura 6.3: Uma lista mais complexa

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)
  
```

¹Sempre que julgarmos conveniente e não afectar o entendimento, usaremos uma notação gráfica simplificada

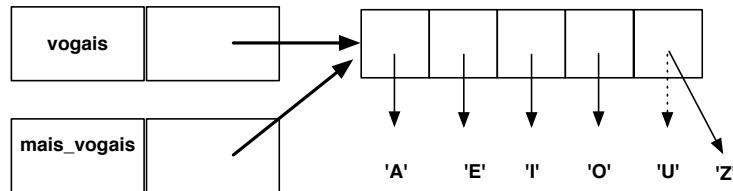
```

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 uma **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).

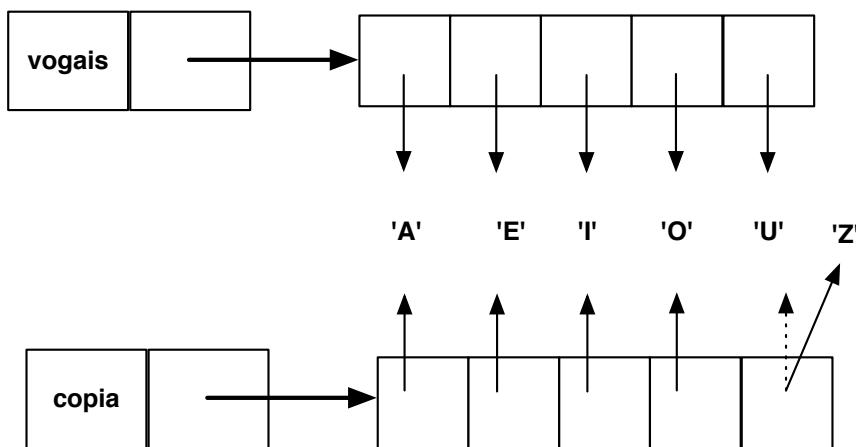


Figura 6.5: Alterar sem efeitos não desejados

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.

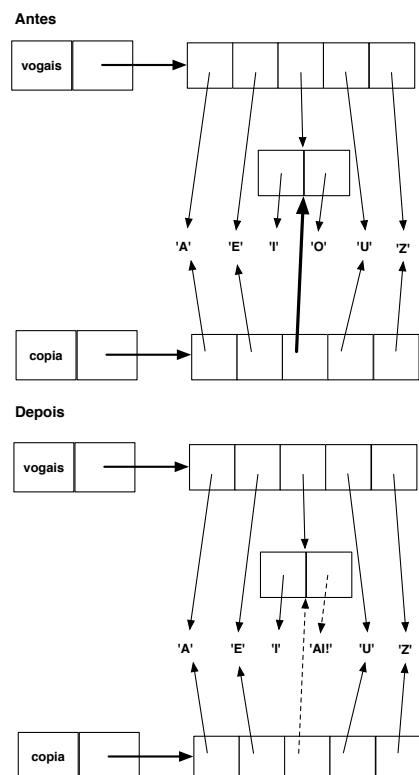


Figura 6.6: Tudo se complica...

Cópia profunda

Para termos a certeza que não temos nenhuma surpresa devemos fazer uma **cópia profunda**² 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)

```

²do inglês *deepcopy*.

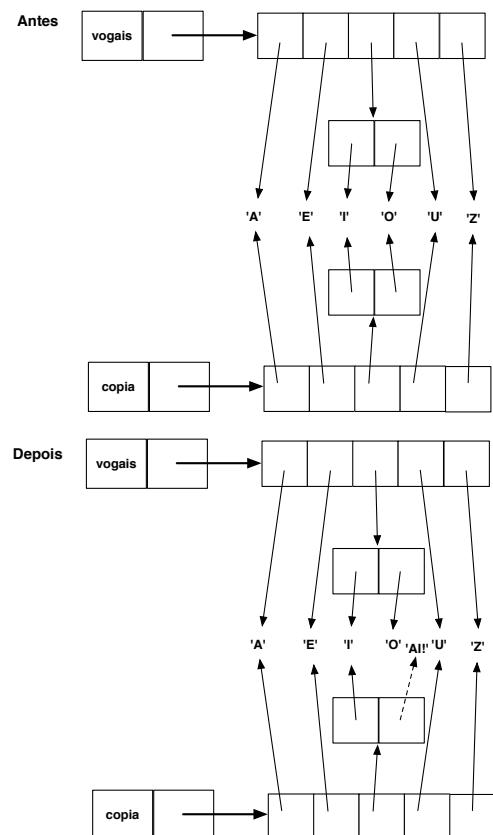


Figura 6.7: Sem problema...

```

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 não³.

```

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')

```

³Na realidade devolvem o objecto `None`, e esse facto é fonte de muitas surpresas desagradáveis.

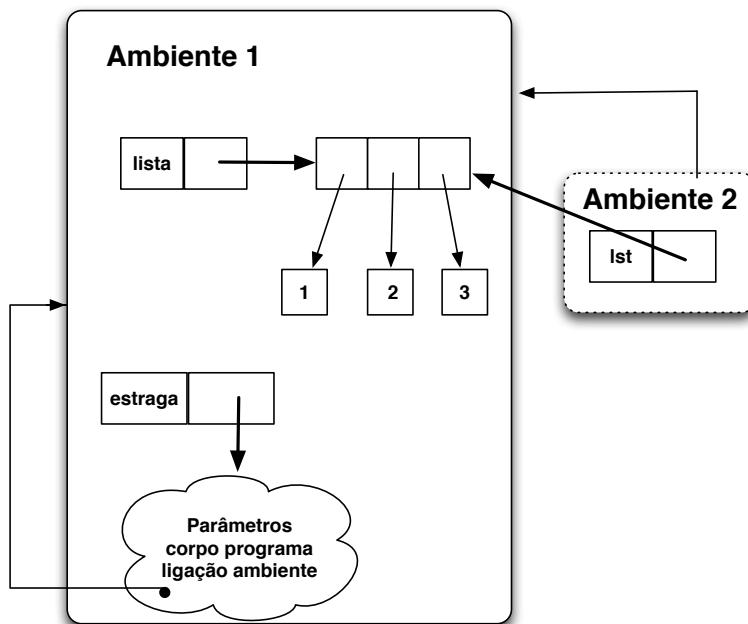


Figura 6.8: Ambientes, objectos e nomes

Método	Operação
Não Modificam	
<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>
Modificam	
<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` acrescenta 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 matriz⁴ de um modo muito elegante.

⁴A transposta de uma matriz é o que se obtém quando o elemento na posição (i,j) vai para a posição (j,i).

⁵Existe 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 6.6⁶.

```

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 temos 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:

⁶É 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
2 plt = matplotlib.pyplot
3
4 def mostra(lista):
5     """Gráfico simples de uma lista de valores."""
6     plt.xlabel('Dias')
7     plt.ylabel('Quantidade')
8     plt.title('Baleias')
9     plt.plot(lista)
10    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

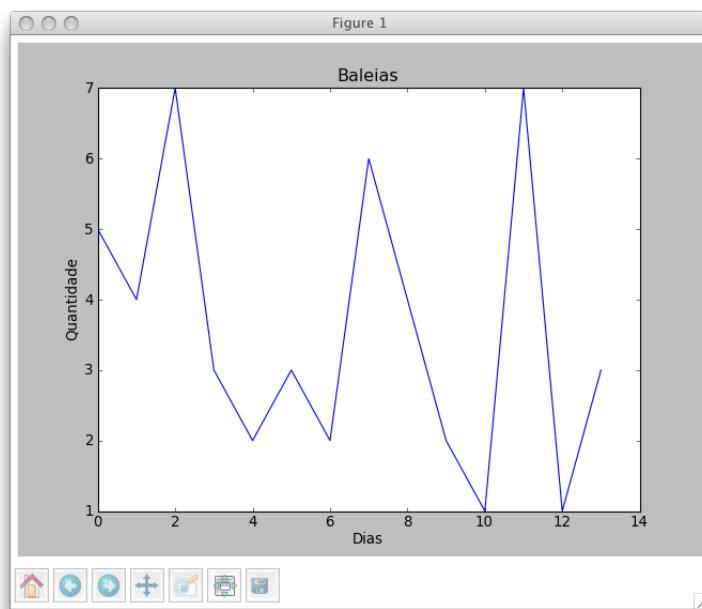


Figura 6.9: As Baleias dia a dia

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 obtêm-se graças ao código genético (ver figura 6.10): a cada três bases⁷ corresponde um aminoácido .

São as sequências de aminoácidos codificados nos genes que originam as proteínas⁸. A figura 6.11 ilustra este processo.

Mas o que é que tudo isto pode ter que ver com a programação, sinto o

⁷Cada triplete é designado por codão

⁸Todo 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 letter of codon (5' end)	U	UUU	Phe	UCU	Ser	UAU	Tyr	UGU	Cys
	U	UUC	Phe	UCC	Ser	UAC	Tyr	UGC	Cys
	U	UUΑ	Leu	UCA	Ser	UAA	Stop	UGΑ	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	CUΑ	Leu	CCA	Pro	CAA	Gln	CGΑ	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	AUΑ	Ile	ACA	Thr	AAA	Lys	AGΑ	Arg
	A	AUG	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	GUΑ	Val	GCA	Ala	GAA	Glu	GGΑ	Gly
	G	GUG	Val	GCG	Ala	GAG	Glu	GGG	Gly

Figura 6.10: Código Genético

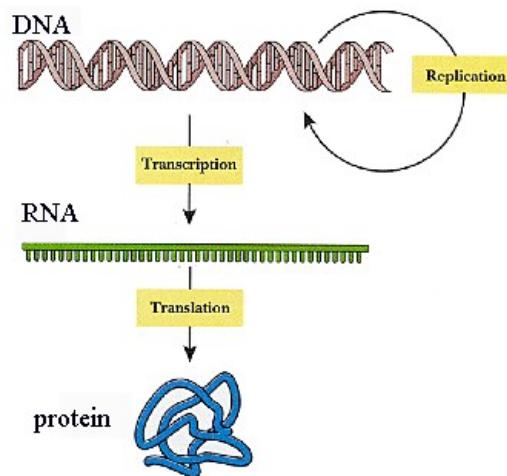


Figura 6.11: Expressão Genética

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 ideia 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 cadeias 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 colecção 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 as chavetas. Como nas listas, as vírgulas separam os elementos. Cada elemento

Definição

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
<i>Elimina</i>	<i>del dict[key]</i>	Retira o item associado a <i>key</i>

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 = {'A': 'Adenina', 'T': 'Timina'}
2 >>> d
3 {'A': 'Adenina', 'T': 'Timina'}
4 >>> d['A']
5 'Adenina'
6 >>> d['C']
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 KeyError: 'C'
```

```

10  >>> 'A' in d
11  True
12  >>> len(d)
13  2
14  >>> del d['T']
15  >>> d
16  {'A': 'Adenina'}

```

Tentar obter um valor através de uma chave inexistente dá erro. A heterogeneidade existe nas chaves e nos valores. Sendo mutável, podemos não apenas consultar como também modificar o valor sem alterar a identidade, como o exemplo seguinte ilustra.

```

1  >>> dicio = {'A': 'Adenina', 2:'dois', 4.6: 'quatro ponto seis',
2      5+4.5j: 'complexo'}
3  >>> dicio
4  {'A': 'Adenina', (5+4.5j): 'complexo', 2: 'dois', 4.6: 'quatro
5      ponto seis'}
6  >>> id(dicio)
7  140184927536800
8  >>> dicio['A'] = 'Académica'
9  >>> dicio
10 >>> id(dicio)
11 140184927536800

```

Métodos

Para além das operações básicas , existe um conjunto de métodos relevantes, uns de consulta outros de modificação, como a tabela 6.6 ilustra.

Comecemos por ilustrar o funcionamento dos métodos que não alteram o dicionário.

```

1  >>> dicio = {'A': 'Adenina', 'C': 'Citosina', 'G': 'Guanina', '
2      'T': 'Timina'}
3  >>> id(dicio)
4  4558053520
5  >>> copia_dicio = dicio.copy()
6  >>> copia_dicio
7  {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina
8      '}
9  >>> id(copia_dicio)
10 >>> 4558096032

```

Método	Operação
Não Modificam	
<i>dict.copy()</i>	Devolve uma cópia do <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>
Modificam	
<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.popitem()</i>	Retira aleatoriamente e devolve um par
<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>

Tabela 6.6: Métodos Pré-Definidos para Dicionários

```

9  >>> dicio.items()
10 dict_items([('A', 'Adenina'), ('C', 'Citosina'), ('T', 'Timina',
11   ), ('G', 'Guanina')])
12 >>> dicio.keys()
13 dict_keys(['A', 'C', 'T', 'G'])
14 >>> dicio.values()
15 dict_values(['Adenina', 'Citosina', 'Timina', 'Guanina'])
16 >>> list(dicio.items())
17 [(('A', 'Adenina'), ('C', 'Citosina'), ('T', 'Timina'), ('G', 'Guanina'))]
18 >>> dicio.get('T')
19 'Timina'
20 >>> dicio.get('U')
21 >>> dicio.get('U', 'Oops!')
'Oops!'

```

Os métodos que alteram o dicionário obrigam a alguns cuidados. Eis alguns exemplos de utilização.

```

1 >>> dicio.clear()
2 >>> dicio
3 {}
4 >>> id(dicio)
5 4558053520
6 >>> dicio = {'A': 'Adenina', 'C': 'Citosina', 'G': 'Guanina', 'T': 'Timina'}

```

```

    T': 'Timina'}
7  >>> dicio.pop('T')
8  'Timina'
9  >>> dicio
10 {'A': 'Adenina', 'C': 'Citosina', 'G': 'Guanina'}
11 >>> dicio['U'] = 'Uracil'
12 >>> dicio
13 {'A': 'Adenina', 'C': 'Citosina', 'U': 'Uracil', 'G': 'Guanina'
   '}
14 >>> dicio.popitem()
15 ('A', 'Adenina')
16 >>> dicio
17 {'C': 'Citosina', 'U': 'Uracil', 'G': 'Guanina'}
18 >>> dicio_2 = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}
19 >>> dicio.update(dicio_2)
20 >>> dicio
21 {'A': 'T', 'C': 'G', 'U': 'Uracil', 'T': 'A', 'G': 'C'}
22 >>> dicio.setdefault('Bases', ['A', 'T', 'C', 'G'])
23 ['A', 'T', 'C', 'G']
24 >>> dicio
25 {'A': 'T', 'C': 'G', 'Bases': ['A', 'T', 'C', 'G'], 'U': 'Uracil',
   'T': 'A', 'G': 'C'}
26 >>>

```

Por vezes existe alguma confusão entre os métodos *get* e *setdefault*. A grande diferença é que o primeiro não altera o dicionário, faz apenas uma consulta, enquanto que o segundo no caso de não existir a chave no dicionário cria uma nova entrada.

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  >>> dicio
2  {'A': 'Adenina', 'C': 'Citosina', 'T': 'Timina', 'G': 'Guanina'
   '}
3  for elem in dicio:
4      print(elem)
5
6  A
7  C
8  T

```

```

9  G
10 >>> for c in dicio.keys():
11     ...     print(c)
12 ...
13 A
14 C
15 T
16 G
17 >>> for v in dicio.values():
18     ...     print (v)
19 ...
20 Adenina
21 Citosina
22 Timina
23 Guanina
24 >>> for c,v in dicio.items():
25     ...     print ('d[',c,'] = ',v)
26 ...
27 d[ A ] = Adenina
28 d[ C ] = Citosina
29 d[ T ] = Timina
30 d[ G ] = Guanina
31 >>>

```

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"
2  >>> A = seq.count("A")
3  >>> B = seq.count("B")
4  >>> C = seq.count("C")
5  >>> D = seq.count("D")

```

	A	T	C	G
A	Green			
B		Orange	Orange	Orange
C			Green	
D	Orange	Orange		Orange
G				Green
H	Orange	Orange	Orange	
K		Yellow		Yellow
M	Yellow		Yellow	
N	Purple	Purple	Purple	Purple
R	Yellow			Yellow
S			Yellow	Yellow
T		Green		
V	Orange		Orange	Orange
Y		Yellow	Yellow	
W	Yellow	Yellow		

Figura 6.12: Códificação das Bases

```

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, "
      H =", H, "K =", K, "M =", M, "N =", N, "R =", R, "S =", S,
      "T =", T, "V =", V, "W =", W, "Y =", Y)
18 A = 4 B = 0 C = 2 D = 0 G = 0 H = 0 K = 3 M = 1 N = 0 R = 3 S
      = 0
19 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 "ABCDGHKMRSTVWY":
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':
8      1, 'N': 0, 'S': 0, 'R': 3, 'T': 2, 'W': 1, 'V': 0, 'Y': 0}
9  >>>

```

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',
16        'AGU': 'S', 'AGC': 'S', 'AGA': 'R', 'AGG': 'R',
17        'GUU': 'V', 'GUC': 'V', 'GUA': 'V', 'GUG': 'V',
18        'GCU': 'A', 'GCC': 'A', 'GCA': 'A', 'GCG': 'A',
19        'GAU': 'D', 'GAC': 'D', 'GAA': 'E', 'GAG': 'E',
20        'GGU': 'G', 'GGC': 'G', 'GGA': 'G', 'GGG': 'G'
21    }
22    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':'gato56'}
6     if nome_utilizador not in codigos :
7         return 'Não o conheço!'

```

```

8     codigo_correcto = codigos[nome_utilizador]
9     if segredo == codigo_correcto:
10        return 'Bem vindo!'
11    else:
12        return 'Enganou-se'
13

```

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 com dicionário com a frequência da ocorrência
3         dos valores. """
4     freq = dict()
5     for item in valores:
6         freq[item] = freq.get(item,0) + 1
7     return freq
8

```

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
13

```

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

Um pouco mais complexo é encontrar o progenitor de uma dada pessoa:

```
1 def progenitor(arv_gen, pessoa):
2     for c,v in arv_gen.items():
3         if pessoa in v:
4             return c
5     return []
```

Listagem 6.13: Genealogia III

Com base neste modelo ultra-simplificado irão ser apresentados como exercícios alguma questões sobre árvores genealógicas.

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 todas 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 6.15 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 6.14: Pares e Ímpares

Exercício 6.3 F

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 6.15 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 6.15: alterna

Exercício 6.4 F

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 6.16 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 6.16: 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 M

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 6.13 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**.

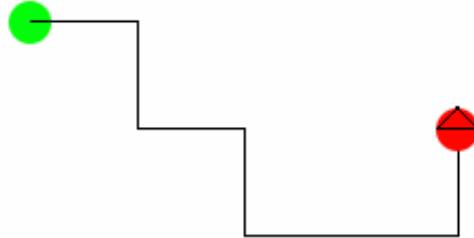


Figura 6.13: Passeando na cidade

```

1 def main_tarta(n):
2     tartaruga = turtle.Turtle()
3     comandos = gera_comandos(n)
4     navega(comandos, tartaruga)
5     turtle.exitonclick()

```

Vai ter que implementar, para além de navega, a função `gera_comandos(n)` que gera uma sequência válida de comandos de tamanho `n`.

Exercício 6.10 M

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 6.17 ilustra o pretendido.

```

1 >>> print(posicoes('agora e que vao ser elas, Ai, Ai!'))
2 {'a': [0, 4, 13, 22], 'A': [26, 30], 'e': [6, 10, 17, 20], 'i':
   : [27, 31], 'o': [2, 14], 'u': [9]}
3 >>>

```

Listagem 6.17: 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"}.
```

- a) Acrescente um elemento ao dicionário
- b) Altere o autor do python para "Guido van Rossum".
- c) Remova o elemento com chave "xpto".
- d) Quantos elementos tem o dicionário?
- e) 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?

Exercício 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: "4/5/6/2006"

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 uma árvore genealógica, organizada como um dicionário, escreva um programa que determine se duas pessoas são **irmãos/irmãs**.

Exercício 6.19 M

Dada uma árvore genealógica, organizada como um dicionário, escreva um programa que determine os **netos** de uma pessoa, caso existam.

Exercício 6.20 M

Dada uma 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 [informação¹](#). Existem dois grandes tipos de ficheiros: de texto e binários. Na ausência de informação em contrário o que vamos descrever aplica-se aos

¹Como 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

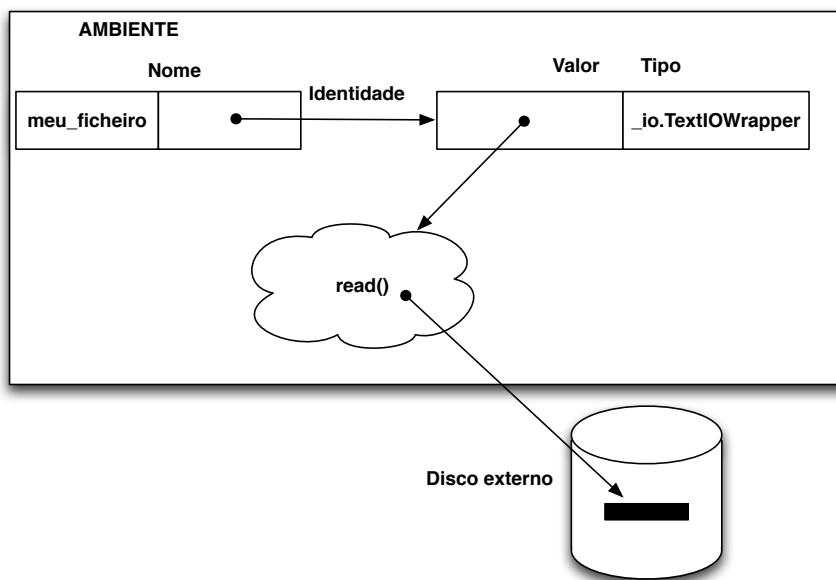


Figura 7.1: Representação em memória

dependendo da operação pretendida e da forma como deve ser interpretado o conteúdo. Para o caso de ficheiros de texto a tabela 7.1 mostra as alternativas..

Modo	Interpretação
r	Modo de leitura
w	Modo de escrita
a	Modo de acrescentar
r+	Modo de leitura e escrita sem truncamento
w+	Modo de leitura e escrita com truncamento

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:

¹ `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 7.2.

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 tudo 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 leitura².

```

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

```

²Mais à frente explicaremos porque fechamos e abrimos o ficheiro antes de cada operação e como podíamos ter evitado esse procedimento.

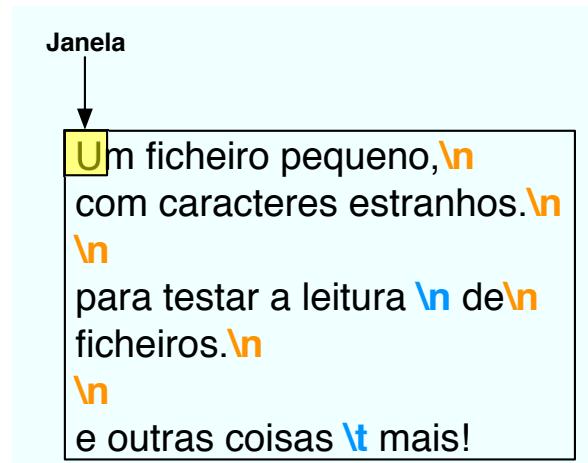


Figura 7.2: Um ficheiro depois de aberto

```
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(leitura)
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 def leitura():
2     nome_f= input("\n Nome absoluto do ficheiro:\t")
3     fich_ent=open(nome_f, 'r')
4     bytes=fich_ent.read(8)
5     print(Bytes: ", bytes")
6     linha= fich_ent.readline()
7     print("Linha: ", linha)
8     linhas=fich_ent.readlines()
9     print("Linhas: ", linhas)
10    fich_ent.close()
11    return "Fim"
12
13 if __name__ == '__main__':
14     leitura()

```

Listagem 7.4: Operações de entrada

A sessão no interpretador foi a seguinte:

```

1 >>> import leitura
2 >>> dir(leitura)
3 [ '__builtins__', '__doc__', '__file__', '__name__', 'leitura']
4 >>> leitura.leitura()

```

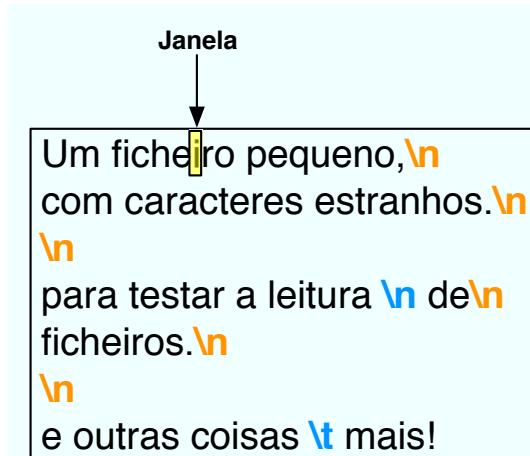


Figura 7.3: Situação depois de lidos os primeiros 8 bytes

```

5
6 Nome absoluto do ficheiro: /Users/ernestojfcosta/python/
    fichen1.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 bytes³. 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.

³Cada 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 7.3 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 abaixo 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

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 abeced\xelrio!\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 *ficheiro2013.txt*⁴ de conteúdo:

```

1 um ficheiro pequeno,

```

⁴Use 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.

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

```

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
23 'um ficheiro pequeno,\ncom caracteres estranhos.\n\npara
     testar a leitura \\n de\\nficheiros.\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).

Movimentos Relativos 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 movimentarmos 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()
```



Ficheiros de texto Unicode

Se os ficheiros de texto que queremos manipular apenas dispuserem de caracteres ASCII, o que foi dito atrás é o essencial de que precisa saber. No entanto, a probabilidade de ter que lidar com ficheiros com caracteres não ASCII é muito elevada. afinal somos portugueses e o que temos mais são caracteres acentuados, cês com cedilha, acentos circunflexos, e tiles. Felizmente que a versão 3 de **Python** resolveu o problema maior ao **automatizar** a codificação e descodificação dos caracteres não ASCII para nós, através da instrução `open`. Vejamos um exemplo simples.

```
1  >>> f_out = open('/data/unicode.txt', 'w')
2  >>> f_out.write('Luísa Conceição Hortência\n')
3  Traceback (most recent call last):
4      File "<string>", line 1, in <fragment>
5      builtins.UnicodeEncodeError: 'ascii' codec can't encode
6          character '\xed' in position 2: ordinal not in range
7          (128)
8  >>> f_out = open('/data/unicode.txt', 'w', encoding='utf-8'
9      )
10 >>> f_out.write('Luísa Conceição Hortência\n')
11 26
12 >>> f_out.close()
13 >>> f_in = open('/data/unicode.txt', 'r')
14 >>> data = f_in.read()
15 Traceback (most recent call last):
16     File "/Applications/WingIDE.app/Contents/MacOS/src/debug
17         /tserver/_sandbox.py", line 1, in <module>
18     # Used internally for debug sandbox under external
19         interpreter
20     File "/usr/local/pythonbrew/pythons/Python-3.2.3/lib/
21         python3.2/encodings/ascii.py", line 26, in decode
22         return codecs.ascii_decode(input, self.errors)[0]
23 builtins.UnicodeDecodeError: 'ascii' codec can't decode
24     byte 0xc3 in position 2: ordinal not in range(128)
25 >>> f_in = open('/data/unicode.txt', 'r', encoding='utf-8'
26     )
27 >>> data = f_in.read()
28 >>> data
29 'Luísa Conceição Hortência\n'
```

Como se vê basta explicitar a codificação na instrução `open`.

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 um contexto, tendo como sintaxe:

Gestores de contexto

```
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
7  com caracteres estranhos.

8
9
10
11 para testar a leitura \n de
12
13 ficheiros.

14 e outras coisas \t mais!
15 >>> f_ent.seek(0)
16 0
17
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
2 plt = matplotlib.pyplot
3
4 def ler(ficheiro):
5     with open(ficheiro,'r') as f_ent:
6         dados_car = f_ent.read().split()
7         dados = []
8         for elem in dados_car:
9             dados.append(float(elem))
10    return dados
11
12 def mostra(xetiq, yetiq,tit,x,y):
13     plt.xlabel(xetiq)
14     plt.ylabel(yetiq)
15     plt.plot(x,y)
16
17
18 def main(ficheiro):
19     dados = ler(ficheiro)
20     mostra('Meses','Temperatura','','',range(1,13),dados)
21     plt.show()
22
23
24 if __name__ == '__main__':
25     main('/data/dados_simples.txt')
```

Listagem 7.7: Ler e mostrar pemeraturas

O programa principal executa em sequênci 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
2 plt = matplotlib.pyplot
3
4 def le_todas_temperaturas(fich):
5     """
6     Extrai os dados de temperaturas relativos a Portugal.
7     """
8     with open(fich, 'r') as f_ent:
9         portugal = list()
10        dados = le_uma_temperatura(f_ent)
11        while dados != -1:
12            portugal.append(dados)
13            dados = le_uma_temperatura(f_ent)
14        f_ent.close()
15        return portugal
16
17 def le_uma_temperatura(f_ent):
18     """
19     Ler dados da temperatura de uma cidade.
20     Devolve -1 se fim de ficheiro
21     """
22     linha = f_ent.readline()

```

```

23     while (linha != '') and (linha == '\n'):
24         linha = f_ent.readline()
25     if linha == '':
26         return -1
27     else:
28         linha = linha[:-1].split()
29         return [float(dado) for dado in linha]
30
31 def mostra_todas(xetiq,yetiq,tit,dados):
32     plt.xlabel(xetiq)
33     plt.ylabel(yetiq)
34     plt.title(tit)
35     for cidade in dados:
36         plt.plot(cidade)
37
38
39 def main(ficheiro):
40     dados = le_todas_temperaturas(ficheiro)
41     mostra_todas('Meses','Temperatura','Temperaturas Médias
42                 das Cidades',dados)
43     plt.show()
44
45 if __name__ == '__main__':
46     main('/data/temperaturas.txt')

```

Listagem 7.10: Temperaturas

Ao correr o programa obtemos o gráfico da figura 7.4.

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 == '':

```

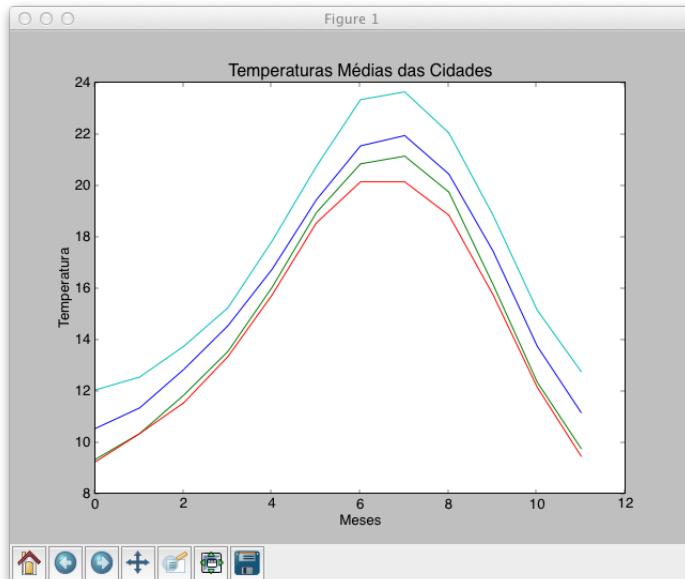


Figura 7.4: Temperaturas das cidades de Portugal

```

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.

Para concluir este exemplo, vamos considerar a situação em que o ficheiro é mais complexo pois contém informação sobre a temperatura e a pluviosi-

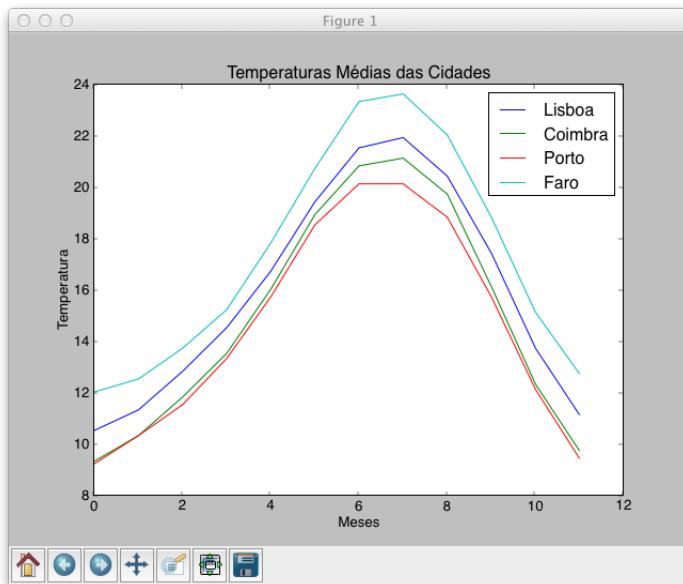


Figura 7.5: Gráfico com legenda

dade. 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 doze 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     # procura primeira linha significativa
7     linha = f_ent.readline()
8
9     while (linha != '') and (linha == '\n'):
10         linha = f_ent.readline()
11
12     if linha == '':
13         return -1
14     else:
15         # extrai dados
16         cidade = linha[:-1]
17         pluviosidade = [float(dado) for dado in f_ent.readline()
18                         ()[:-1].split('\t')[1:]]
19         temperatura = [float(dado) for dado in f_ent.readline()
20                         ()[:-1].split('\t')[1:]]
21
22     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 dicionáro. 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'])
19         cidades.append(c)
20
21
22     figura = plt.figure()
23     fig_1 = figura.add_subplot(211)
24     plt.title('Cidades de Portugal')
25     fig_2 = figura.add_subplot(212)
26
27
28     for indice in range(len(cidades)):
29         fig_1.plot(chuva[indice])
30         fig_2.plot(temp[indice])
31
32
33     fig_1.set_ylabel('Pluviosidade (mm)')
34     fig_2.set_ylabel('Temperatura (C)')
35     plt.xticks(range(0,12),meses, rotation=17)
36     plt.legend(cidades, loc=0)
37
38
39     plt.show()

```

Quando executamos o programa obtemos o resultado da figura 7.6.

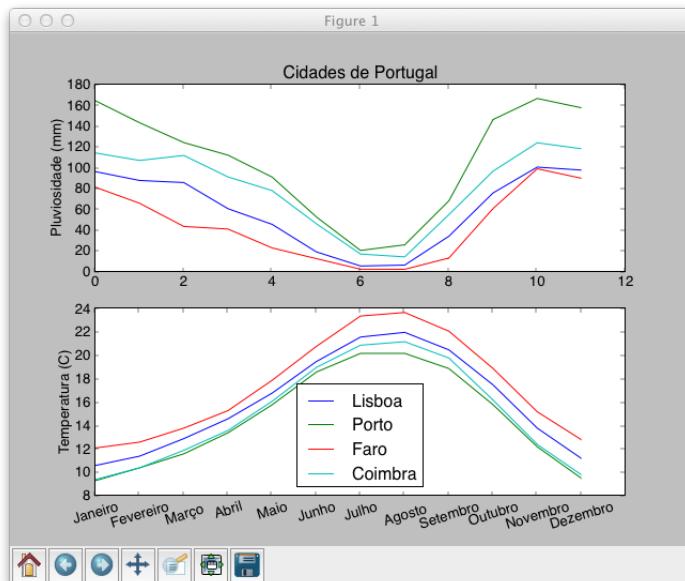


Figura 7.6: Temperatura e pluviosidade

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', '?'], ['Zeus
3 Euclides', '12', '34', '45', '30', '?'], ['Anaximandro
4 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',
2                                '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)
7     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', ,
2   'Custo'], [ 'Manel dos Ossos', 'Coimbra', 'Portuguesa', '$' ], [
3   'Chez Lui', 'Paris', 'Francesa', '$$$$$' ], [ 'McMe': 'Burgos', '
4   Fast Food', '$$' ]], ':')

```

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:] # <-- 0 cabeçalho não interessa
6     !
7     # Manipula
8     for i in range(len(notas_parciais)):
9         nome, teste,projecto,normal,recurso,nota =
10            notas_parciais[i]
11            notas_parciais[i][5] = str(0.2 * int(teste) + 0.2 *
12              int(projecto) + 0.6 * max(int(normal),int(recurso)))
13
14    dados = [dados[0]] + notas_parciais
15
16    # Escreve
17    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()

```

bytes

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 de-

terminado 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).

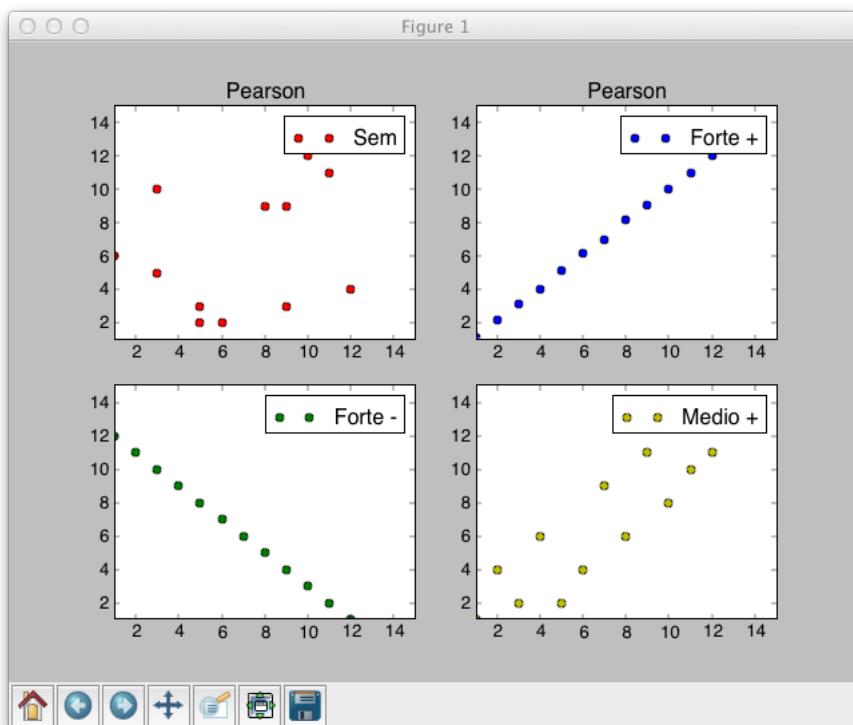


Figura 7.7: Coeficiente de Correlação de Pearson: exemplos

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):
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     for indice in range(n):
27         soma = soma + (lista_a[indice] - media_a) * (lista_b[
28             indice] - media_b)
correlacao = soma / ((n - 1) * desvio_a * desvio_b)
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]
23        nome_2 = url_2.split('=')[-1]
24
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 if __name__ == '__main__':
```

```

37     url_apple = 'http://ichart.finance.yahoo.com/table.csv?s=
38         AAPL'
39     url_coke = 'http://ichart.finance.yahoo.com/table.csv?s=
40         Coke'
41
42     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.

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?

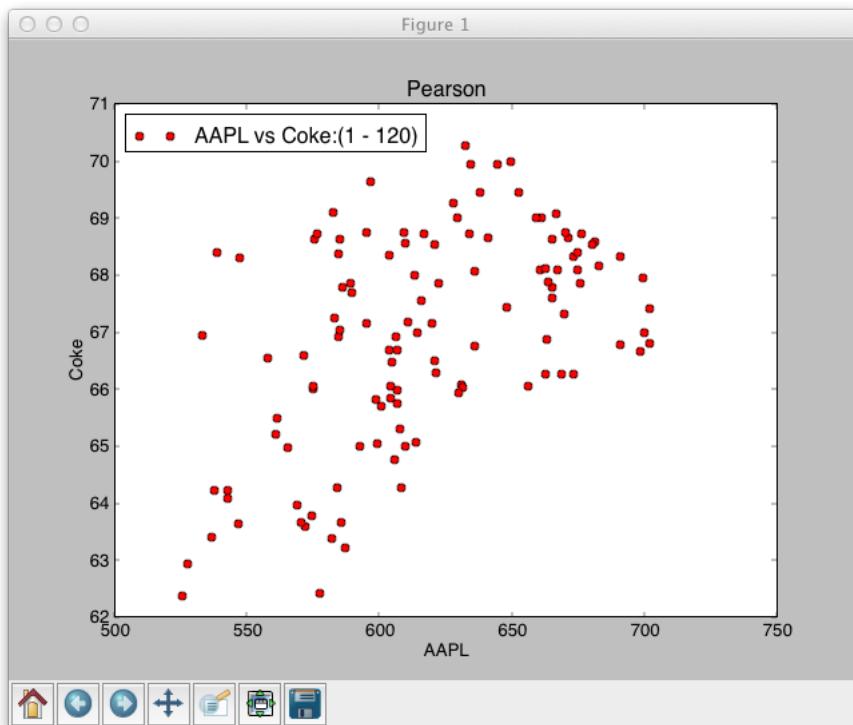


Figura 7.8: A Apple e a Coca-cola

- 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 [.](http://iprp.net)

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**⁵, 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

⁵Disponível em <http://iprp.net>.

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 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 transação, na forma: número da transação, 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

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 tabela 7.5 ilustra uma situação possível.

Tabela 7.5: Ficheiro de Dados: entrada

Ernesto Costa	60	102	1
Ana Paz	44	411	2
Carlos Ferreira	20	203	2

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 tabela 7.6 mostra a saída resultante de aplicar o programa ao ficheiro de entrada da figura ???. 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.

Tabela 7.6: Ficheiro transformado

EC	60	Professor	Casado
AP	44	Advogado	Solteiro
CF	20	Estudante	Solteiro

Exercício 7.13 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 7.14 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 7.15 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 7.16 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 7.17 Módulo csv D O ficheiro *zoo.csv*⁶ 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 7.18 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>.

⁶Pode 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.

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

8.1 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**¹. Uma representação comum das imagens em computador consiste em representar cada pixel separadamente, falando-se

¹Acrónimo derivado do inglês *picture element*.

então em imagens *bitmap*². 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**³, 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 diferentes⁴.

A figura 8.1 mostra algumas cores e respectivas codificações RGB.

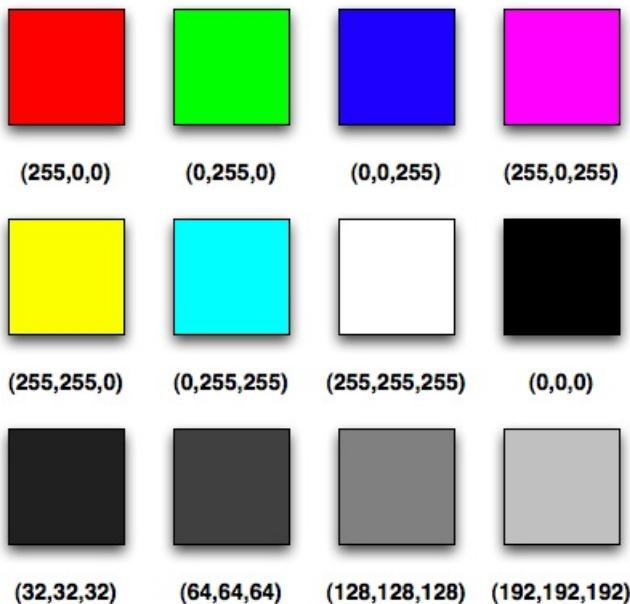


Figura 8.1: Mapeamento RGB cores

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).

²Uma 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.

³É possível a existência de mais canais, como seja o canal **alfa** para a transparência.

⁴Outros modelos de cor são o HSB e o CMYK, este último usado pelas impressoras.

As imagens ocupam muito espaço. A tabela 8.1 mostra alguns valores, isto é, o espaço ocupado para diferentes **resoluções** das imagens.

Tabela 8.1: Imagens e tamanhos

	320×240	640×480	1024×768
24-bits cor	230400 bytes	921600 bytes	2359296 bytes

É pois com naturalidade que se procuraram métodos de compressão de imagem. Existem vários métodos de compressão, como é o caso da norma JPEG⁵, 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 aceitável⁶. Existem algoritmos que tornam possível comprimir a imagem sem perda de informação, como acontece com a técnica conhecida por RLE⁷. Existem vários outros formatos de imagem, nomeadamente os que não provocam perdas como sejam o TIFF, GIF ou PNG.

8.2 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:

⁵Acrónimo derivado do inglês *Joint Photographic Experts Group*

⁶Em certas aplicações, e.g., na área médica, essa perda de qualidade não pode ser tolerada.

⁷Do 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 manipulação⁸. 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='')
```

⁸Os 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.

8.3 O módulo cImage

Regressemos às imagens e ao problema da sua construção, consulta e manipulação. Vamos utilizar o módulo **cImage**⁹ 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**¹⁰. Enquanto o **Tkinter** nos permite definir uma Interface de Utilizador Gráfica¹¹, 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**¹² o que permite ser compatível com a versão 3 da linguagem. Estes módulos dependem ainda de um conjunto importante

⁹TBD:...colocar informação sobre o modo de obter...

¹⁰Acrônimo de *Python Imaging Library*. Pode ser obtido em <http://www.pythonware.com>.

¹¹Em inglês *Grapgical User Interface* (GUI).

¹²Mais 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 8.2). 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.

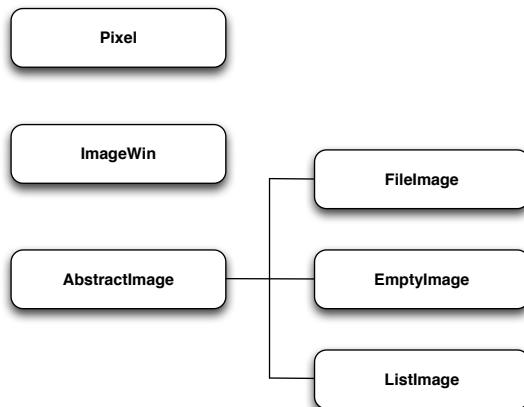


Figura 8.2: Os tipos de cImage

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()`).

8.4 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 8.3.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 vermelha¹³.

```

1 def cria_janela_cor(nome, largura, altura, cor):
2     janela= cImage.ImageWin(nome, largura, altura)
```

¹³A 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 8.3: 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 8.4: 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 8.5: 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 8.6.

```
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)
```

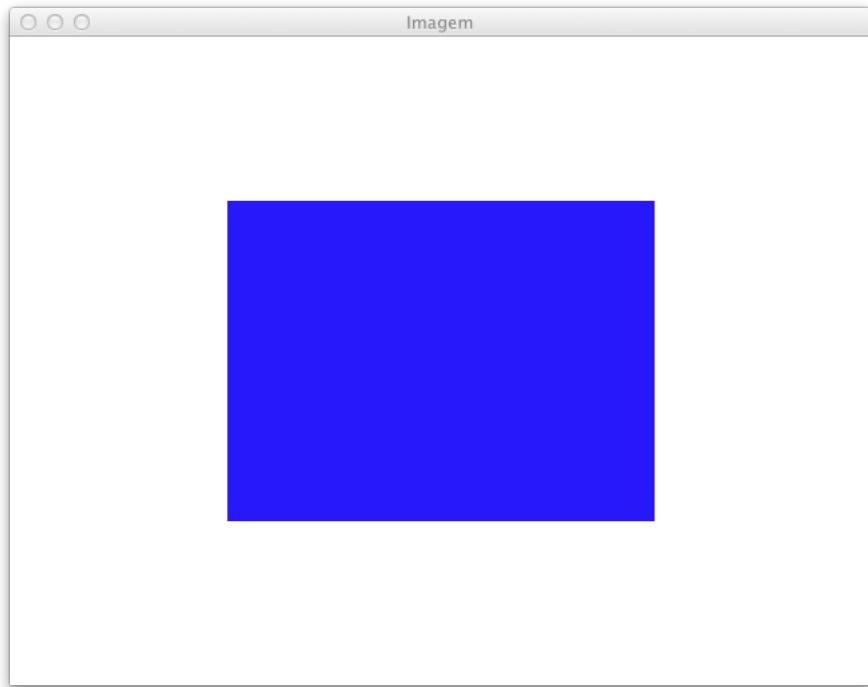
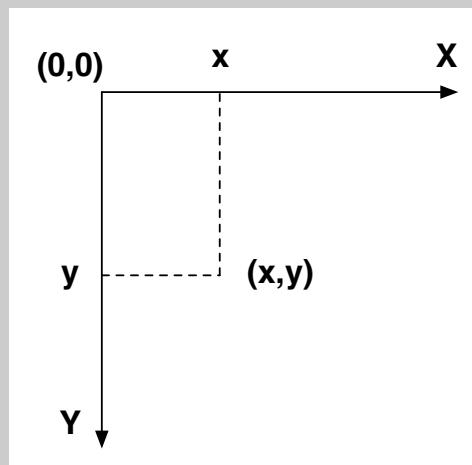


Figura 8.6: Lidar com a cor e a posição



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 8.7.

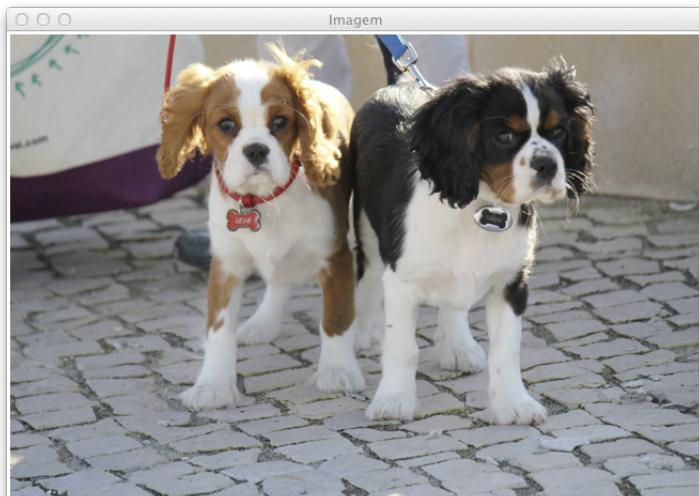


Figura 8.7: Reproduzir uma imagem

8.5 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 matrizes¹⁴. 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 8.8). 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.

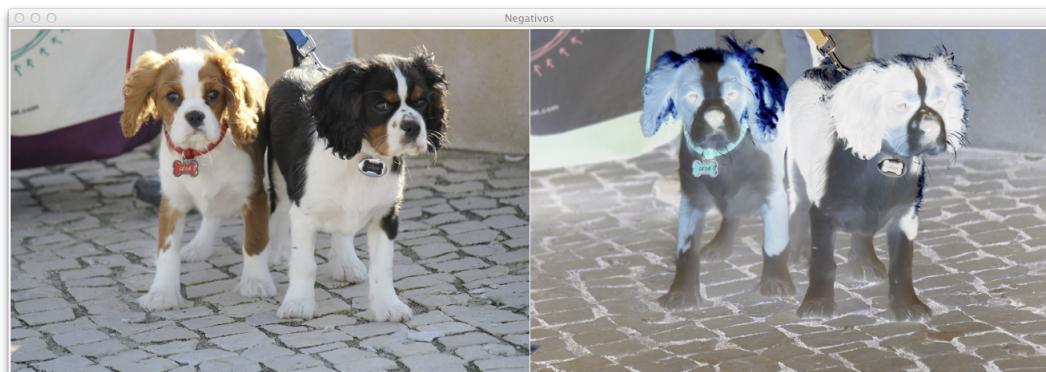


Figura 8.8: Negativo de uma imagem

¹⁴Devemos 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 8.7 obtemos o resultado da figura 8.9.



Figura 8.9: 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 8.10.



Figura 8.10: 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 8.11.



Figura 8.11: Sepia

8.6 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 8.12) 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).

8.7 Exemplos complementares

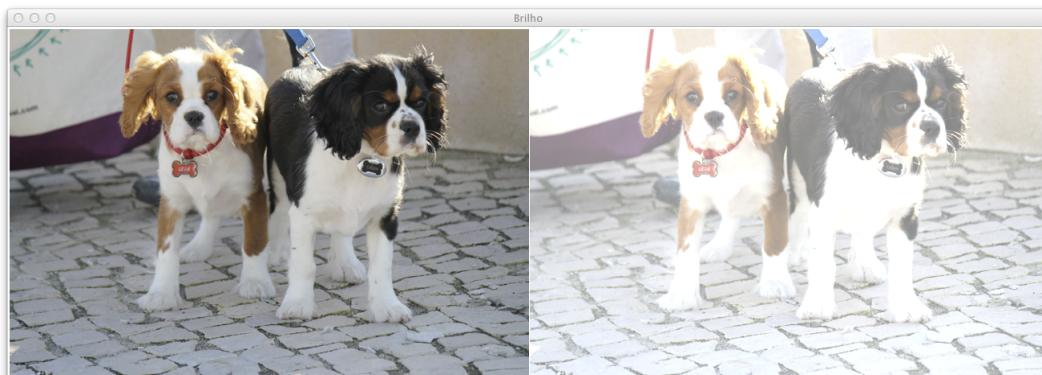
Apresentados os princípios básicos e alguns exemplos de programas que modificam de modo uniforme uma imagem, dando origem a um padrão de programa, está na altura de passar a exemplos diferentes.

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 8.13 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 4×3 , 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;



(a) Mais Claro



(b) Mais Escuro

Figura 8.12: 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

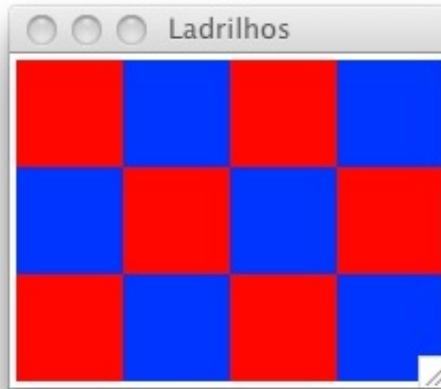


Figura 8.13: O chão da cozinha

programa que gera um quadrado de uma cor específica¹⁵. 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 8.14, 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

¹⁵ Aplicamos aqui de novo o princípio da abstracção procedural.

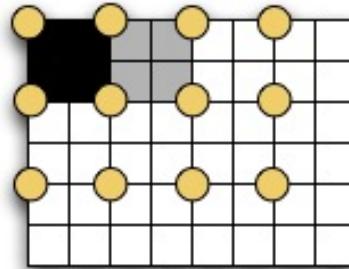


Figura 8.14: O problema do posicionamento

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 8.15 ilustra o que se pretende. No exemplo apresentado a imagem foi ampliada duas vezes em largura e três vezes em altura.

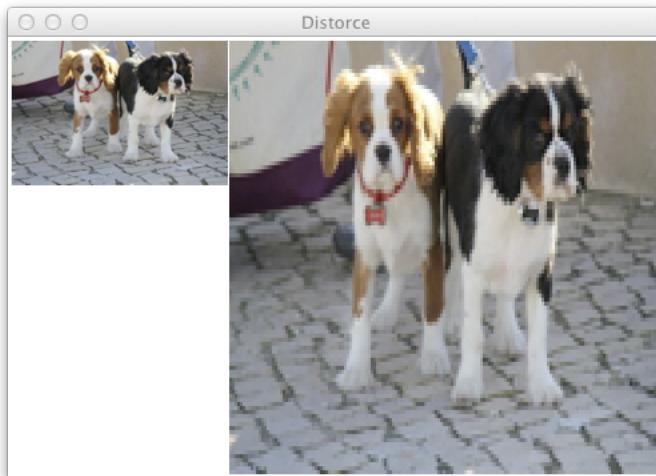


Figura 8.15: Distorcer uma imagem

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 8.16 exemplifica a ideia para o caso de 3×2 .

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 8.16):

```

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

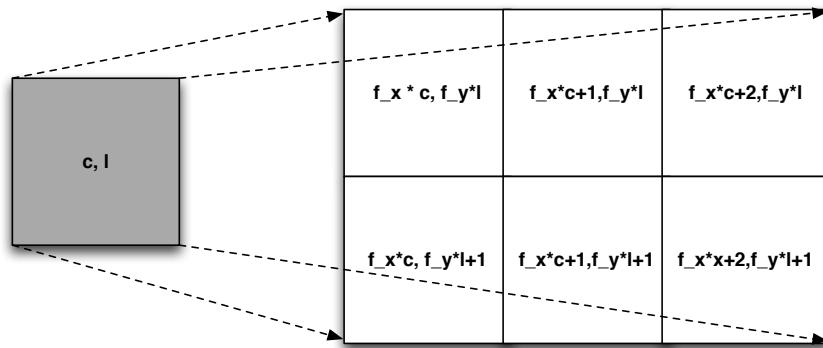


Figura 8.16: Clonar um pixel

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 8.17 mostra o efeito pretendido.



Figura 8.17: 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 8.18 tenta mostrar a situação.

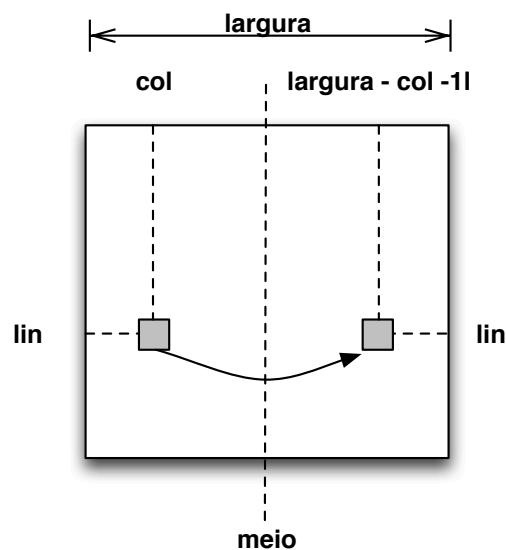


Figura 8.18: Onde colocar os pixeis?

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.

8.8 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 8.19 mostra uma imagem pixelizada e o resultado da suavização das transições.

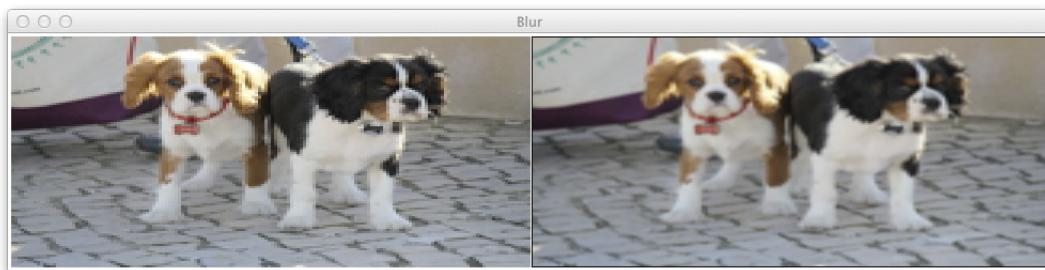


Figura 8.19: Tratamento da pixelização

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 8.20).

$(x-1,y-1)$	$(x,y-1)$	$(x+1,y-1)$
$(x-1,y)$	(x,y)	$(x+1,y)$
$(x+1,y+1)$	$(x,y+1)$	$(x+1,y+1)$

Figura 8.20: 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 8.9).

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.

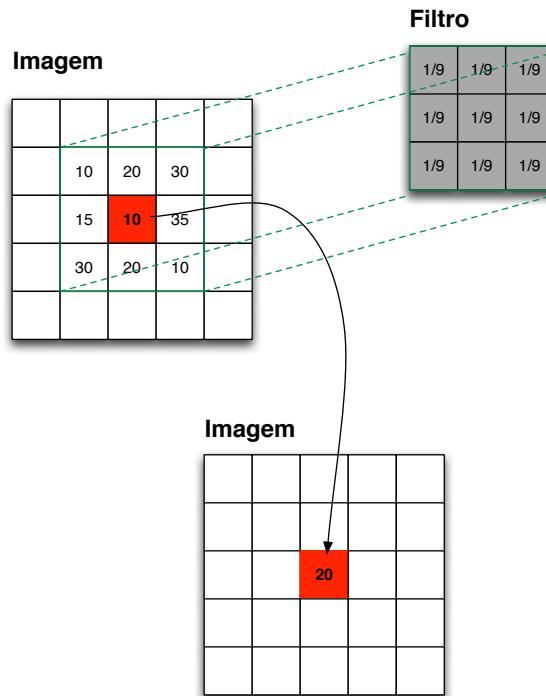
$$pixel(x, y) = \left(\sum_{c=-1}^1 \sum_{l=-1}^1 \frac{1}{9} \times pixel(x + c, y + l) \right)$$

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 8.21. Na figura, para simplificar, apresentamos apenas os valores hipotéticos para um dos canais.

Filtro

Dizemos que aplicamos um **filtro** à imagem¹⁶. 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 8.22 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.

¹⁶Também designado por máscara ou núcleo.



$$pixel(x, y) = \left(\sum_{c=-1}^1 \sum_{l=-1}^1 \frac{1}{9} \times pixel(x + c, y + l) \right)$$

Figura 8.21: Aplicar um filtro a uma imagem

Podemos adaptar a fórmula anterior de modo trivial:

$$pixel(x, y) = \left(\sum_{c=-1}^1 \sum_{l=-1}^1 filtro(c, l) \times pixel(x + c, y + l) \right)$$

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 8.22: 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 8.23 este filtro melhora a qualidade da imagem contrariando o efeito da pixelização.



Figura 8.23: 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 finalidade¹⁷. 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 8.24 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 8.24: 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 $\sqrt{val_x^2 + val_y^2}$ ¹⁸. 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()
```

¹⁷ 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.

¹⁸ 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
21 def transforma_convol(imagem,mascara_1, mascara_2,limiar):
22     """Imagem em escala de cinzentos."""
23     largura = imagem.getWidth()
24     altura = imagem.getHeight()
25     nova_imagem = cImage.EmptyImage(largura, altura)
26     index = len(mascara_1)//2
27     for coluna in range(index,largura-index):
28         for linha in range(index,altura-index):
29             val_x = convolve(coluna,linha,imagem, mascara_1)
30             val_y = convolve(coluna,linha,imagem, mascara_2)
31             val = restringe(math.sqrt(val_x**2 + val_y**2), 0,
32                             255)
32             novo_pixel = preto_branco_pixel(cImage.Pixel(val,
33                                         val, val), limiar)
34             nova_imagem.setPixel(coluna,linha,novo_pixel)
35     return nova_imagem
36
36 def convolve(x,y, imagem, filtro):
37     """Imagem em escala de cinzentos."""
38     index = len(filtro) // 2
39     r = 0
40     for i in range(-index, index+1):
41         for j in range(-index, index+1):
42             pixel = imagem.getPixel(x+j,y+i)
43             r += pixel.getRed() * filtro[j+index][i+index]
44     r = restringe(int(r),0,255)
45     return r

```

Executando o programa para diferentes valores do limiar obtemos resultados diversos (ver figura 8.25). A qualidade das resultado também depende das características das imagens.

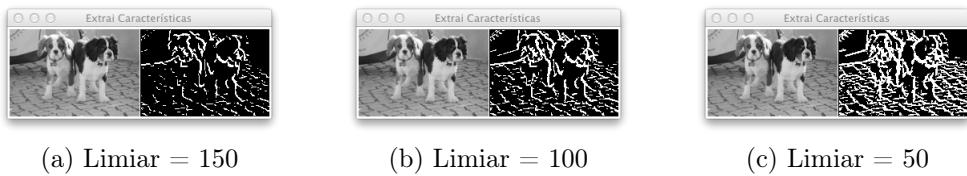


Figura 8.25: Operadores de Sobel

8.9 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 GIMP¹⁹. 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

```

¹⁹Pode 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 8.1 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

²⁰Para 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 8.2 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:

```
print(sub_matriz([[1,2,3,4],[5,6,7,8],[9,10,11,12]],1,1,2,2))
```

o resultado obtido deverá ser:

```
[[6, 7], [10, 11]]
```

Exercício 8.3 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 8.4 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 8.5 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 8.6 F

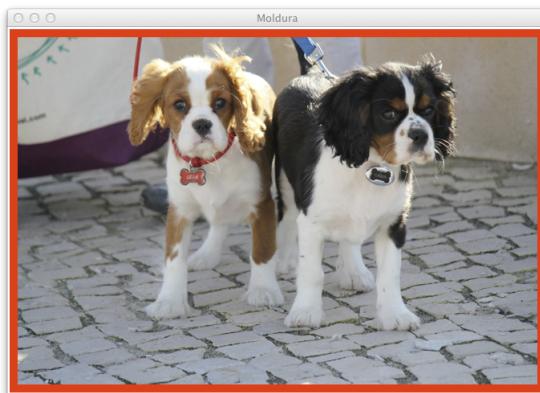


Figura 8.26: Acrescentar uma moldura

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 8.26 ilustra o pretendido.

Exercício 8.7 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 8.27 ilustra a ideia pretendida.



Figura 8.27: Corte de imagem

Exercício 8.8 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 8.28 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 8.28: A preto e branco

Exercício 8.9 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 8.10 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 8.29 ilustra o que se pretende.

Exercício 8.11 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 8.30 mostra o efeito pretendido.

Exercício 8.12 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

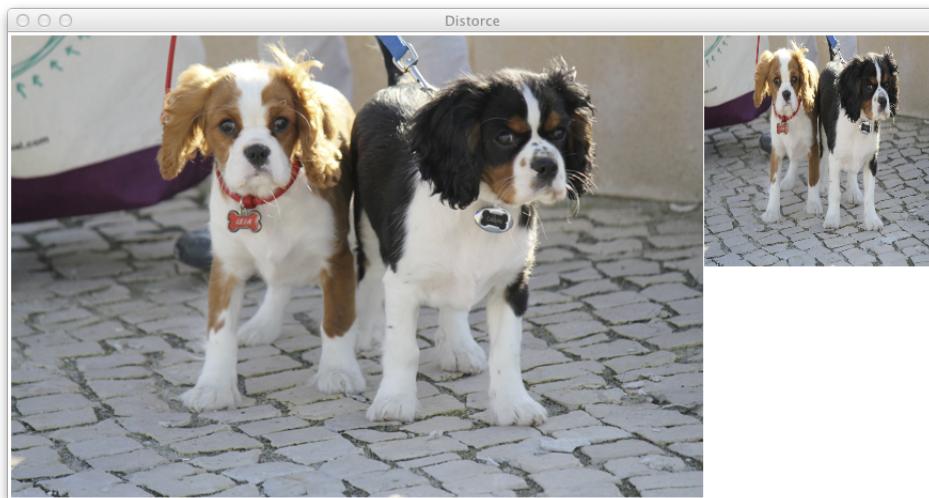


Figura 8.29: Redução de uma imagem

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 8.31 ilustra.

Exercício 8.13 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 8.14 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 8.32 mostra o resultado pretendido. **Sugestão:** no bordo calcule a média dos vizinhos existentes.

Exercício 8.15 M

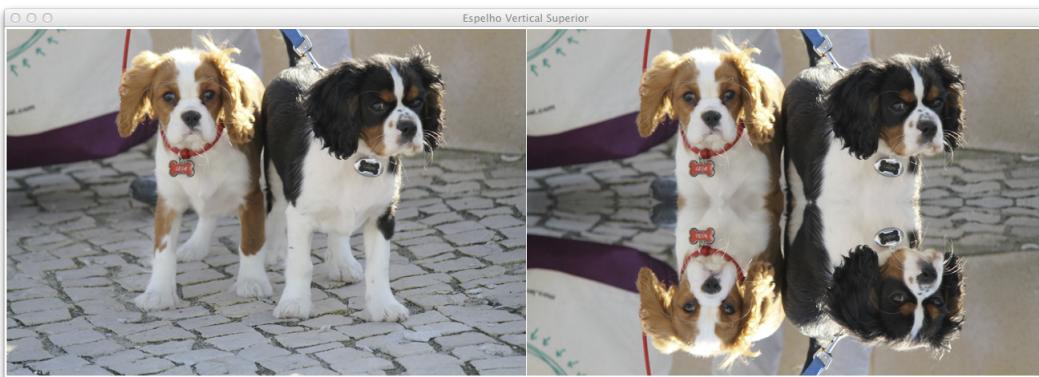


Figura 8.30: Espelho horizontal

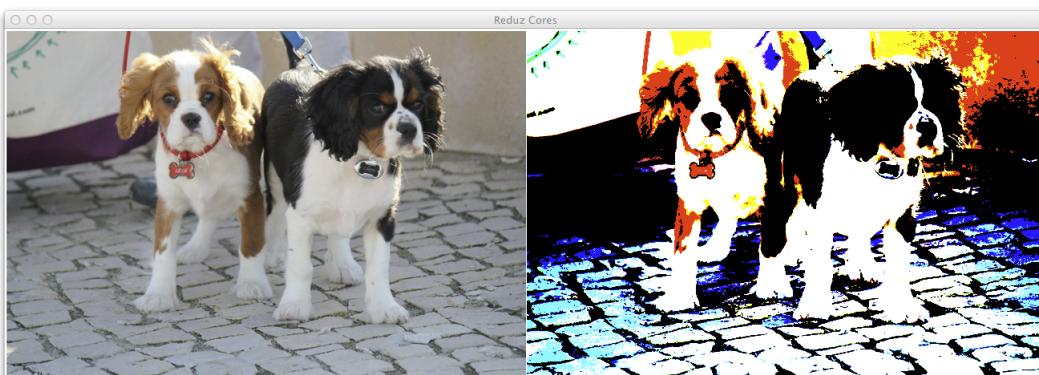


Figura 8.31: Redução de cores

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 8.16 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

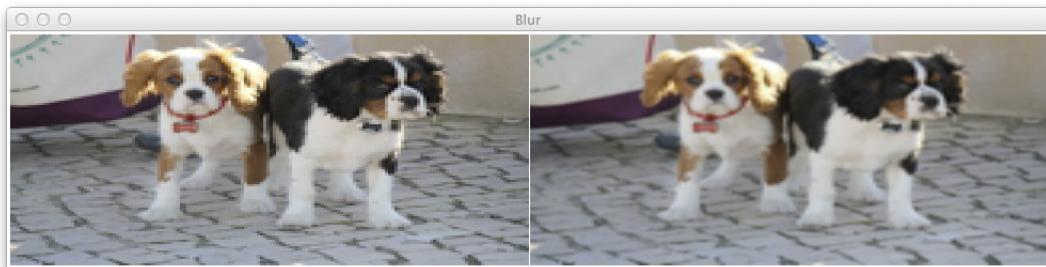


Figura 8.32: Suavizar sem moldura

que está na posição do meio. Implemente o respectivo programa.

Exercício 8.17 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 8.33 apresentamos uma combinação entre Einstein e Gandhi.

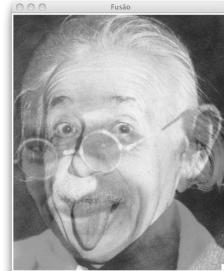


Figura 8.33: Gandstein

Exercício 8.18 D

Desenvolva um programa que dada uma imagem a roda de 90 graus. O sentido é à sua escolha.

Exercício 8.19 MD

Neste exemplo pretendemos resolver uma questão semelhante à do exemplo 8.9. 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 8.20 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 8.34 ilustra o pretendido.

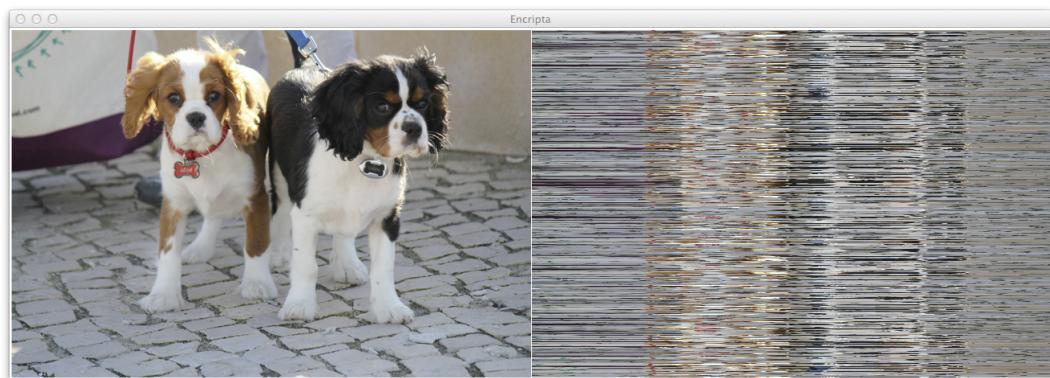


Figura 8.34: Encriptar uma imagem

Capítulo 9

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

9.1 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 9.1 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 9.2 mostra o processo usado nas três primeiras etapas¹.

Estamos perante exemplos do conceito de recursividade: quando um ob-

[Recursividade](#)

¹As escalas das duas figuras são diferentes.

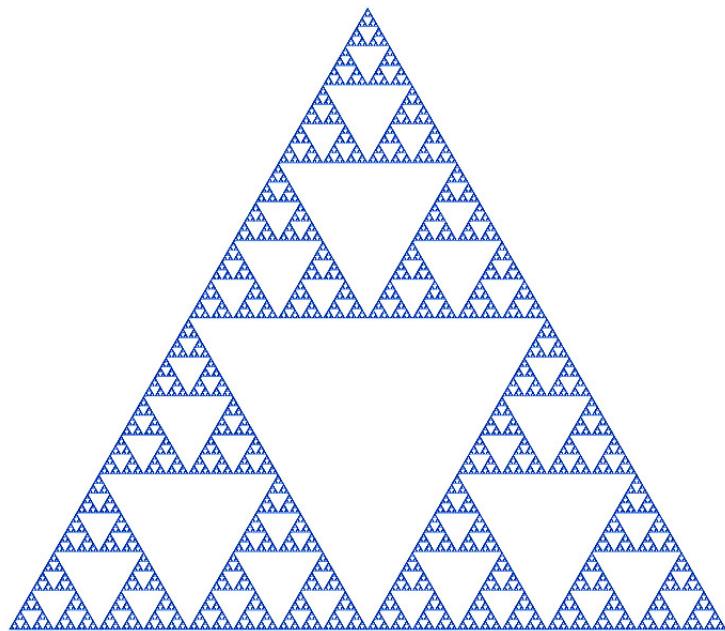


Figura 9.1: *Sierpinski Gasket*

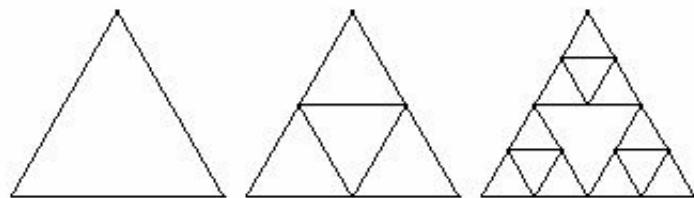


Figura 9.2: Sierpinski: construção

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*².

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 9.3 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.

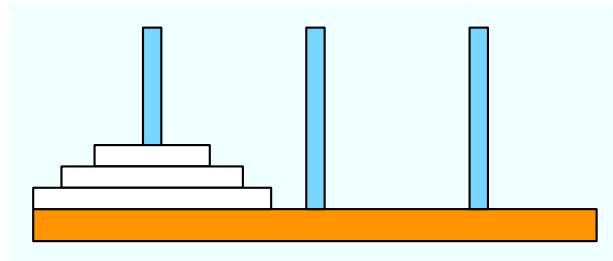


Figura 9.3: Torres de Hanói

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

Semelhança

²Como discutiremos mais adiante estas últimas manifestações de recursividade têm um problema importante: nunca nos permitem parar.

cumprem papéis diferentes. A figura 9.4 mostra a situação em que conseguimos deslocar dois discos da vara da esquerda para a do meio respeitando as regras do jogo.

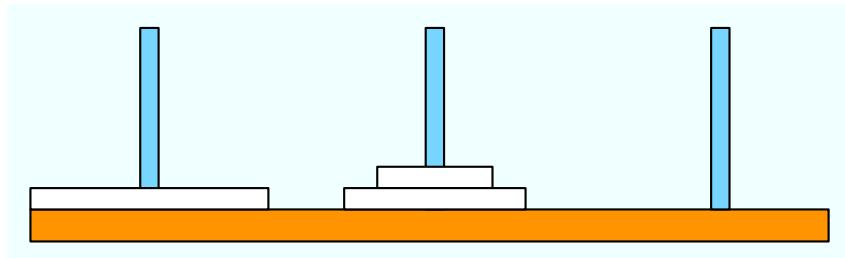


Figura 9.4: Resolução de um sub problema semelhante

Se tivermos conseguido resolver o sub problema do modo indicado,³ 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 9.5.

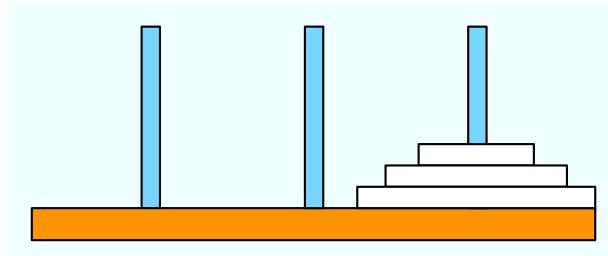


Figura 9.5: Torres de Hanói: solução final

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

³Por agora vamos admitir que sim, embora ainda não saibamos bem como foi isso possível!

de novo semelhantes ao inicial. Claro que não podemos fazer isto infinitamente, pelo que terá que haver uma situação em que não necessitamos de 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 9.1 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 9.1: '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 9.2. 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 9.2: '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.

[Caso de Base](#)

[Caso Recursivo](#)

Recursividade

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 problema⁴.

- Quando um objecto de define em função dele próprio⁵
- 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.

9.2 Exemplos

9.2.1 Números

Um exemplo clássico de recursividade é a função **factorial**. Por definição temos:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases} \quad (9.1)$$

A partir dela construímos o programa 9.3.

```

1 # factorial
2 def fact(n):
3     if n == 0:
4         return 1
5     else:
6         return n * fact(n - 1)

```

Listagem 9.3: 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

⁴Estas duas visões estão inter-relacionados como adiante se verá.

⁵Estamos no domínio das chamadas definições indutivas.

mais simples para o argumento $(n - 1)$.

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**⁶. A **Desenrolar**, **Enrolar**

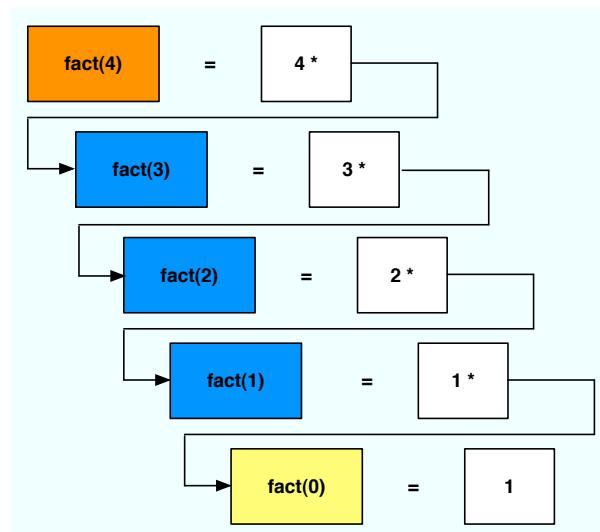


Figura 9.6: Factorial: fase de desenrolar

Como já foi referido, e a figura 9.6 ilustra, há um conjunto de cálculos que ficaram em suspenso e agora precisam ser concluídos. Entramos na segunda fase do processo, denominada **enrolar**⁷. 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 9.7, tenta ilustrar o processo.

⁶Do inglês *unfold*.

⁷Do inglês *fold*.

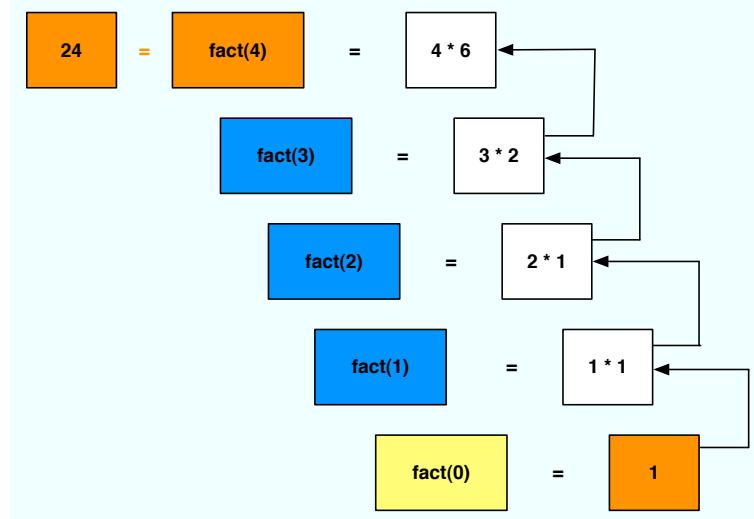


Figura 9.7: Factorial: fase de enrolar

Síntese Antes de prosseguir podemos já fazer uma pequena síntese. A **recursividade** caracteriza-se por:

- 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.

Soma

```

1 def soma(m,n):
2     if n == 0:
3         return m
4     else:
5         return 1 + soma(m,n-1)

```

Listagem 9.4: 'Soma'

Produto

```

1 def prod(m,n):
2     if n == 0:
3         return 0
4     else:
5         return soma(m, prod(m,n-1))

```

Listagem 9.5: 'Produto'

Exponencial

```

1 def exp(m,n):
2     if n == 0:
3         return 1
4     else:
5         return prod(m, exp(m,n-1))

```

Listagem 9.6: '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 abstrato que apresentamos na listagem 9.7⁸.

Recursividade
Linear

```

1 def f(n):
2     if n == 0:
3         return <constante>
4     else:

```

⁸O 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.

```
5   return h(g(n),f(n-1))
```

Listagem 9.7: 'Modelo de recursividade linear'

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:

$$\sum_{i=0}^n i = \begin{cases} 0 & n = 0 \\ n + \sum_{i=0}^{n-1} i & n > 0 \end{cases} \quad (9.2)$$

Daqui retiramos imediatamente o programa ilustrado na listagem 9.8.

```
1 def somat(n):
2     if n == 0:
3         return 0
4     else:
5         return n + somat(n-1)
```

Listagem 9.8: '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 9.9.

```
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 9.9: '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 9.10.

```

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 9.10: '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 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 termina⁹. 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**¹⁰. Actualmente os compiladores modernos são capazes de detectar uma definição recursiva terminal e gerar automaticamente o código iterativo correspondente.

Recursividade
Terminal

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 Fibonacci¹¹. Na sua origem Fibonacci

⁹Na realidade termina por exaustação dos recursos da máquina!

¹⁰Saliente-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.

está um problema que pode ser enunciado do seguinte modo. Admitamos 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 9.8 mostra a evolução da população de casais ao longo dos meses. A amarelo a indicação dos coelhos em condição de se reproduzirem.

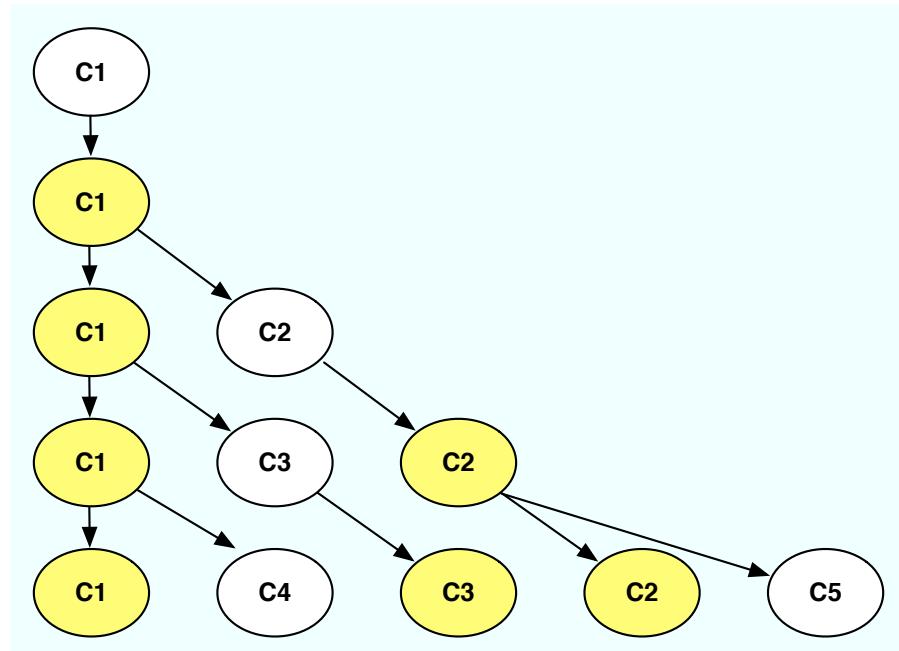


Figura 9.8: Fibonacci: coelhos e reprodução

O número de casais é dado pela sequência $1, 1, 2, 3, 5, \dots$, 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:

$$Fib(n) = \begin{cases} 1 & n = 1 \text{ ou } n = 2 \\ Fib(n - 1) + Fib(n - 2) & n > 2 \end{cases} \quad (9.3)$$

Esta definição tem uma tradução directa no programa da listagem 9.11.

```
1 def fib(n):
```

¹¹O nome da sequência deve-se ao matemático italiano Leonardo de Pisa, também conhecido por filho de Bonacci, que viveu na Idade Média

```

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 9.11: '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](#)

$$\binom{n}{k} = \begin{cases} 1 & k = 0, k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{caso contrário} \end{cases} \quad (9.4)$$

Daqui decorre trivialmente o programa da listagem 9.12.

```

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 9.12: 'Binómio de Newton'

Curiosamente existe uma relação entre os números de Fibonacci e os coeficientes de binómio. A figura 9.9 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 de que uma chamada recursiva num dos ramos da condicional.

[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

[Par e Ímpar](#)

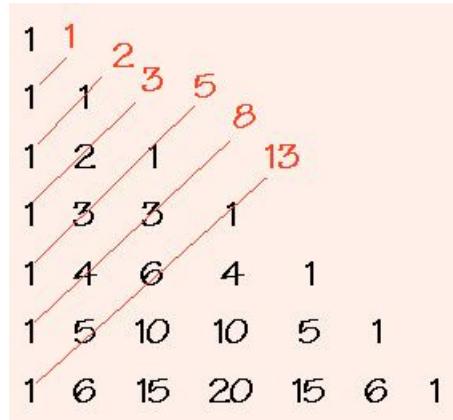


Figura 9.9: Números de Fibonacci e Binómio de Newton

Recursividade Cruzada

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 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 9.13: '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 [Sequência Alternada](#) 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 9.14.

```

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 9.14: '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.

9.2.2 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 9.15

```

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 9.15: '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 9.16 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 9.16: '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 *algum* 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 9.10 mostra a ideia.

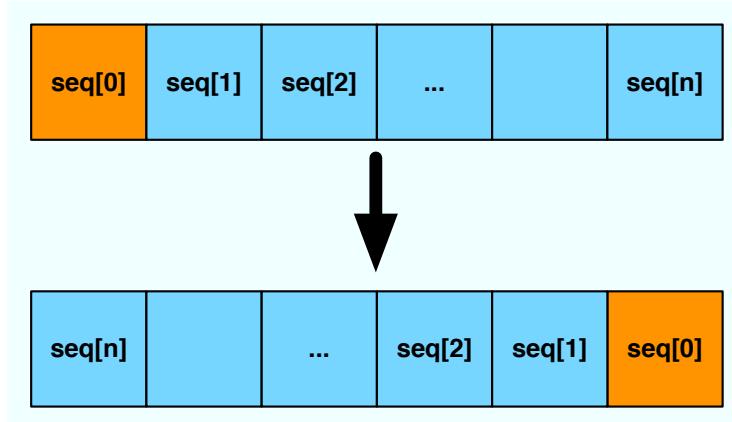


Figura 9.10: Inverter uma sequência

O programa correspondente é dado na listagem 9.17.

```

1 def inverte(seq):

```

```

2   if len(seq) == 0:
3       return seq
4   else:
5       return inverte(seq[1:]) + seq[0]

```

Listagem 9.17: '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 9.11 ilustra o processo e a listagem 9.18 apresenta o programa.

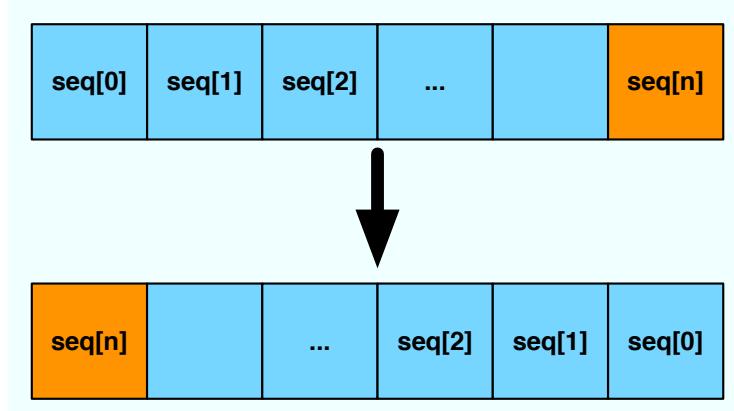


Figura 9.11: Inversão: alternativa

```

1 def inverte2(seq):
2     if len(seq) == 0:
3         return seq
4     else:
5         return seq[-1] + inverte2(seq[:-1])

```

Listagem 9.18: '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 9.19. Notar 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 9.19: '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 9.12 ilustra a ideia e o programa 9.20 mostra a solução.

```

1 def sobe_e_desce(n):
2     if n == 1:
3         return [1,1]

```

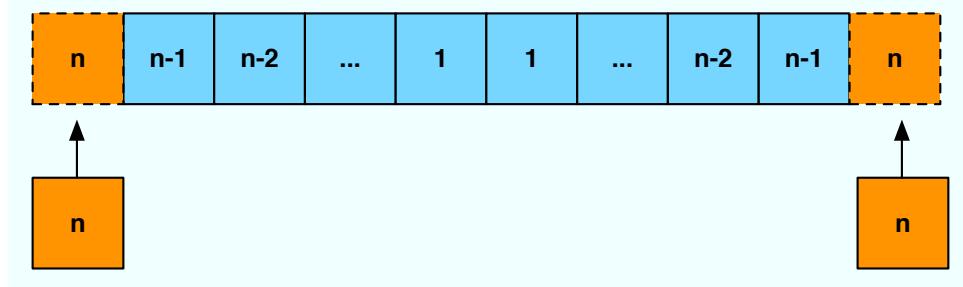


Figura 9.12: Sobe e Desce

```

4   else:
5       return [n] + sobe_e_desce(n-1) + [n]

```

Listagem 9.20: 'Sobe e Desce'

Intercalar

Suponhamos que temos duas sequências de comprimento eventualmente 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 9.21 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 9.21: 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 9.22 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 enumeração, 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 9.22: Intercalar

Se não *gosta* de listas por compreensão¹², a listagem 9.23 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 9.23: Intercalar: variante

Quando uma sequência está ordenada (por exemplo de modo ascendente) [Procura Binária](#) 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

¹²Se não gosta faz mal ...

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. 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 9.24.

```

1 def procura_bin_rec(x,seq, inicio,fim):
2     """ Procura com base no facto dos elementos estarem
3         ordenados
4 """
5
6     if inicio == fim:
7         if seq[inicio] == x:
8             return inicio
9         else:
10            return -1
11
12     else:
13         meio =(inicio + fim)/2
14         if x == seq[meio]:
15             return meio
16         elif x < seq[meio]:
17             return procura_bin_rec(x,seq,inicio,meio-1)
18         else:
19             return procura_bin_rec(x,seq,meio+1,fim)

```

Listagem 9.24: 'Procura Binária'

9.2.3 Desenhos

Figuras Recursivas

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 9.25 mostra o código e a figura 9.13 mostra o desenho que se obtém com a chamada **figura(30,45)**.

```

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.

```

```

4      ....
5      pd()
6      if lado:
7          forward(lado)
8          right(angulo)
9          figura(lado-1,angulo)
10     ht()
11     return 0

```

Listagem 9.25: 'Uma figura simples'



Figura 9.13: 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 9.14 mostramos o resultado da chamada **figura(30,75)**.



Figura 9.14: Mudando o ângulo

Vamos alterar o programa por forma a controlar o valor do decremento do lado. O resultado pode ser observado na listagem 9.26. 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 9.26: '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 9.27: 'Ainda outra figura'

Deixamos para o leitor a tarefa de simular o programa com diferentes valores e observar os resultados obtidos. A figura 9.15 ilustra o caso da chamada do programa com `figura_inc_lado_ang(100,120,5,3)`.

Pirâmide

Suponhamos que alguém nos diz que pretende um programa para desenhar o objecto que vemos na figura 9.16.

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 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 9.17 tenta clarificar a questão.

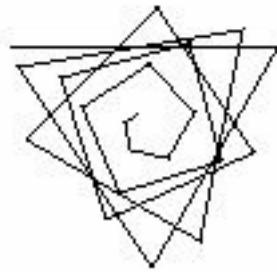


Figura 9.15: Variando o lado e o ângulo

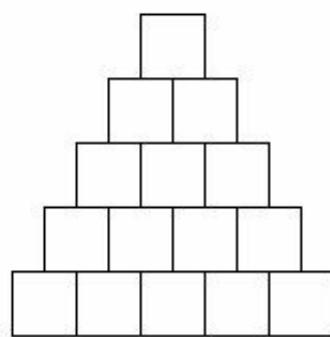


Figura 9.16: Que linda pirâmide

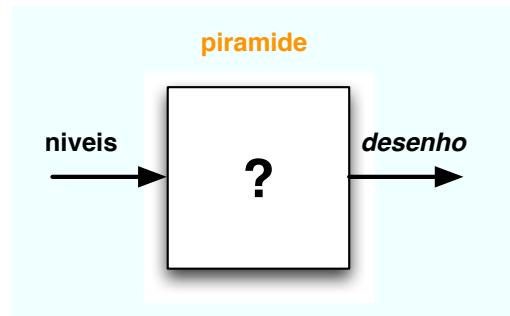


Figura 9.17: Um desenho diz mais do que mil palavras?

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 9.18)! 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?

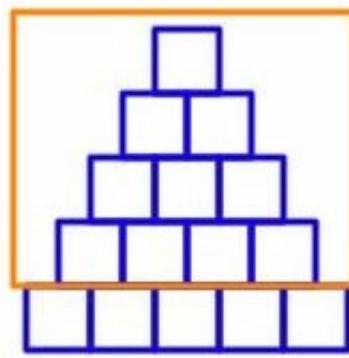


Figura 9.18: Pirâmide: pensar recursivo

Desta discussão decorre uma solução simples que o programa 9.28 propõe.

```

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 9.28: '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 9.29.

```

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 9.29: '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 9.30.

```

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 9.30: '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 9.19 mostra um exemplo de resultado quando o lado vale 20 unidades¹³.

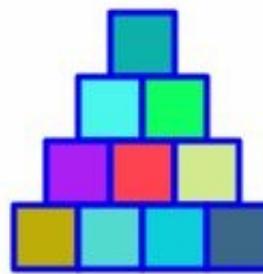


Figura 9.19: Uma pirâmide colorida

9.3 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 obter uma solução recursiva? Vamos decompor o problema nos seguintes sub problemas: gerar os anagramas para a palavra inicial sem o primeiro carácter (sub problema semelhante) para depois inserir o carácter inicial em **todas** as posições possíveis de **todos** os anagramas gerados. É esta solução que apresentamos na listagem 9.31.

```

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):

```

¹³Este valor pode ser tornado variável, levando a uma pequena modificação no código.

```

8     resp.append(perm[:pos] + cad[0] + perm[pos:])
9     return resp

```

Listagem 9.31: 'Anagrama'

Os leitores que gostam de usar listas por compreensão podem chegar a outra solução com a que apresentamos na listagem 9.32.

```

1 def anag(cad):
2     if cad == '':
3         return [cad]
4     else:
5         return [perm[:pos] + cad[0] + perm[pos:] for perm in anag(
6             cad[1:])] for pos in range(len(perm)+1)]

```

Listagem 9.32: '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 9.33 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 9.33: 'Permutações'

O problema do ordenamento de uma sequência é um problema 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.

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 9.20). O caso de base ocorre quando as sequências têm tamanho unitário, situação em que estão forçosamente ordenadas. Notar o uso da função `deepcopy`.

[Ordenamento](#)

[Ordenamento
por Fusão](#)

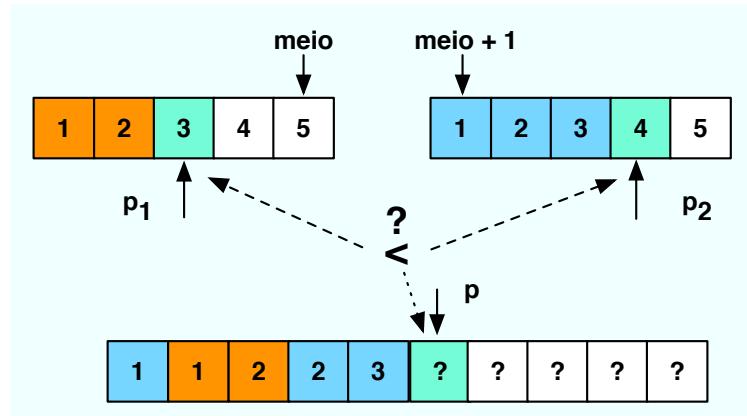


Figura 9.20: Ordenamento por Fusão

```

1 def ordena_fusao(seq,esquerda,direita):
2     """Dividir a sequência ao meio. Ordenar cada uma das partes
3         separadamente.
4     Depois fundir as duas partes"""
5     if esquerda == direita:
6         return [seq[esquerda]]
7     else:
8         seq=deepcopy(seq)
9         meio = (esquerda + direita)//2
10        seq_esq=ordena_fusao(seq,esquerda,meio)
11        seq_dir=ordena_fusao(seq,meio+1,direita)
12        return fusao(seq_esq,seq_dir)
13
14 def fusao(seq1,seq2):
15     seq=[]
16     p_1=0
17     p_2=0
18     while (p_1 < len(seq1)) and (p_2 < len(seq2)):
19         if seq1[p_1] < seq2[p_2]:
20             seq.append(seq1[p_1])
21             p_1 = p_1 + 1
22         else:
23             seq.append(seq2[p_2])
24             p_2 = p_2 + 1
25         if p_1 == len(seq1):
26             seq.extend(seq2[p_2:])

```

```

26     else:
27         seq.extend(seq1[p_1:])
28     return seq

```

Listagem 9.34: 'Ordenamento por Fusão'

O ordenamento rápido¹⁴ 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 9.21). Ao contrário do ordenamento por fusão aqui o ordenamento é **feito no lugar**, não sendo por isso necessário proceder a cópias da sequência.

[Ordenamento Rápido](#)

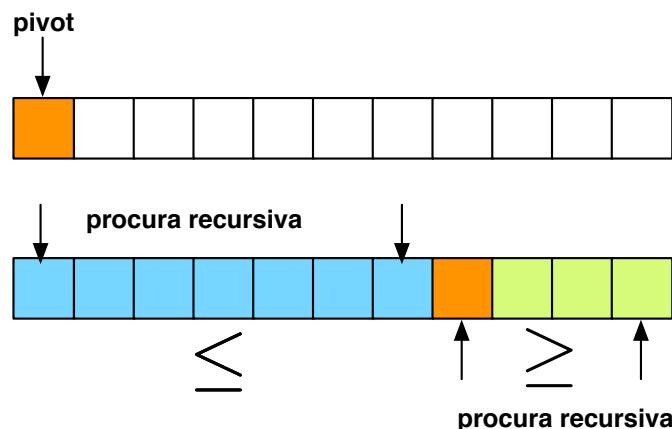


Figura 9.21: Ordenamento Rápido

```

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

```

¹⁴Em inglês *quicksort*.

```

10 def particao(lista, esquerda, direita):
11     """Divide a lista em duas metades em torno de um elemento (
12         pivot).
13         no final todos os elementos menores (maiores) ou iguais ao
14             pivot estão à
15             esquerda (direita) da lista. Devolve o índice que separa a
16                 parte esquerda da direita"""
17 pivot=lista[esquerda]
18 p_esq=esquerda
19 p_dir=direita
20 while True:
21     while lista[p_dir] > pivot:
22         p_dir = p_dir - 1
23     while lista[p_esq] < pivot:
24         p_esq = p_esq + 1
25     if p_esq < p_dir:
26         lista[p_esq],lista[p_dir] = lista[p_dir],lista[p_esq]
27         p_esq=p_esq + 1
28         p_dir=p_dir - 1
29     else:
30         return p_dir

```

Listagem 9.35: 'Ordenamento Rápido'

Fractais

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 9.1). 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 sierpinski(tamanho,nivel):
2     if nivel == 0:
3         return True
4     else:
5         for i in range(3):
6             sierpinski(tamanho/2,nivel -1)
7             fd(tamanho)

```

8 **rt(120)**

Listagem 9.36: 'Sierpinski Gasket'

Na figura 9.22 pode ver-se o resultado de executar o programa com lado 200 e nível 4.

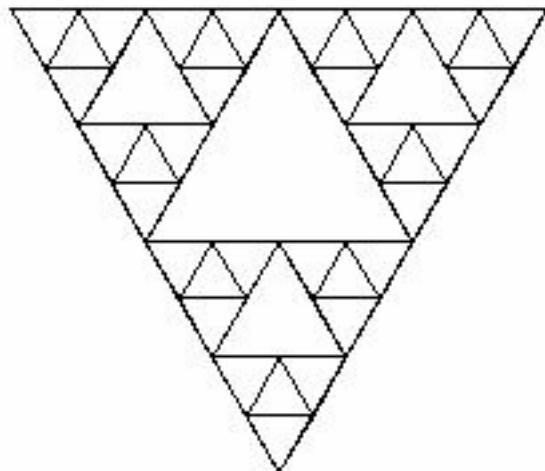


Figura 9.22: Sierpinski: 200,4

9.4 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 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.

Existem também comandos adicionais com que se podem anotar as so- [Memorização](#)

luções 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 9.23 ilustra o problema.

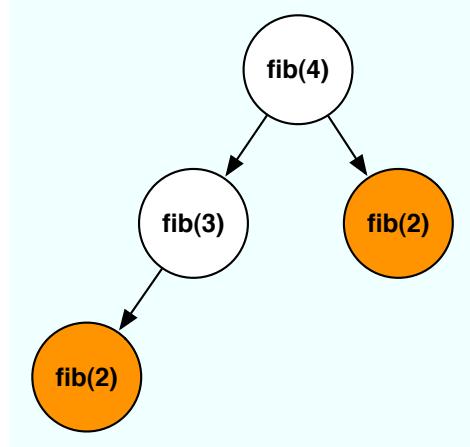


Figura 9.23: Fibonacci: cálculos duplicados

Para nos ajudar a resolver esta questão vamos usar uma técnica conhecida por memorização¹⁵. 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.

```

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
  
```

¹⁵Esta ideia é usada de modo geral em programação dinâmica



módulo functools

O mecanismo de memorização pode ser implementado recorrendo ao módulo `functools`. Vejamos como:

```

1  from functools import lru_cache
2
3  @lru_cache(maxsize=None)
4  def fib(n):
5      if n < 2:
6          return n
7      return fib(n-1) + fib(n-2)
8
9  if __name__ == '__main__':
10     for i in range(1,11):
11         print(fib(i))

```

Criamos uma cache, neste caso com tamanho ilimitado, onde são guardados os resultados que vão sendo calculados. Esta facilidade, é mais geral do que o mecanismo acima apresentado, sendo usado em vários tipos de aplicações, como por exemplo na busca de documentos na web evitando duplicações de acesso. O leitor interessado deve consultar o manual da linguagem.

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**¹⁶. 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 9.37).

```

1  def fact_rec(n):
2      if n == 0:
3          return 1
4      else:
5          return x * fact_rec(n - 1)

```

Listagem 9.37: De novo o factorial

¹⁶No 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*.

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 9.38.

```

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 9.38: Tipo de Dados Pilha

A fazermos o comando `pilha=Stack()` associamos ao nome *pilha* um objecto do tipo *Stack*¹⁷. A construção da solução passa por substituir cada 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 9.39 mostra o resultado do processo.

```

1 def fact_it(n):

```

¹⁷Em termos concretos criamos um objecto de valor lista vazia.

```

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

```

Listagem 9.39: 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 nível 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 9.1 F

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 9.2 F

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 9.3 F

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 9.4 F

Escreva um programa que dados dois conjuntos determina se um está incluído no outro.

Exercício 9.5 F

Escreva um programa que dados dois conjuntos determina a sua intersecção.

Exercício 9.6 D

O conjunto potência de um dado conjunto é o conjunto de todos os seus subconjuntos. Implemente o programa que calcula o conjunto potência para um dado conjunto.

Exercício 9.7 D

Suponha que desenha n ovais no plano que satisfazem as seguintes condições:

- as ovais intersectam-se duas a duas em, exactamente, dois pontos
- nunca três ovais se encontram no mesmo ponto

Quantas regiões distintas no plano são criadas por essas ovais? Encontre a expressão genérica para um dado n e escreva o respectivo programa. Nota: É encontrar esse valor para pequenos valores de n : $n = 1$ são duas, $n = 2$ são quatro, $n = 3$ são 8, como ilustra a figura 9.24.

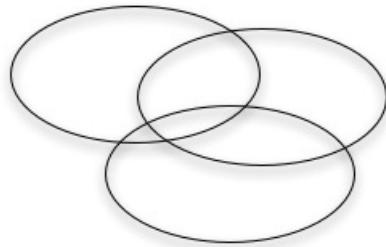


Figura 9.24: Ovais no plano. Caso de $n = 3$

Exercício 9.8 Módulo turtle F

Retome o exemplo da listagem 9.27 e faça variar os próprios incrementos. Simule para alguns valores.

Exercício 9.9 Módulo turtle F Queremos usar o módulo **turtle** para desenhar uma árvore simples como a indicada na figura 9.25.

Desenvolva o respectivo programa recursivo. A figura 9.26 dá uma ideia do processo gerativo.

Exercício 9.10 Módulo turtle M

A árvore do exercício 9.4 não é muito *natural*. Uma maneira de desenhar árvores mais interessantes consiste em desequilibrar a sub árvore direita e a

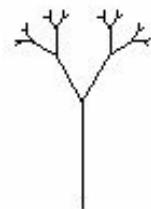


Figura 9.25: Uma árvore recursiva

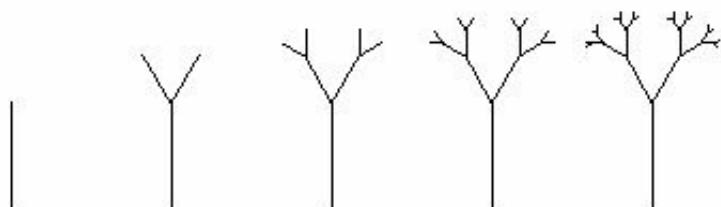


Figura 9.26: O processo

esquerda fazendo, por exemplo os ramos de uma maiores do que a outra. A figura 9.27 ilustra um resultado quando o lado esquerdo é duplo do lado direito. Tente criar o programa recursivo que permite obter estes desenhos.

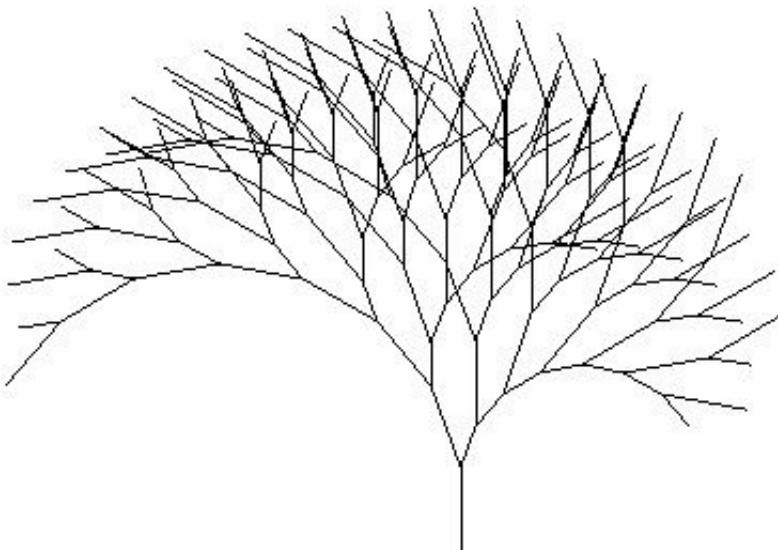


Figura 9.27: Uma árvore mais realista

Exercício 9.11 M

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 9.12 F

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 9.13 M

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 9.14 D

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 2×2 Strassen encontrou um conjunto de fórmulas 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 9.15 D

Um autómato finito é uma máquina de estados usada em várias aplicações informáticas, como por exemplo para implementar um analisador léxical 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, Σ é o alfabeto de entrada, δ 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 9.28 mostra graficamente um autómato finito que reconhece cadeias de 1 e 0 em que o número de uns é par.

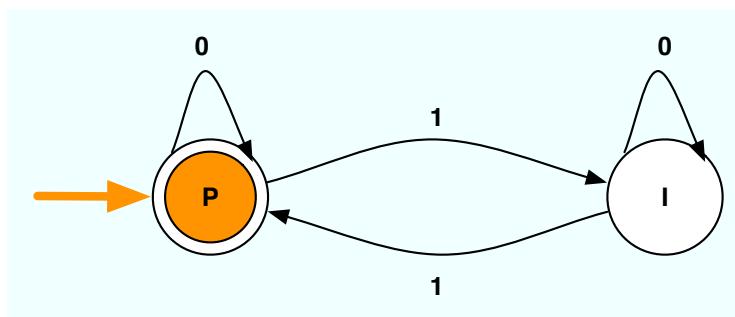


Figura 9.28: Detector de Paridade Par

Para lhe simplificar a vida na listagem 9.40 mostramos como se pode **representar** o autómato indicando explicitamente δ (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 9.40: 'Detector de Paridade Par'

Exercício 9.16 Módulo turtle D

A curva conhecida por Floco de Neve¹⁸ forma-se de acordo com uma regra simples. Num dado nível, cada lado é dividido em três partes iguais sendo retirada a parte do meio. De seguida, a partir das extremidades interiores formam-se dois novos segmentos, de tamanho igual a um terço do original, com uma inclinação de 60° em relação aos segmentos que se mantém e que se unem numa das extremidades. A figura 9.29 mostra o processo de construção desta curva para os níveis 1, 2 e 3. Usando o módulo **turtle** desenvolva um programa que permita desenhar a curvas. O valor do lado e o número de níveis a considerar são parâmetros do programa.

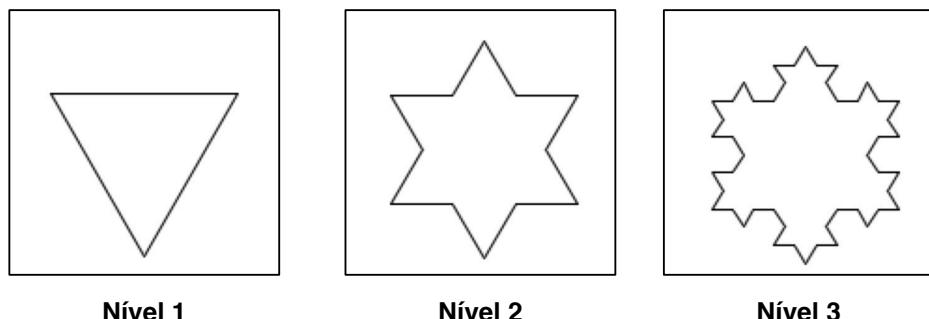


Figura 9.29: Floco de Neve

Exercício 9.17 Módulo turtle MD

A curva de Hilbert é um exemplo de curva de preenchimento de espaço. A sua ideia baseia-se em decompor a curva do nível n em quatro curvas de nível $n - 1$ ligadas entre si. A figura 9.30 mostra a curva desenhada caso só tenha um nível, dois níveis ou três níveis. Usando o módulo **turtle** desenvolva um programa que permita desenhar a curvas. O valor do lado e o número de níveis a considerar são parâmetros do programa.

¹⁸Do inglês *Snowflake*.

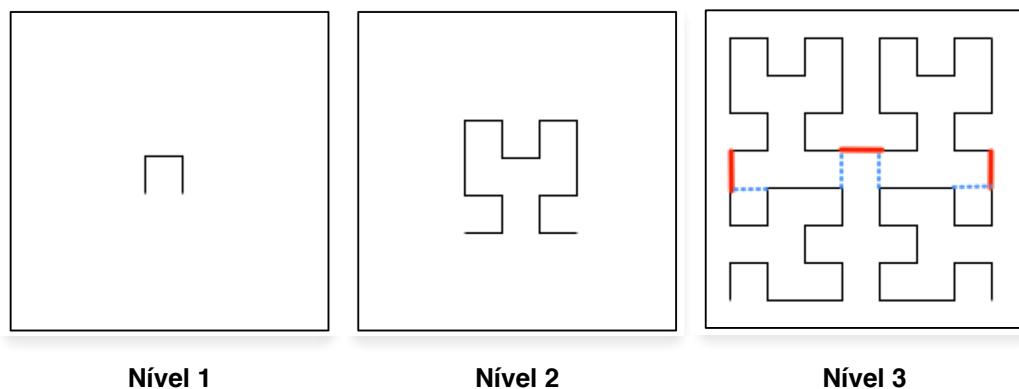


Figura 9.30: Curvas de Hilbert

Capítulo 10

Complementos

Objectivos

- ✓ Aprofundar o conceito de ambiente
- ✓ Completar a discussão sobre módulos
- ✓ Apresentar os vários modos definir os parâmetros dos programas
- ✓ Introduzir o conceito de alcance das variáveis
- ✓ Funções anónimas
- ✓ Iteradores e Geradores
- ✓ Introduzir elementos de programação funcional

10.1 Introdução

Agora que sabemos quais são os elementos de base de uma linguagem de programação e como os podemos usar na resolução de problemas, chegou o momento de aprofundar um pouco os nossos conhecimentos. Começamos por relembrar o que foi dito no capítulo 1 sobre o que é um programa e como os escrevemos. De um modo um pouco circular um programa em **Python** é um ficheiro de texto de extensão **py** contendo no seu interior um conjunto de elementos de programação¹ como se pode ver na figura 10.1.

No exemplo da figura, o ficheiro começa por **comandos para o sistema**, neste caso indicando como pode encontrar o interpretador **Python** num ambiente unix e qual a codificação dos caracteres utilizada. De seguida vem um

¹Tecnicamente estamos a falar de módulos.

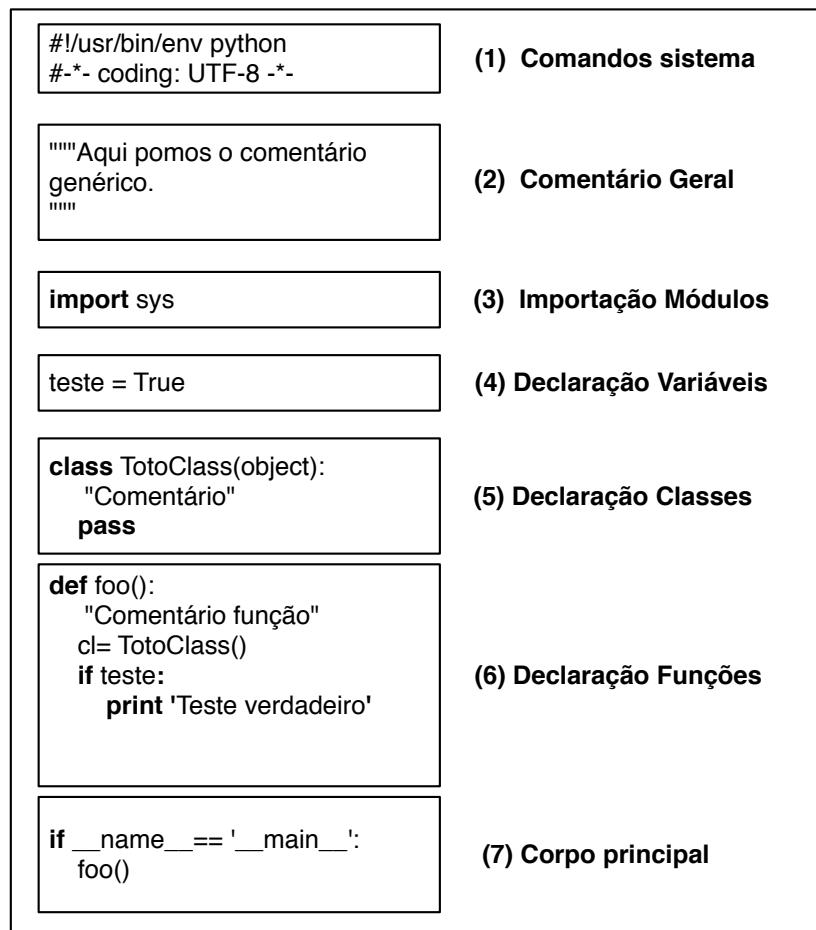


Figura 10.1: Um programa

comentário geral que permite explicar qual o objectivo do programa. Em terceiro lugar vem a importação de módulos, seja do sistema seja definidos pelo utilizador. Podemos **declarar** variáveis , classes e funções, o que fazemos de seguida. Finalmente, temos uma parte que designamos livremente por **programa principal** e que envolve uma execução condicional. Este modo de organizar os nossos ficheiros não é rígido, embora seja o que se aconselha. Claro que há partes que têm que estar nas posições relativas indicadas, como os comandos do sistema, a documentação e o corpo principal.

Cadeias de Caracteres de Comentário

Todas as cadeias de caracteres que aparecem associadas a um objecto **antes** de que qualquer código executável são conhecidas por cadeias de caracteres de documentação^a, e são tratadas automaticamente pelos sistemas de documentação como o **PyDoc**. Por exemplo imaginemos que temos o seguinte ficheiro:

```
1      """
2      Para testar a documentação.
3
4      minha_doc.py
5      """
6
7      __author__ = 'Ernesto Costa'
8
9      __version__ = 'June 2013'
10
11     def toto(n):
12         """ Imprime o dobro de n."""
13         print(2 * n)
14
15
16     if __name__ == '__main__':
17         toto(5)
```

Suponhamos que lançamos o interpretador de **Python** e importamos o veja-se 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
   2336.1.00)]
3 Type "help", "copyright", "credits" or "license" for more
   information.
4 >>> import minha_doc
5 >>> print(minha_doc.__doc__)
6
7 Para testar a documentação.
8
9 minha_doc.py
10
11 >>> help(minha_doc)
12 Help on module minha_doc:
13
14 NAME
15     minha_doc - Para testar a documentação.
16
17 DESCRIPTION
18     minha_doc.py
19
20 FUNCTIONS
21     toto(n)
```

10.2 Ambiente e Alcance das Variáveis

Se há tema que já referimos muitas vezes é o de, em Python , tudo serem objectos: números, cadeias de caracteres, tuplos, listas e dicionários. Mas também ficheiros, módulos e definições. Tudo. Em geral esses objectos têm um ou mais nomes associados que nos permitem aceder ao objecto e aos seus atributos. Acontece que quando corremos um programa a partir do terminal ou através de um IDE, ou interagimos com o interpretador, uma associação entre os nomes e os objectos tanto pode ser criada, como pode ser alterada ou mesmo destruída. Assim é preciso saber, sempre que encontramos um nome, a que objecto se refere e quais os seus atributos. Os nomes existentes encontram-se agrupados em **espaços de nomes**. Existe um espaço de nomes onde se encontram os nomes dos objectos pré-definidos pelo sistema, denominado **builtins**. São também criados espaços de nomes quando importamos um módulo, ou quando chamamos uma definição. Existe uma relação **hierárquica** entre estes três categorias de espaços de nomes, como a figura 10.2 tenta ilustrar.

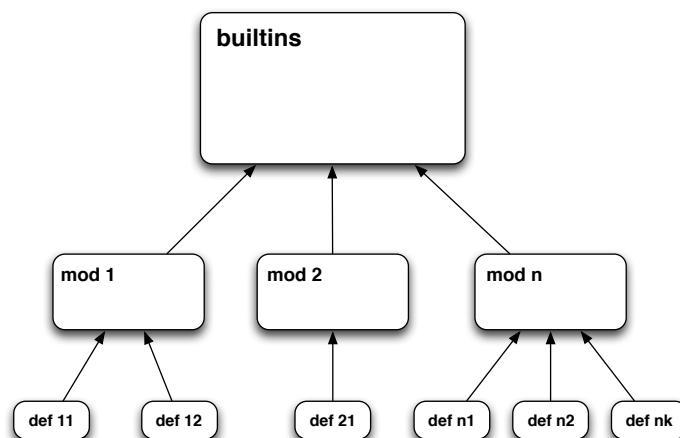


Figura 10.2: A hierarquia dos espaços de nomes

Esta hierarquia é fundamental quando se tratar de saber a que objecto concreto um nome está associado numa zona particular do código, como veremos de seguida.

O espaço de nomes liga-se ao **espaço dos objectos** através das referências aos objectos. Num dado instante estes dois espaços interligados definem um **ambiente**. É em função desta organização que se determina o **alcance**²

²Do inglês *scope*. Também há quem designe por domínio, contexto ou mesmo escopo.

de cada nome de variável que ocorre no nosso programa. Em Python o alcance de um nome é determinado de modo **estático** e depende do **bloco** de código onde foi feita a associação entre o nome e o objecto. Falamos ainda de **alcance lexical**.

[Alcance Lexical](#)

Modo interactivo

Vamos começar a concretizar estes conceitos considerando uma sessão no interpretador. No inicio da sessão existe desde logo um espaço de nomes, apelidado de **builtins**, que nos permite aceder aos comandos e objectos pré-definidos pelo sistema. Por exemplo, podemos calcular uma potência recorrendo à função pré-definida **pow** (linhas 7 e 8). O acesso a esse espaço de nomes pode ser feito através do nome **__builtins__**. Como se pode ver na listagem abaixo o nome designa um módulo cujos atributos podemos inspecionar graças ao comando **dir(__builtins__)**.

```

1 ernestojfcosta@Ernesto-Costas-Mac-Pro-8 ~ $ python
2 Python 3.2.3 (default, Sep  5 2012, 20:52:27)
3 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
   2336.1.00)] on darwin
4 Type "help", "copyright", "credits" or "license" for more
   information.
5 >>> pow
6 <built-in function pow>
7 >>> pow(2,3)
8
9 >>> __builtins__
10 <module 'builtins' (built-in)>
11 >>> dir(__builtins__)
12 ['ArithmeticError', 'AssertionError', 'AttributeError', ,
   BaseException', 'BufferError', 'BytesWarning', ,
   DeprecationWarning', 'EOFError', 'Ellipsis', ,
   EnvironmentError', 'Exception', 'False', ..., 'dir', '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']
13 >>>
```

A listagem foi editado tendo sido retirados muitos dos nomes pré-definidos.



`__builtins__` e `builtins`

O nome `__builtins__` remete para o módulo `builtins` (ver linhas 9 e 10), que implementa as operações e objectos pré-definidos (linhas 11 e 12). A relação pode ser directa ou indirecta, dependendo da implementação. No nosso caso é indirecta, podendo nós aceder ao dicionário que materializa a relação nomes - objectos através do atributo `__dict__` (linhas 15 e 16). As linhas 17 e 18 mostram como podemos invocar as operações. As restantes linhas ilustram o facto de existirem duas vias para chegar aos mesmos objectos.

```

1 ernestojfcosta@Ernesto-Costas-Mac-Pro-8 ~ $ python
2 Python 3.2.3 (default, Sep  5 2012, 20:52:27)
3 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
   2336.1.00)] on darwin
4 Type "help", "copyright", "credits" or "license" for more
   information.
>>> dir()
5 ['__builtins__', '__doc__', '__name__', '__package__']
>>> dir(__builtins__)
6 ['ArithError', 'AssertionError', 'AttributeError',
   ... 'pow', 'print', 'property', 'quit', 'range', 'repr',
   'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
   'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
   'vars', 'zip']
>>> __builtins__.__name__
7 'builtins'
>>> __builtins__.__doc__
8 "Built-in functions, exceptions, and other objects.\n\
   nNoteworthy: None is the 'nil' object; Ellipsis
   represents '...' in slices."
>>> pow(2,3)
9 8
10 >>> __builtins__.__dict__
11 {'bytearray': <class 'bytearray'>, 'IndexError': <class 'IndexError'>, 'all': <built-in function all>, 'help': Type help() for interactive help, or help(object) for help about object., 'vars': <built-in function vars>, 'SyntaxError': <class 'SyntaxError'>, 'UnicodeDecodeError': <class 'UnicodeDecodeError'>, 'pow': <built-in function pow>, 'RuntimeError': <class 'RuntimeError'>, 'float': <class 'float'>, 'MemoryError': <class 'MemoryError'>, 'StopIteration': <class 'StopIteration'>, 'globals': <built-in function globals>, ..., 'str': <class 'str'>, 'property': <class 'property'>, ..., 'OverflowError': <class 'OverflowError'>}
12 >>> __builtins__.__dict__['pow'](2,3)
13 8
14 >>> import builtins
15 >>> dir(builtins)
16
```

Fica também disponível um outro espaço de nomes designado por `__main__` que está hierarquicamente ligado ao espaço `__builtins__`. Podemos saber como está no início invocando o comando `dir()`.

```

1 >>> dir()
2 [ '__builtins__', '__doc__', '__name__', '__package__']
3 >>> __name__
4 '__main__'
5 >>>

```

A listagem mostra, os (pouco) nomes conhecidos no início³. Como se pode verificar está presente `__builtins__`. Depois destas considerações vamos procurar mostrar como fica o ambiente depois de invocado o interpretador no modo interactivo. A figura 10.3 mostra a situação.

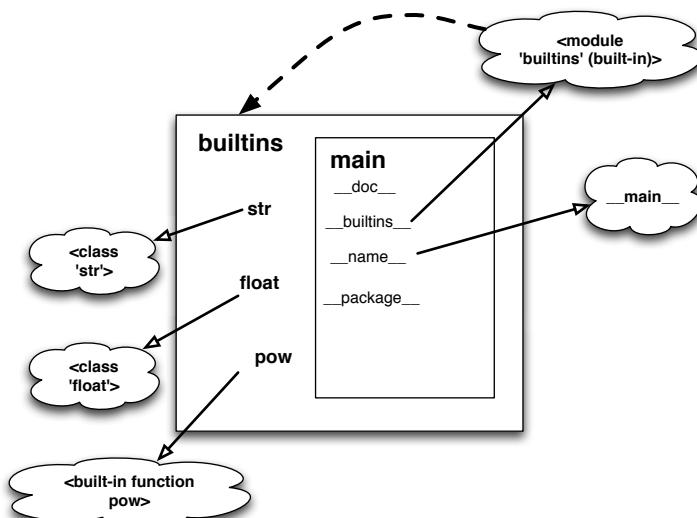


Figura 10.3: Ambiente inicial. Existem dois espaços de nomes, sendo que `__main__` está ligado a `__builtins__` (seta a tracejado). Sob a forma de nuvens aparecem os (descritores dos) objectos.

Vamos supor que agora começamos a interagir com o interpretador importando módulos, fazendo atribuições e definindo funções.

```

1 >>> import raizes
2 >>> dir(raizes)

```

³Para além dos nomes pré-definidos.

```

3 [ '__builtins__', '__cached__', '__doc__', '__file__', '__name__',
4   '__package__', 'cmath', 'raizes']
5 >>> x = 5
6 >>> def toto(n):
7     ...
8     ...
9     >>> dir()
10    [ '__builtins__', '__doc__', '__name__', '__package__', 'raizes',
11      'toto', 'x']
12 >>> raizes
13 <module 'raizes' from '/Users/ernestojfcosta/my_python_env/
14   raizes.py'>
15 >>> x
16 5
17 >>> toto
18 <function toto at 0x10953aea8>

```

São criados novos nomes que vão para o espaço de nomes `__main__`. São eles `raizes`, `x` e `toto`, como se observa após o comando `dir()` (linhas 8 e 9). Por outro lado, ao importamos o módulo foi criado um novo espaço de nomes com o nome do módulo. Ao inspecionarmos este espaço com o comando `dir(raizes)` ficamos a conhecer os nomes que habitam este espaço. Lá se encontra o nome `__builtins__`: o módulo está ligado hierarquicamente ao módulo com os elementos do sistema. Pode-se também observar (linhas 11 a 16) a ligação entre o espaço de nomes e o espaço dos objectos. A figura 10.4 procura dar uma ideia da situação neste momento.

O que acontece quando chamamos a definição `toto`? Por exemplo, se chamarmos com `toto(3)` não temos dúvidas que o resultado vai ser 6. Mas como evolui o nosso ambiente? A figura 10.5 explica a situação.

O espaço de nomes local `toto` é criado pela invocação da definição. Os parâmetros formais fazem parte desse espaço que está hierarquicamente subordinado ao espaço do módulo onde foi definido, neste caso `main`. Durante a activação é feita a associação entre o parâmetro formal e o objecto passado como argumento como se duma vulgar instrução de atribuição se tratasse. A partir daqui o programa é executado originando o cálculo do valor associado à expressão `2 * n` que é devolvido pela instrução de `return`. Devolvido o resultado a chamada da definição termina e igualmente é desfeito o espaço de nomes local.

E no caso de chamarmos a definição `raizes` do módulo com o mesmo nome? Este é uma situação um pouco mais complexa. Comecemos por ter em atenção o código do módulo `raizes`.

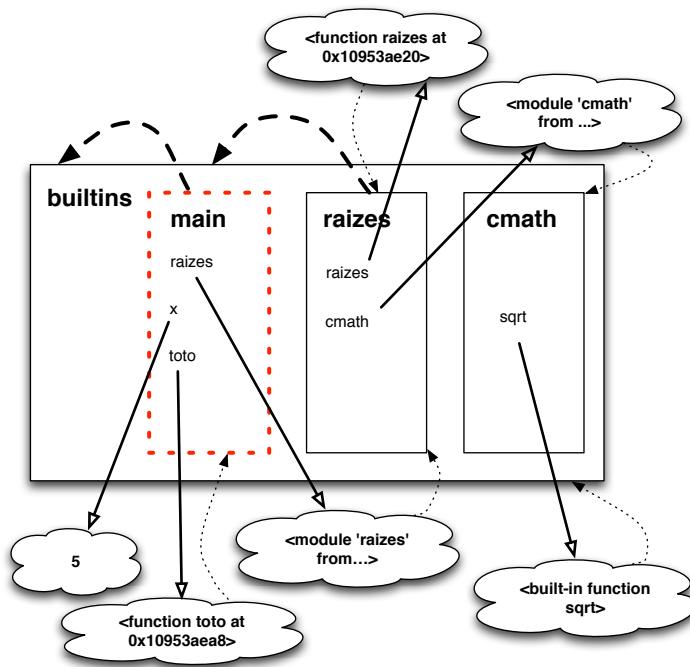


Figura 10.4: O ambiente com os vários espaços de nomes. O espaço de nomes `main` é o espaço activo e através dele accedemos aos diferentes objectos. Não é possível acceder directamente ao módulo `cmath` que foi importado pelo módulo `raizes`. A tracejado fino indicam-se as ligações dos descritores dos módulos e das definições e os seus espaços de nomes. Para não sobrecarregar a imagem, e sempre que tal não comprometa a compreensão da figura passaremos a omitir algumas destas ligações.

```

1 import cmath
2
3 def raizes(a,b,c):
4     """
5         Calcula as raízes de um polinómio do segundo grau.
6     """
7     comum =cmath.sqrt(b**2 - 4 * a * c)
8     print(comum)
9     raiz1= (-b + comum)/ (2 * a)
10    raiz2= (-b - comum)/ (2 * a)
11    return raiz1, raiz2
12

```

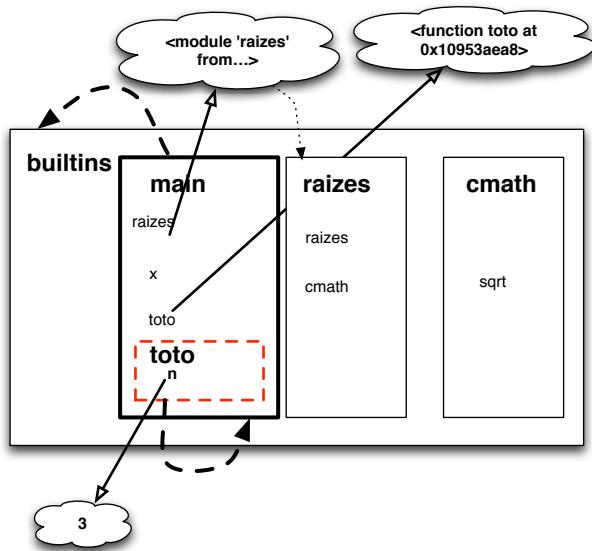


Figura 10.5: Criação de um espaço de nomes local à definição

```

13 if __name__ == '__main__':
14     print(raizes(4,8,2))

```

e admitamos que no interpretador efectuámos a chamada `raizes.raizes(4,8,2)`. Como no caso anterior é criado um espaço de nomes local para `raizes` ligado ao módulo onde foi definido. É feita a ligação entre os parâmetros formais e os parâmetros reais ou argumentos de um modo semelhante ao explicado para o caso anterior. Executado o corpo da definição é invocado o método `sqrt` do módulo `cmath` passando-lhe o argumento apropriado. Tal vai originar outro espaço de nomes local, neste caso para a raiz quadrada, ligado ao módulo onde foi definido. Executado o código do corpo de `sqrt` vai ser definido o objecto que vai ser associado ao nome `comum`, e este espaço de nomes é desfeito. A execução prossegue com o cálculo das duas raízes e devolução do respectivo valor, findo o que o espaço de nomes de `raizes` é, também ele, desfeito (ver figura 10.6.).

Estes exemplos concretizam o conceito de uma hierarquia de três níveis para os nomes.

Regras de alcance

Se em vez de estarmos no modo interactivo mandarmos correr um programa `Python` a partir do terminal ou de um IDE o que foi descrito não muda

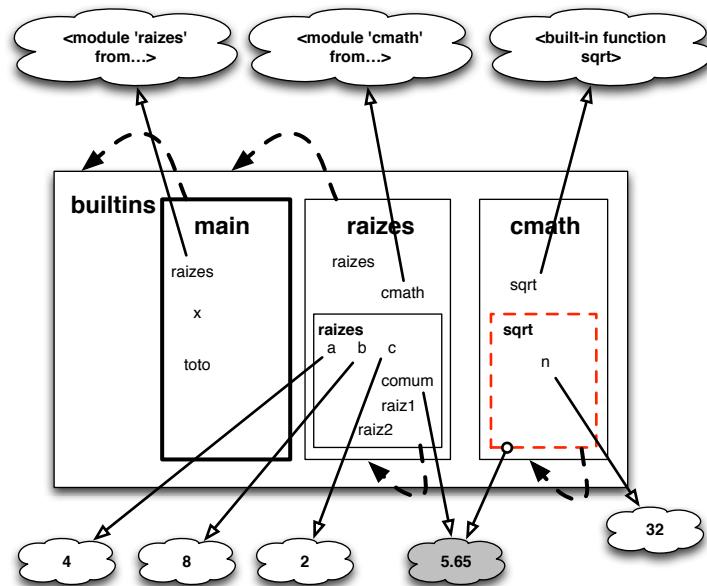


Figura 10.6: A partir do módulo `main` accedemos ao módulo `raizes`, e a partir deste ao módulo `cmath`. É então possível calcular qual o objecto que vai ser associado com o nome `comum`. Os espaços criados pelas chamadas das definições são locais ao módulo onde foram definidas. A figura ilustra o momento da devolução do resultado do cálculo da raiz.

substantivamente: o ficheiro `.py` tem a natureza de um módulo e vai assumir o papel do módulo `main`. Explicado isto estamos em condições de explicitar as **regras de alcance** ou de visibilidade dos nomes (de variáveis).

Regras de alcance

1. Por defeito, todos os nomes atribuídos numa definição pertencem ao espaço de nomes criado pela definição, e dizem-se **locais**. Isso aplica-se às atribuições explícitas (usando `=`) e implícitas (parâmetros e `import`);
2. Todos os nomes que não são locais e são atribuídos (explicita ou implicitamente) no nível do topo de um módulo, são acessíveis em todo o módulo, e dizem-se **globais**;
3. Todos os nomes pré-definidos pertencentes ao espaço de nomes `__builtins__` são acessíveis em todo o programa.

A figura 10.7 procura ilustrar a situação.

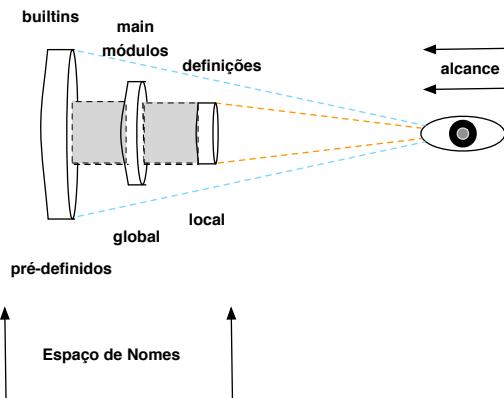


Figura 10.7: Alcance dos nomes (de variáveis).

Dito de outra maneira: quando se procura saber a que espaço de nomes pertence um nome que aparece numa definição, mas não é objecto de atribuição, a procura é feita do interior da definição para o exterior, parando no primeiro nível da hierarquia onde aparece o nome atribuído. Caso o nome que aparece na definição esteja associado a um objecto por atribuição, ele é local por defeito. Os nomes por nós atribuídos exteriormente a qualquer definição pertencem ao espaço de nomes do módulo onde aparecem.

Vejamos um exemplo concreto, admitindo que temos o ficheiro (módulo) `toto.py` com o seguinte código:

```

1 y = pow(2,3)
2
3 def toto(z):
4     x = z + y
5     return x
6
7 print(toto(5))

```

Neste exemplo temos objectos pertencentes aos três espaços de nomes, conforme se indica na figura 10.8.

A execução do programa dá origem aos cálculos da figura 10.9.

O exemplo que se segue permite ver o modo como se processa a resolução de conflitos entre nomes.

```

1 x = 4 # x global
2
3 def tata(n):
4     x = 2 # x local

```

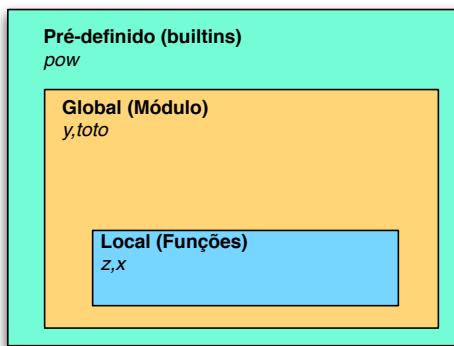


Figura 10.8: Os nomes e os respectivos espaços.

```

5   return x + n # x e n locais
6
7   print(tata(6)) # --> 8
8   print(x) # --> 4
  
```

No interior da definição o **x** utilizado é local. Observar também que a associação do **x** global ao objecto 2 não foi alterada.

Devemos reforçar a ideia de que cada função corre no contexto onde se deu a sua definição. Veja-se um exemplo em que usamos dois módulos em que há a importação de um deles pelo outro e em que ambos usam o mesmo nome de variável no topo do módulo.

```

1 # módulo tete.py
2 import tata
3
4 x = 10
5
6 def tete(m):
7     return tata.tata(m)
8
9 print(tete(5))
  
```



```

1 # módulo tata.py
2
3 def tata(n):
4     return x + n
  
```

A execução do primeiro módulo vai provocar uma chamada à definição **tata** do segundo que usa o nome **x** no seu corpo para definir o valor final.

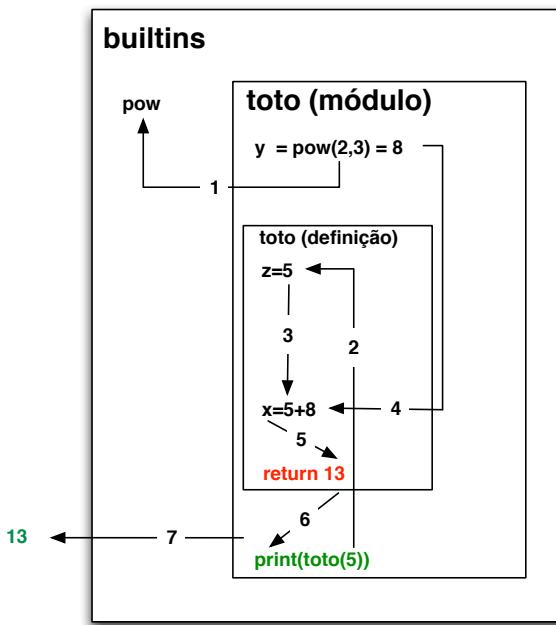


Figura 10.9: Calculando um valor

Esse valor é o que é encontrado no módulo em que está definido e não no módulo de onde foi feita a chamada, pelo que o resultado impresso vai ser 9 e não 14.

Declaração global

Do que foi explicado decorre de modo claro que não podemos alterar o objecto associado a uma variável global no interior de uma definição. Isto porque tal só pode ser feito por recurso da instrução de atribuição e isso, com acabámos de ver, transforma-a em variável local isolando-a do exterior. Mas devemos ter em atenção que as alterações **no-lugar**⁴ não as qualificam como locais. Vejamos a primeira situação.

```

1 lista = [1,2,3]
2 def toto(x):
3     lista = x
4     print(lista)
5 toto(5)    # ---> 5
6 print(lista) # ---> [1,2,3]
```

⁴Do inglês *in-place*.

Passamos a ter dois nomes `lista`, bem separados, um de alcance local e outro global. Vejamos o segundo caso.

```

1 lista = [1,2,3]
2 def toto(x):
3     lista.append(x)
4     print(lista)
5 toto(5)    # ---> [1,2,3,5]
6 print(lista) # ---> [1,2,3,5]
```

A variável `lista` mantém-se como global. O ponto a reter é que na primeira situação associamos através de uma referência e na segunda não. Mas e se queremos mesmo alterar a variável através de uma atribuição mas mantendo-a global? A resposta está no recurso à indicação explícita, **declarando**, que o nome se refere a uma variável global.

```

1 lista = [1,2,3]
2 def toto(x):
3     global lista # lista é global!
4     lista = x
5     print(lista)
6 toto(5)    # ---> 5
7 print(lista) # ---> 5!!
```

Executando o código verificamos que a alteração foi feita na variável global.

Variáveis não locais

Sabemos que as definições são objectos, logo tanto podem usadas como argumentos de outras definições, como podem ser devolvidas como resultado de definições. Podemos usar a primeira situação, por exemplo, para calcular o integral de uma função.

```

1 import math
2
3 def integral(func, a,b):
4     """
5         Calcula o integral definido de uma função entre a e b.
6     """
7     delta_x = 0.001
8     valor = 0
9     x = a
10    while x <= b:
```

```

11     valor = valor + func(x) * delta_x
12     x = x + delta_x
13     return valor
14
15 print(integral(math.sin,0,math.pi/2))

```

Podemos implementar o cálculo da derivada de uma função num ponto socorrendo-nos da segunda possibilidade.

```

1 import math
2
3 def derivada(func):
4     def d_func(x):
5         return (func(x + dx) - func(x)) / dx
6     return d_func
7
8 dx = 0.0001
9 a = math.pi/2
10 print(derivada(math.sin)(a))

```

Este último exemplo é interessante a vários títulos. Um deles é o facto de se ter duas definições imbricadas em que a mais externa devolve como valor a definição mais interna. Isto permite criar uma forma de **memória**. Vejamos como.

```

1 def externa(x):
2     def interna(y):
3         return y**x
4     return interna
5
6 if __name__ == '__main__':
7     func_1 = externa(2)
8     print(func_1(3)) # ---> 9
9     print(func_1(4)) # ---> 16
10    func_2 = externa(4)
11    print(func_2(3)) # ---> 81
12    print(func_2(4)) # ---> 256

```

Reparar como mesmo depois da definição **externa** ter terminado, devolvendo um objecto que é do tipo **function**, o valor associado a **x** não é esquecido. Estamos perante o que se designa em programação funcional por **fecho**⁵. Neste exemplo estamos a **referenciar** um objecto associado a um

⁵Do inglês *closure*.

nome definido na envolvente. E já sabemos que isso não causa problema. Mas e se o objectivo for **alterar** o objecto associado? Debrucemo-nos sobre um exemplo em que temos um contador que é incrementado de uma unidade cada vez que uma acção é realizada.

```

1 def externa(x):
2     def interna():
3         # realiza acção
4         x = x + 1
5         return x
6     return interna
7
8
9 if __name__ == '__main__':
10    func_1 = externa(0)
11    print(func_1()) #
12    print(func_1()) #

```

Quando executamos este código obtemos como resultado:

```

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 closure_1.py]
5 Traceback (most recent call last):
6   File "/Applications/WingIDE.app/Contents/MacOS/src/debug/
      tserver/_sandbox.py", line 12, in <module>
7   File "/Applications/WingIDE.app/Contents/MacOS/src/debug/
      tserver/_sandbox.py", line 5, in interna
8 builtins.UnboundLocalError: local variable 'x' referenced
   before assignment

```

A mensagem de erro diz-nos que tentámos referenciar uma variável local antes de ela ter sido associada a um objecto. E é verdade! A forma de resolver esta questão é declarar a variável `x` como `nonlocal`.

`nonlocal`

```

1 def externa(x):
2     def interna():
3         nonlocal x
4         x += 1
5         return x
6     return interna

```

```

7
8
9 if __name__ == '__main__':
10     func_1 = externa(0)
11     print(func_1()) # ---> 1
12     print(func_1()) # ---> 2

```

Devemos ter em atenção alguns aspectos relacionados com o uso de `nonlocal`. A variável declarada como `nonlocal` deve ter um valor associado numa definição envolvente àquela onde ocorre. Ilustremos situações que levam a um erro.

```

1 >>> def toto():
2 ...     nonlocal x
3 ...     return x
4 ...
5 SyntaxError: no binding for nonlocal 'x' found
6 >>>

```

Neste primeiro exemplo, o nome `x` não está associado a nada.

```

1 >>> x = 5
2 >>> def toto():
3 ...     nonlocal x
4 ...     return x
5 ...
6 SyntaxError: no binding for nonlocal 'x' found
7 >>>

```

Neste caso `x` tem um valor atribuído mas ao nível de topo do módulo `main`.

Vejamos um exemplo sem erros e em que ocorrem duas variáveis com o mesmo nome, mas habitando dois espaços de nomes diferentes: uma global e outra não local.

```

1 >>> x = 5
2 >>> def tete():
3 ...     x = 20
4 ...     def toto():
5 ...         nonlocal x
6 ...         x = 2 * x
7 ...         print(x)
8 ...     print(x)
9 ...

```

```

10 >>> tete()
11 20
12 >>> print(x)
13 5
14 >>>

```

Um outro exemplo com uma variável global e duas locais em definições diferentes.

```

1 x = pow(2,3)
2 >>> def toto(y):
3     ...     x = 2 * y
4     ...     def tete(z):
5         ...         x = z ** 2
6         ...         print(x)
7     ...     tete(x)
8     ...     print(x)
9 ...
10 >>> toto(2)
11 16
12 4
13 >>> x
14 8
15 >>>

```

Vejamos um exemplo um pouco mais interessante. Trata-se de uma aplicação que actualiza o saldo de uma conta bancária sempre que se retira uma determinada quantia de dinheiro.

```

1 def actualiza_saldo(saldo):
2     def retira(quantia):
3         nonlocal saldo
4         if quantia > saldo:
5             return 'Impossível realizar operação: saldo
6                 insuficiente.'
7         saldo = saldo - quantia
8         return saldo
9     return retira
10
11 if __name__ == '__main__':
12     conta_1 = actualiza_saldo(20)
13     conta_2 = actualiza_saldo(100)

```

```

13   print(conta_1(40)) # ---> Impossível realizar operação:
        saldo insuficiente.
14   print(conta_2(60)) # ---> 40

```

Neste exemplo criamos duas contas com um certo valor de saldo inicial. Apenas conseguimos retirar uma dada quantia se a conta tiver saldo superior ou igual à quantia pretendida. Neste exemplo combinamos o conceito de **fecho** com o de nome **nonlocal**.

As regras actualizadas

Agora que introduzimos a possibilidade de declarar nomes como global ou como nonlocal, dentro de definições, podendo estas estar imbricadas, vamos ter que actualizar as nossas regras. Eis como.

1. Todos os nomes atribuídos numa definição, e não declarados como global ou como nonlocal, pertencem ao espaço de nomes criado pela definição, e dizem-se **locais**. Isso aplica-se às atribuições explícitas (usando `=`) e implícitas (parâmetros e `import`);

A possibilidade de existirem definições imbricadas leva-nos também a uma visão diferente da busca do ambiente apropriado para cada nome, que ilustramos na figura 10.10.

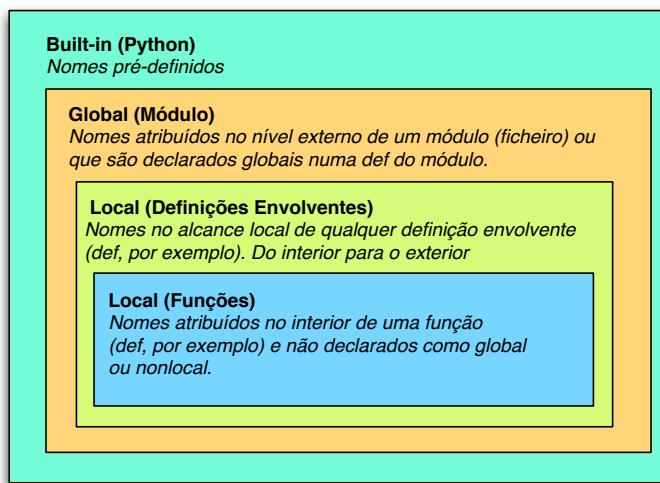


Figura 10.10: Alcance das variáveis

Fica claro que quando procuramos o espaço de nomes de uma variável, a busca é feita primeiro no espaço local de uma definição, depois de todas as

definições envolventes, a seguir no módulo e finalmente em `builtin`. Esta regra de resolução dos nomes é conhecida por **LEGB[?]**.

Um exemplo concreto. Consideremos o código:

```

1 # alcance global
2 y = pow(2,3) # pow é builtin
3
4 def func1():
5     # local de func1
6     k = 2
7     def func2(z):
8         #local de func2
9         w = y + k + z
10        return w
11    return 2 * func2(5)
12
13 print(func1()) # ---> 30

```

A figura 10.11 caracteriza a natureza das variáveis.

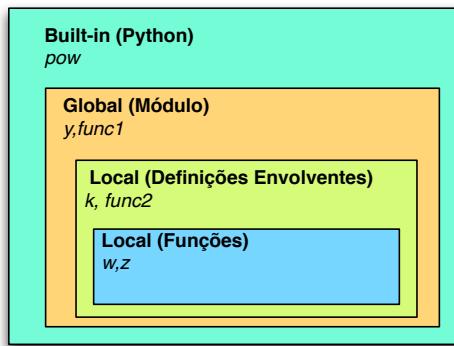


Figura 10.11: O seu a seu dono

Executando o código acima obtemos o resultado de 30 (ver figura 10.12).

Decoradores Vamos supor que queremos um programa que quando activado mostra como saída a chamada e o respectivo resultado. Para concretizar vejamos o pretendido no caso de termos uma função que calcula o cubo do seu argumento.

```

1 def cube(x):
2     return x**3

```

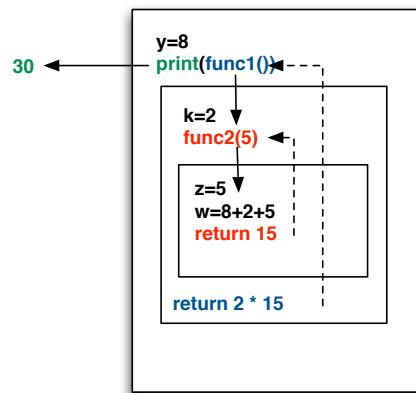


Figura 10.12: Como se chega ao resultado

```

3   if __name__ == '__main__':
4       print(cube(3)) # saída pretendida: cube ( 3 ) = 27
5

```

Podemos obter este efeito construindo uma definição auxiliar para traçar o resultado.

```

1 def tracer(func):
2     def wrapper(x):
3         print(func.__name__, '(', x, ') = ', end=' ')
4         return func(x)
5     return wrapper
6
7 def cube(x):
8     return x**3
9
10 cube = tracer(cube)
11
12 if __name__ == '__main__':
13     print(cube(3))

```

A definição `tracer` devolve uma função que envolve a sua função argumento, transformando-a, produzindo o resultado desejado. Mas para que tal aconteça é crucial a atribuição que aparece na linha 10. Este mesmo efeito pode ser obtido recorrendo a `decoradores`.

```

1 def tracer(func):
2     def wrapper(x):

```

```

3     print(func.__name__, ',(,x, ') = ', end=' ')
4     return func(x)
5
6
7
8
9 @tracer
10 def cube(x):
11     return x**3
12
13 if __name__ == '__main__':
14     print(cube(3))

```

Define-se um decorador através do uso do carácter @ seguido do nome do decorador. Imediatamente a seguir aparece a definição que vai ser objecto de transformação.

10.3 Módulos

Os módulos são uma componente essencial da linguagem Python⁶. Na sua versão mais comum são ficheiros de texto de extensão .py. Permitem manter o interpretador com uma dimensão razoável e serem usadas em caso de necessidade. Permitem também que o utilizador, definindo ele próprio módulos, aumente as capacidades da linguagem. Os módulos, vistos desta perspectiva são mais um exemplo do conceito de abstracção ao definir componentes que são reutilizáveis em diferentes contextos.

Os módulos para serem utilizados têm que ser importados, seja pelo utilizador seja por outro módulo. A importação é feita como já sabemos de acordo com a sintaxe:

```
1 import módulo
```

e envolve fundamentalmente três passos:

- Procura
- Compilação (Eventualmente)
- Execução

import

⁶Na realidade as linguagens modernas têm todos um conceito equivalente.

A procura envolve uma busca **ordenada** por caminhos possíveis para a localização dos módulos: na directória actual de trabalho, na variável **PYTHONPATH**, caso esta tenha sido definida no nosso ficheiro de configuração, na pasta **site-packages** e, finalmente em ficheiros de extensão **.pth** caso existam. Depois de ser encontrado o módulo é compilado para *byte-code*, caso ainda não o tenha sido ou a versão compilada seja mais antiga que o ficheiro fonte. Isto permite carregar de modo mais rápido o módulo, seguindo-se a execução do código nele contido. Depois de importado deste modo todos os nomes são atributos do módulo e passam a ser acedidos através da notação por ponto, envolvendo o uso do nome do módulo, seguido de um ponto, seguido do nome do atributo, como se procura ilustrar na figura 10.13, para o caso do método **sin** do módulo **math**. Tecnicamente o módulo define um **espaço de nomes**, organizado como um dicionário e que pode ser acedido através do atributo **__dict__** do módulo.

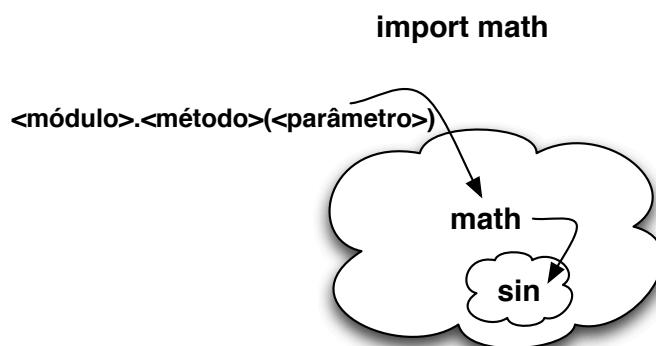


Figura 10.13: Importação simples

Devemos ter em atenção que os módulos só são importados uma vez. Admitamos que temos o seguinte módulo de nome **importa.py**, contendo apenas uma atribuição a uma variável **x**.

```
1 x = 'ah!ah!'
```

Vamos agora ver o que acontece quando importamos duas vezes o módulo e no meio alteramos o valor da variável.

```
1 >>> import importa
2 >>>importa.x
3 'ah!ah!'
```

```

4 >>> importa.x = 'aeiou' # foi alterado...
5 >>> import importa
6 >>> importa.x
7 'aeiou' #... manteve o valor

```

O valor alterado mantém-se mesmo depois da nossa tentativa de carregar de novo o módulo. Para recuperar o estado inicial do módulo este tem que ser re-importado, usando a função `reload` do modulo pré-definido `imp`.

```

1 >>> import imp # continuação da listagem anterior.
2 >>> imp.reload(importa)
3 <module 'importa' from 'importa.py'>
4 >>> importa.x
5 'ah!ah!' # cá está de novo!

```

Durante a execução de um módulo todas as atribuições feitas ao nível mais exterior, explícitas como em `x = 'Oops'`, ou implícitas como por exemplo numa definição `def toto(n)...` fazem com que os respectivos nomes passem a **atributos** do módulo sendo acedidos por recurso à notação por ponto e pertençam ao espaço de nomes do módulo. Exemplificando:

```

1 #módulo.atributo...
2 >>> import importa
3 >>> importa.x
4 'ah!ah!'
5 >>> importa.toto('ih!ih!', 3)
6 ih!ih!ih!ih!ih!

```

Admitamos que temos um módulo muito extenso mas que apenas queremos usar um dos seus métodos. Por exemplo, queremos apenas usar o método `sin` do módulo `math`. Não faz muito sentido importar **todo** o módulo, pelo que **Python** permite efectuar uma importação selectiva. A sua sintaxe é : `from`

```
1 from modulo import nome_1, nome_2, ... , nome_n
```

Na realidade esta importação selectiva é apenas uma facilidade que equivale, caso não existisse, a fazermos:

```

1 import módulo
2 nome_1 = módulo.nome_1
3 nome_2 = módulo.nome_2
4 del módulo

```

Ou seja, passamos a poder aceder aos atributos assim importados sem os prefixar com o nome do módulo, nome esse que deixa de ser conhecido como disponível, como se exemplifica na listagem seguinte.

```

1 >>> from importa import x
2 >>> x
3 'ah!ah!'
4 >>> importa.toto('uiui',2)
5 Traceback (most recent call last):
6   File "<string>", line 1, in <fragment>
7 builtins.NameError: name 'importa' is not defined
8 >>>

```

Podemos visualizar esta situação como se ilustra na figura 10.14.

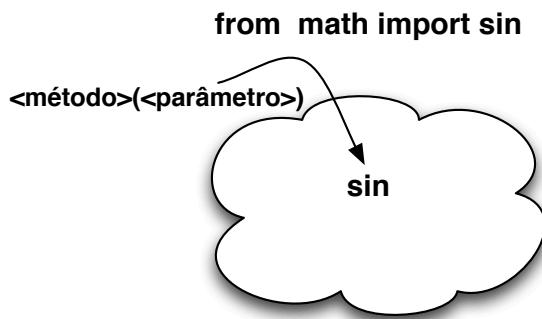


Figura 10.14: Importação selectiva

No limite podemos importar de forma selectiva . . . tudo, usando a construção:

```
1 from módulo import *
```

Esta forma de fazer não é aconselhável, pois passando a existir acessíveis de modo directo todos os nomes, as probabilidades de existirem conflitos com objectos distintos com o mesmo nome é grande. No caso de necessitarmos de um número grande de métodos de um módulos devemos usar a primeira versão de importação que nos obriga a prefixar o nome do método como o nome do módulo evitando assim os problemas.

Existe uma variante de importação que podemos usar quando o nome do módulo é muito comprido, ou quando queremos evitar um conflito com um nome existente no programa, ou ainda quando temos que percorrer directorias até chegar ao módulo. A sua sintaxe é simples:

```

1 import nome_grande_do_módulo as nome_p
2 from módulo import nome_grande_do_método as nome_p

```

Em todos estes casos o nome inicial é eliminado. Já encontrámos uma forma incompleta desta variante nalguns programas anteriores, nomeadamente quando fizemos:

```
1 import matplotlib.pyplot
2 plt = matplotlib.pyplot
```

É incompleta pois existe uma diferença: procedendo deste modo não eliminamos o nome inicial.

Uso e execução

Os módulos podem ter uma dupla utilização: podem ser importados, como foi descrito para um programa usar os métodos que disponibiliza, ou pode ser executado como sendo ele próprio um programa. Podemos escrever módulos que permitem esta **dupla** utilização através de um teste ao atributo `__name__` do modulo: quando executado este atributo é igual a `__main__`, quando é importado o atributo `__name__` é igual a o nome do módulo. Vamos um exemplo concreto. Comecemos pela definição do módulo.

```
1 """Dupla utilização de um módulo."""
2
3 print('O meu nome é: ', __name__)
4 print('e fui importado')
5
6 if __name__ == '__main__':
7     print('... e executado!')
```

Agora a simulação do que acontece quando importamos e quando executamos o módulo.

```
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 >>> import teste # Importar o módulo
5 O meu nome é: teste
6 e fui importado
7 >>>
8 Python 3.2.3 (default, Sep  5 2012, 20:52:27)
9 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
   2336.1.00)]
```

```

10 Type "help", "copyright", "credits" or "license" for more
   information.
11 [evaluate teste.py] # Executar o módulo
12 O meu nome é: __main__
13 e fui importado
14 ... e executado!

```

Pacotes

Ao longo do texto temos escrito programas que guardamos em ficheiros de extensão `.py` e que, por isso, são também módulos que podemos importar. Podemos também importar módulos guardados em pastas ou directorias, que designamos por **pacotes**. Um pacote não é mais do que uma pasta contendo no seu interior um ficheiro com o nome especial `__init__.py`, módulos e eventualmente outros pacotes^a. O ficheiro `__init__.py` pode estar vazio ou conter código importante que deve ser executado antes de usar os módulos do pacote. Pode também conter informação necessária sobre os módulos a importar quando se usa a versão `from .. import *`.

Suponhamos que temos a seguinte estrutura do pacote:

```

1 A/
2   __init__.py
3   toto.py
4   tete.py
5 B/
6   __init__.py
7   tata.py
8   titi.py

```

Podemos agora usar os vários tipos de `import` referidos.

```

1 import A.B.titi
2 import A.B.titi as titi
3 from A.B.titi import baba

```

^aComo se observa trata-se de uma definição recursiva.

10.4 Definições e Argumentos

Já referimos por diversas vezes o princípio PCAP (Primitivas, Composição, Abstracção, Padrão). As **definições** são um mecanismo fundamental de abstracção em programação. Permite-nos dominar a complexidade inerente ao desenvolvimento de programas, produzir código modular, reutilizável e capaz de resolver problemas gerais. Recordando a sua sintaxe:

```
1 def <nome>(<parâmetros>):
2     <corpo>
```

É preciso usar a palavra reservada **def** seguida do nome que demos ao programa, seguido de um parênteses de abertura, seguido dos **parâmetros formais**, eventualmente nenhum, separados por vírgulas, seguida do parênteses de fecho, seguida de dois pontos. Isto constitui o chamado **cabeçalho** da definição. Na linha seguinte, e avançado um certo número de posições para a direita, isto é, indentado, vem o **corpo** do programa, formado por instruções em Python . Sem isto a definição está mal feita e dará erro de sintaxe. As definições são objectos e por isso têm identidade valor e tipo.

```
1 >>> def toto(x):
2     ...     return 2*x
3 ...
4 >>> toto
5 <function toto at 0x103658a68>
6 >>> id(toto)
7 4351953512
8 >>> type(toto)
9 <class 'function'>
10 >>>
```

O nome que damos à função é apenas isso, um nome. Por outro lado, sendo uma definição um objecto pode ter atributos associados. Daí ser possível o que a listagem ilustra.

```
1 >>> dobro = toto
2 >>> dobro(5)
3 10
4 >>> id(dobro)
5 4351953512
6 >>> toto.ano_criacao = 2014
7 >>> toto.ano_criacao
8 2014
9 >>>
```

O primeiro exemplo mostra como podem existir mais do que um nome associado à mesma definição, enquanto que o segundo prova que podemos acrescentar atributos à definição. Podemos usar esta característica para, por exemplo, contar o número de vezes que um programa é chamado com se ilustra de seguida.

```
1  def fib(n):
2      fib.conta += 1
3      if n < 2:
4          return 1
5      else:
6          return fib(n-1) + fib(n-2)
7
8  if __name__ == '__main__':
9      fib.conta = 0
10     print(fib(3))
11     print(fib.conta)
```



Introspecção e Anotações

Quando definimos um atributo de uma função ele passa a ser um mais, juntando-se aos que estão pré-definidos.

```

1 >>> dir(toto)
2 ['__annotations__', '__call__', '__class__', '__closure__',
3  , '__code__', '__defaults__', '__delattr__', '__dict__',
4  , '__doc__', '__eq__', '__format__', '__ge__', '__get__',
5  , '__getattribute__', '__globals__', '__gt__', '__hash__',
6  , '__init__', '__kwdefaults__', '__le__', '__lt__',
7  , '__module__', '__name__', '__ne__', '__new__',
8  , '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
9  , '__sizeof__', '__str__', '__subclasshook__'],
10 'ano_criacao']
11
12 >>> toto.__name__
13 'toto'
14 >>> toto.__code__
15 <code object toto at 0x100ea2cf0, file "<stdin>", line 1>
16 >>> toto.__code__.co_varnames
17 ('x',)
18 >>> toto.__code__.co_argcount
19 1
20
21 >>>
```

A listagem mostra as diferentes propriedades do objecto `toto` (lá se encontra `ano_criacao`) e como se pode consultar o valor dessas propriedades. Um outro aspecto interessante consiste na possibilidade de acrescentar **anotações** às definições.

```

1 >>> def func(a: 'Nome', b: (1900,2100), c:float) -> int:
2 ...     return a + ' ' + str(b) + ' ' + str(c)
3 ...
4 >>> func('Ernesto', 1953, 100.54)
5 'Ernesto 1953 100.54'
6 >>> func.__annotations__
7 {'a': 'Nome', 'c': <class 'float'>, 'b': (1900, 2100),
8  'return': <class 'int'>}
9
10 >>>
```

As anotações não interferem com a execução do programa, mas é criado um dicionário, associado ao atributo `__annotations__`, em que as chaves são os argumentos e os valores as anotações. Este dicionário pode ser consultado.

Quando executamos código que contém uma definição, internamente é feita a associação do nome do programa com um objecto do tipo **function** que representa a definição, num dado ambiente. Esse objecto tem a identificação do tipo, os parâmetros formais, o corpo da definição e um ponteiro para o ambiente a que pertence. É pois semelhante ao que se passa quando usamos a instrução de atribuição como a figura 10.15 procura ilustrar para o caso da definição **toto**.

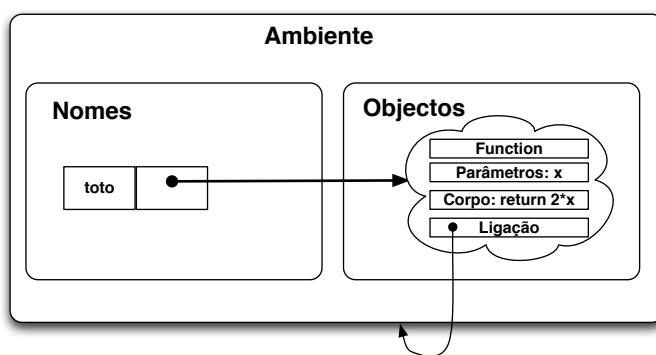


Figura 10.15: Definições e memória

Chamar uma Definição

Posição

As definições não tinham interesse nenhum se não fossem usadas, processo que normalmente se designa por **chamar** a definição. Chamamos o programa invocando o seu nome e comunicando os parâmetros reais em cada chamada concreta. No caso mais simples os parâmetros reais são associados aos parâmetros formais por **posição**: o primeiro parâmetro formal é associado ao primeiro parâmetro real, o segundo parâmetro formal é associado ao segundo parâmetro real, e assim sucessivamente. Decorre deste modo que o número dos parâmetros formais e reais tem que ser o mesmo. Tecnicamente a ligação é feita como se se efectuasse uma atribuição do parâmetro formal ao parâmetro real no início do corpo da definição. Considerando o segundo exemplo da listagem acima (linhas 3 e 4) é como se o corpo da definição fosse:

```

1 x = y
2 return 2*x
  
```

Os parâmetros reais são **expressões** que têm um objecto associado. Podem ser mais simples (um nome, uma constante) ou mais complexas, como se ilustra.

```

1 >>> dobro(5)
2 10
  
```

```

3 >>> y = 7
4 >>> dobro(y)
5 14
6 >>> dobro(5+y)
7 24
8 >>> dobro(dobro(5))
9 20
10 >>>

```

Vamos ver com um exemplo concreto, envolvendo o cálculo das raízes de um polinómio do segundo grau, esta ideia de associação e o que se passa antes, durante ou depois da chamada de uma definição. Comecemos pela definição.

```

1 import cmath
2
3 def raizes(a,b,c):
4     """
5     Calcula as raízes de um polinómio do segundo grau.
6     """
7     comum = cmath.sqrt(b**2 - 4 * a * c)
8     raiz1= (-b + comum)/ (2 * a)
9     raiz2= (-b - comum)/ (2 * a)
10    return raiz1, raiz2

```

Imaginemos agora que importamos o módulo e efectuamos duas chamadas à definição `raizes`.

```

1 >>> import raizes
2 >>> raizes.raizes(4,8,2) # 4,8 e 2 são os parâmetros reais
3 ((-0.29289321881345243+0j), (-1.7071067811865475+0j))
4 >>> p = 5
5 >>> q = 2
6 >>> r = 7
7 >>> raizes.raizes(p,q,r) # p,q,r são os parâmetros reais
8 ((-0.20000000000000001+1.1661903789690602j),
9 (-0.20000000000000001-1.1661903789690602j))
>>>

```

Antes da chamada, em qualquer das situações temos a situação da figura 10.16.

A situação é depois diversa no primeiro e no segundo casos. No primeiro, quando se dá a chamada os parâmetros formais são inicializados por associação do seu nome aos objectos correspondentes aos parâmetros reais, como

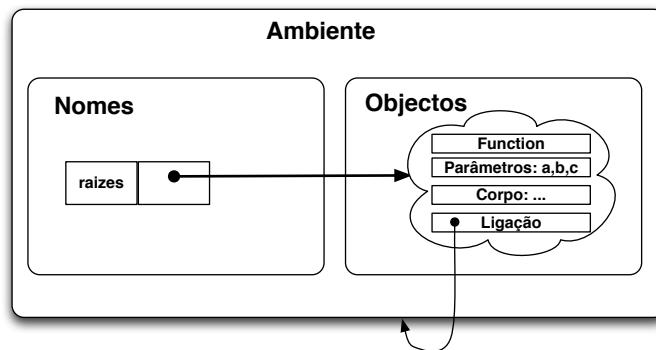


Figura 10.16: Antes

se mostra na figura 10.17

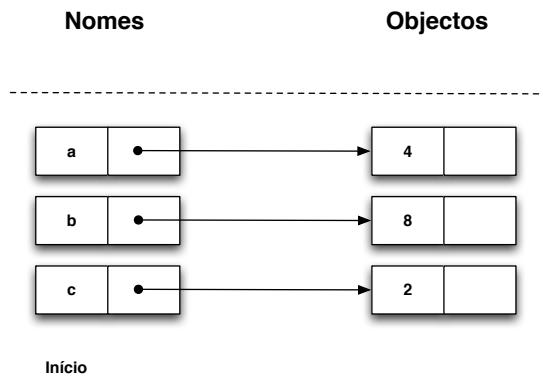


Figura 10.17: No início: caso 1

Já no caso 2 a situação é a retratada pela figura 10.18. Notar que, neste segundo caso, embora a execução da definição ocorra num ambiente diferente daquele onde ocorreu a definição, existe partilha de memória dos objectos.

Efectuados os cálculos, quando a chamada termina com a devolução do resultado através da instrução de `return`, voltamos mos dois casos à mesma situação. Isto é assim porque os objectos que foram passados como argumentos na chamada são imutáveis. Tecnicamente, podemos dizer que efectuámos uma passagem de parâmetros por valor.

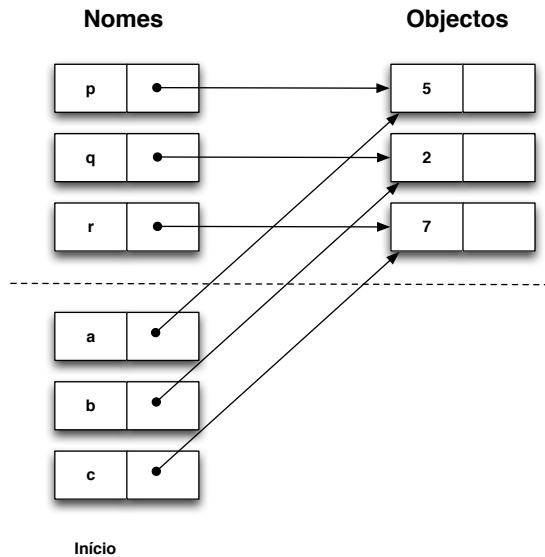


Figura 10.18: No início: caso 2

Mutabilidade

Mas o que acontece se os objectos forem mutáveis, como é o caso, por exemplo, das listas. Vejamos um exemplo.

```

1  >>> def teste(x,y):
2      ...     print 'entrada'
3      ...     x = 2 * x
4      ...     y[1] = 'toto'
5      ...     print id(x)
6      ...     print id(y)
7      ...     print 'saída'
8      ...     return x,y
9
10
10 >>> a = 4
11 >>> b = 3 * a
12 >>> l = [1,2,3]
13 >>> m = l
14 >>> p = l[:]
15 >>> id(a)
16 16793956
17 >>> id(b)
18 16793860

```

```

19 >>> id(1)
20 11797728
21 >>> id(m)
22 11797728
23 >>> id(p)
24 11760344
25 >>>

```

Como se pode ver a definição tem dois parâmetros, sendo que o segundo tem que ser uma sequência. Por outro lado, enquanto `l` e `m` referenciam o mesmo objecto, já `p` embora tenha o mesmo valor tem identidade diferente, tendo sido obtido fabricando uma cópia de `l`. Do ponto de vista gráfico (com simplificações óbvias) a situação antes da chamada é a da figura 10.19.

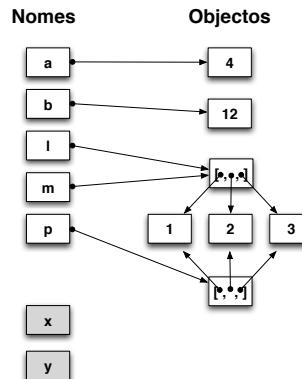


Figura 10.19: Antes da chamada

E no início? Temos a associação e uma vez mais ambientes diferentes mas partilha da memória dos objectos (ver figura 10.20.)

O programa prossegue alterando o valor de `x` e o de `l`. O efeito desta modificação é diferente pois um dos objectos é imutável e o outro mutável, situação que a figura 10.21 ilustra.

Mas quando o programa termina a sua execução as alterações ao objecto mutável são permanentes pelo que `l` e `m` foram afectados (ver figura 10.22).

Podemos dizer que no caso dos parâmetros que correspondem a objectos mutáveis a passagem de parâmetros é feita por referência pelo que as alterações são permanentes.

Se este efeito for indesejado existem algumas formas de o evitar. Uma óbvia é usar na chamada uma **cópia** do objecto obtida como o fizemos para

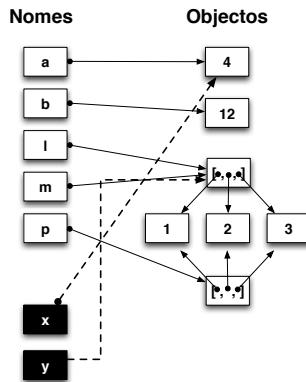


Figura 10.20: Chamada: associação dos parâmetros

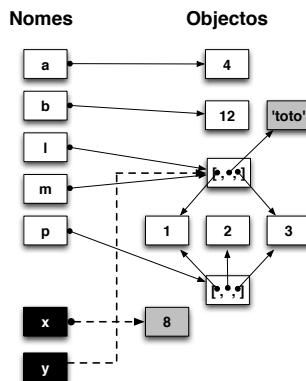


Figura 10.21: Durante a execução os objectos são alterados

p. Outra é efectuar uma cópia no início do corpo da definição (fazendo $y = y[:]$). Outra ainda é usar uma sequência sim mas imutável, transformando a lista num tuplo antes de chamar a definição. Em relação às duas primeiras alternativas é preciso ter algumas precauções pois o método indicado apenas faz uma cópia ao primeiro nível da estrutura. Todas as alterações mais profundas tornam-se permanentes. Neste caso a única alternativa segura é usar o método `deepcopy` do módulo do pré-definido `cópia`. A figura 10.23 mostra as diferenças.

Não somos obrigados a ligar os parâmetros por posição. Também o podemos fazer por nome. Um exemplo.

```
>>> def criatura(nome, especie, idade, peso):
```

nome

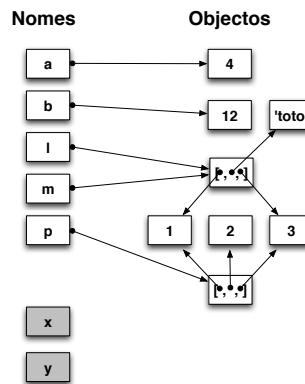


Figura 10.22: Após a execução

```

2 ...     return '%s (%s): %d anos, %d quilos' % (nome,especie,
3           idade,peso)
4 ...
5 >>> print criatura(nome='Ernesto',especie='Homo Sapiens',idade
6 =57,peso=65)
5 Ernesto (Homo Sapiens): 57 anos, 65 quilos
6 >>> print criatura(peso=65,nome='Ernesto',idade=57,especie='
7 Homo Sapiens')
7 Ernesto (Homo Sapiens): 57 anos, 65 quilos
8 >>>

```

Esta é uma forma de proceder conveniente quando temos muitos parâmetros e não queremos ter que fixar a ordem.

Parâmetros com valores por defeito

Sabemos da existência de operadores sobrecarregados. Por exemplo a listagem abaixo mostra como a operação realmente realizada depende do tipo dos objectos⁷.

```

1 >>> 5 + 6
2 11
3 >>> 'ab' + 'bc'
4 'abbc'
5 >>> [1,2,3] + ['a','b','c']

```

⁷Trata-se de uma manifestação de **polimorfismo** que discutiremos mais à frente no contexto da programação orientada aos objectos.

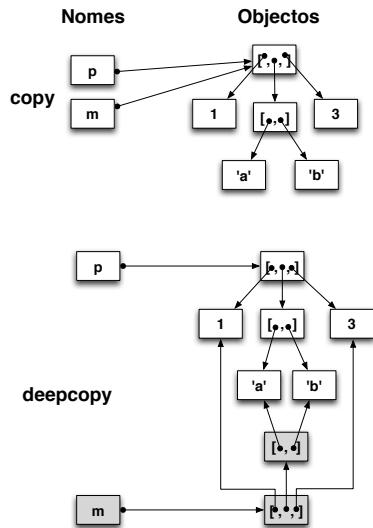


Figura 10.23: copy versus deepcopy

```

6 [1, 2, 3, 'a', 'b', 'c']
7 >>> (1,2,3) + ('a', 'b', 'c')
8 (1, 2, 3, 'a', 'b', 'c')
9 >>>

```

Questão distinta é a do uso de uma mesma operação mas com o número variável de argumentos. Já vimos que tal é possível quando discutimos o iterador `range`.

```

1 >>> list(range(5))
2 [0, 1, 2, 3, 4]
3 >>> list(range(2,6))
4 [2, 3, 4, 5]
5 >>> list(range(2,10,3))
6 [2, 5, 8]
7 >>>

```

Para se conseguir este efeito socorremo-nos de parâmetros inicializados com valores por defeito. Vejamos uma solução implementada por nós⁸.

```

1 def my_range(inicio,fim=None,incremento=1):
2     .....
3     A minha versão da função range para inteiros.
4     .....

```

⁸Esta solução não é um iterador.

```

5     if fim == None:
6         fim = inicio
7         inicio = 0
8     lista = []
9     proximo = inicio
10    while (incremento >= 0 and proximo < fim) or (incremento <
11        0 and proximo > fim):
12        lista.append(proximo)
13        proximo = proximo + incremento
14    return lista

```

Caso seja dado com apenas um argumento o valor dado passa a ser o valor final e o valor inicial é zero. O incremento é mantido a um. Com dois argumentos temos esses valores a corresponder ao início e ao fim, e o incremento é um. Se usarmos os três argumentos então serão esses os valores usados. O leitor pode testar e verificar que funciona.

A utilização de valores por defeito exige algum cuidado. Repare neste caso:

```

1 >>> def junta_valores(valores, inicio = []):
2     ...     inicio.extend(valores)
3     ...     return inicio
4 ...
5 >>> junta_valores([1,2,3])
6 [1, 2, 3]
7 >>> junta_valores([1,2,3])
8 [1, 2, 3, 1, 2, 3]
9 >>>

```

O que aconteceu? O problema está em que a ligação do valor por defeito é feita no momento da definição e não no momento da chamada. Deste modo sendo um objecto mutável apenas existirá uma instância do objecto qualquer que seja o número de vezes que efectuamos a chamada. Uma solução será usar como valor por defeito o objecto `None` e no início do código testar a existência ou não de valor.

```

1 >>> def junta_valores(valores, inicio=None):
2     ...     if inicio == None:
3     ...         inicio=[]
4     ...     inicio.extend(valores)
5     ...     return inicio
6 ...

```

```

7 >>> junta_valores([1,2,3])
8 [1, 2, 3]
9 >>> junta_valores([1,2,3])
10 [1, 2, 3]
11 >>>

```

Podemos misturar valores por defeito com argumentos chamados pelo nome.

```

1 >>> def func(x,y='toto',z='tete'):
2     ...     print(x,y,z)
3 ...
4 >>> func('ah!ah!')
5 ah!ah! toto tete
6 >>> func('ah!ah!', 'titi')
7 ah!ah! titi tete
8 >>> func('ah!ah!',z= 'tata')
9 ah!ah! toto tata
10 >>>

```

Número variável de argumentos

No exemplo anterior de valores por defeito podemos **simular** a existência de um número variável de parâmetros, mas à partida o seu número é fixo. Situação diferente é quando realmente não sabemos no momento da execução quantos argumentos vamos ter. Vejamos um exemplo em concreto: calcular o máximo de um conjunto de valores. Eis uma solução:

```

1 def my_max(*valores):
2     """
3         Qual o máximo de um número de valores?
4     """
5     if not valores:
6         return None
7     else:
8         maior = valores[0]
9         for val in valores[1:]:
10            if val > maior:
11                maior = val
12
13     return maior

```

A sintaxe foi aumentada prefixando o nome do parâmetro com um **asterisco**. Agora, aquando da chamada, todos os valores são juntos num tuplo e

o respectivo objecto associado ao (nome do) parâmetro.

```

1 >>> print(my_max(1,8,4,10,7,12,7))
2 12
3 >>> print(my_max(4,10,7))
4 10

```

Outra extensão consiste em usar um **duplo asterisco** antes do (nome do) parâmetro. No entanto este mecanismo só pode ser usado numa chamada com argumentos por nome e origina a construção de um dicionário que depois se associa ao parâmetro formal. Um exemplo simples.

```

1 >>> def func(**args):
2     ...     print args
3 ...
4 >>> func()
5 {}
6 >>> func(a=1,b=2)
7 {'a': 1, 'b': 2}
8 >>>

```

Existem limitações ao uso destes diferentes modos de considerar os argumentos. Por exemplo, não é possível:

```

1 >>> def teste(*frente, *cauda):...
2 >>> def soma(inicio=0, valores, fim=len(valores))...

```

Na primeira situação não se pode determinar por quem se dividiam os argumentos passados à definição. No segundo caso, não é possível estabelecer o valor por defeito do último argumento, no momento de criar a definição. É possível, no entanto, ter argumentos **depois** do uso de **um** asterisco desde que sejam passados por nome.

```

1 >>> def teste(a,*b,c):
2     ...     print(a,b,c)
3 ...
4 >>> teste(1,2,c=3)
5 1 (2,) 3
6 >>> teste(a=1,c=3)
7 1 () 3
8 >>>

```

Pode-se usar também o mecanismo de asterisco e de duplo asterisco na **chamada** da definição. Neste caso tem o efeito inverso do que tem na definição, ou seja, desconstrói a estrutura.

```

1  >>> def canal(r,g,b):
2      ...     print(r)
3      ...     print(g)
4      ...     print(b)
5
6  ...
7  >>> pixel = (128,255,64)
8  >>> canal(*pixel)
9  128
10 255
11 64
12 >>> dicio = {'a':1,'b':2,'c':3,'d':4}
13 >>> def exemplo(a,b,c,d):
14     ...     print(a,b,c,d)
15
16 >>> exemplo(**dicio)
17 1 2 3 4
>>>

```

No segundo caso ter em atenção que existem restrições: as chaves têm que ser cadeias de caracteres e os nomes usados como parâmetros formais têm que ter os mesmos caracteres que os da chave.

Sintetizamos tudo o que foi dito atrás sobre os argumentos das definições na tabela 10.1⁹

Sintaxe	Responsável	Significado
def func(nome)	Definição	Posição ou nome
def func(nome=valor)	Definição	Por defeito
def func(*nome)	Definição	Junta argumentos posicionais num tuplo
def func(**nome)	Definição	Junta argumentos com nome num dicionário
def func(*args,nome)	Definição	Por nome na chamada
func(valor)	Uso	Posição
func(nome=valor)	Uso	Nome
func(*sequência)	Uso	Individualmente por posição
func(**dict)	Uso	Individualmente como argumentos por nome

Tabela 10.1: Tipos de argumentos

⁹Adaptada de [?].

10.5 Iteradores e Geradores

Iteradores

Já sabemos que os ciclos `for` permitem repetir um número fixo de vezes uma sequência de instruções. O controlo do ciclo pode ser feito de várias maneiras. A listagem abaixo mostra alguns exemplos simples do uso da instrução `for`.

```

1 [GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build
2   2336.1.00)] on darwin
3 Type "help", "copyright", "credits" or "license" for more
4   information.
5
6 >>> for elem in [1,2,3]:
7     ...     print(elem, end=' ')
8
9 ...
10 1 2 3
11 >>> for elem in (1,2,3):
12     ...     print(elem)
13
14 ...
15 1 2 3
16 >>> for elem in {1,2,3}:
17     ...     print(elem, end=' ')
18
19 ...
20 1 2 3
21 >>> for ch in {1:'a',2:'b',3:'c'}:
22     ...     print(ch, end=' ')
23
24 ...
25 1 2 3
26 >>>

```

Como se pode ver, em todos os casos em cada execução do ciclo obtemos um elemento constitutivo do objecto seja ele do tipo lista, tuplo, conjunto, dicionário, ou cadeia de caracteres. O que estes objectos têm em comum, sejam eles colecções ou sequências, é precisamente serem de um tipo em que é possível obter os seus elementos um a um. Diz-se que os objectos são **iteráveis**. Se inspecionarmos estes objectos, ou os correspondentes tipo, veremos a existência do método especial `__iter__()` e/ou `__getitem__()`.

Iteráveis

```

1 >>> dir(list)

```

```

2 [ '__add__', '__class__', '__contains__', '__delattr__', '__
   _delitem__', '__doc__', '__eq__', '__format__', '__ge__',
   '__getattribute__', '__getitem__', '__gt__', '__hash__',
   '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
   '__len__', '__lt__', '__mul__', '__ne__', '__new__',
   '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
   '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
   '__str__', '__subclasshook__', 'append', 'count', 'extend',
   'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
3 >>>

```

Mas será que é possível obter o mesmo efeito mesmo **fora** de um ciclo? A resposta é positiva, mas necessita a transformação do objecto iterável num **iterador**: um objecto que tem associado um método **next()** que vai produzindo um novo elemento de cada vez até que estes se esgotem, altura em que é levantada a excepção **StopIteration**. Sem o sabermos já encontrámos iteradores no passado.

```

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 >>> f = open("/Users/ernestojfcosta/my_python_env/tres.txt")
5 >>> f
6 <_io.TextIOWrapper name='/Users/ernestojfcosta/my_python_env/
   tres.txt' mode='r' encoding='UTF-8'>
7 >>> type(f)
8 <class '_io.TextIOWrapper'>
9 >>> next(f)
10 'Este texto\n'
11 >>> next(f)
12 'tem apenas\n'
13 >>> next(f)
14 '3 linhas.'
15 >>> next(f)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
19 >>> z = zip((1,2,3),('a','b','c'))
20 >>> z
21 <zip object at 0x109cad908>

```

```

22 >>> next(z)
23 (1, 'a')
24 >>> next(z)
25 (2, 'b')
26 >>> next(z)
27 (3, 'c')
28 >>> next(z)
29 Traceback (most recent call last):
30   File "<stdin>", line 1, in <module>
31 StopIteration
32 >>> e = enumerate('Python')
33 >>> e
34 <enumerate object at 0x108ea4c80>
35 >>> next(e)
36 (0, 'P')
37 >>> next(e)
38 (1, 'y')
39 >>>

```

Sequências Virtuais

Actualmente em **Python** os iteradores são bastante usados. Uma boa razão é que produzem **sequências virtuais**, ou seja produz os elementos um a um e a pedido, com compreensível economia na memória necessária. Claro que continuamos a poder obter todos os elementos de uma só vez, necessitando apenas de envolver a chamada com **list**.

```

1 >>> list(open("/Users/ernestojfcosta/my_python_env/tres.txt"))
2 ['Este texto\n', 'tem apenas\n', '3 linhas.']
3 >>> list(zip((1,2,3),('a','b','c')))
4 [(1, 'a'), (2, 'b'), (3, 'c')]
5 >>>

```

Podemos construir iteradores a partir de objectos iteráveis, recorrendo ao método **iter**.

```

1 >>> l = iter([1,2,3])
2 >>> l
3 <list_iterator object at 0x109c4eb90>
4 >>> next(l)
5 1
6 >>> next(l)
7 2
8 >>> next(l)
9 3

```

```
10 >>> next(l)
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 StopIteration
14 >>> c = iter('abc')
15 >>> c
16 <str_iterator object at 0x109c4ec10>
17 >>> next(c)
18 'a'
19 >>> next(c)
20 'b'
21 >>> next(c)
22 'c'
23 >>> next(c)
24 Traceback (most recent call last):
25   File "<stdin>", line 1, in <module>
26 StopIteration
27 >>> d = iter({1:'a',2:'b',3:'c'})
28 >>> d
29 <dict_keyiterator object at 0x109c2cd60>
30 >>> next(d)
31 1
32 >>> next(d)
33 2
34 >>> next(d)
35 3
36 >>> next(d)
37 Traceback (most recent call last):
38   File "<stdin>", line 1, in <module>
39 StopIteration
40 >>> r = iter(range(3))
41 >>> r
42 <range_iterator object at 0x109c55f90>
43 >>> next(r)
44 0
45 >>> next(r)
46 1
47 >>> next(r)
48 2
49 >>> next(r)
50 Traceback (most recent call last):
```

```

51  File "<stdin>", line 1, in <module>
52 StopIteration
53 >>>
```

O último exemplo, mostra que `range` é um iterável. Convém perceber que existe uma diferença entre o objecto iterável e o iterador a que dá origem, como se exemplifica na listagem seguinte.

```

1  >>> l = [1,2,3]
2  >>> iter(l) is l
3  False
4  >>> t = (1,2,3)
5  >>> iter(t) is t
6  False
7  >>> d = {1:'a',2:'b',3:'c'}
8  >>> iter(d) is d
9  False
10 >>> r = range(3)
11 >>> iter(r) is r
12 False
13 >>> z = zip((1,2,3), ('a','b','c'))
14 >>> iter(z) is z
15 True
16 >>> f = open('/Users/ernestojfcosta/my_python_env/tres.txt')
17 >>> iter(f) is f
18 True
19 >>>
```

Podemos criar vários iteradores a partir do mesmo iterável. Vejamos como.

```

1  >>> l = [1,2,3]
2  >>> l_1 = iter(l)
3  >>> l_2 = iter(l)
4  >>> l_1
5  <list_iterator object at 0x108ebdc10>
6  >>> l_2
7  <list_iterator object at 0x108ebdc50>
8  >>> next(l_1)
9  1
10 >>> next(l_2)
11 1
12 >>>
```

```
13 >>> r = range(3)
14 >>> r
15 range(0, 3)
16 >>> type(r)
17 <class 'range'>
18 >>> r_1 = iter(r)
19 >>> next(r_1)
20 0
21 >>> next(r_1)
22 1
23 >>> r_2 = iter(r)
24 >>> next(r_2)
25 0
26 >>> next(r_1)
27 2
28 >>> next(r_2)
29 1
30 >>>
```

Iteráveis, iteradores e o ciclo for Estamos agora em condições de explicar o que se passa quando usamos um iterável no cabeçalho de um ciclo `for`: Internamente, no início o iterável é transformado num iterador por uma chamada ao método `iter`. Depois em cada passagem o método `next` permite obter o próximo elemento do iterador. Quando não há mais elementos é levantada uma exceção (`StopIteration`) que é apanhada e faz abandonar o ciclo. O código abaixo ilustra a ideia.

```
1 >>> l = iter([1,2,3])
2 >>> while True:
3     ...     try:
4     ...         e = next(l)
5     ...         print(e)
6     ...     except StopIteration:
7     ...         break
8 ...
9 1
10 2
11 3
12 >>>
```

O módulo itertools Existe em Python um módulo que nos facilita um número apreciável de iteradores para facilitar algumas operações. Alguns exemplos.

```
1  >>> import itertools
2  >>> iter_1 = itertools.count(3)
3  >>> next(iter_1)
4  3
5  >>> next(iter_1)
6  4
7  >>> next(iter_1)
8  5
9  >>> iter_2 = itertools.cycle([1,2,3])
10 >>> next(iter_2)
11 1
12 >>> next(iter_2)
13 2
14 >>> next(iter_2)
15 3
16 >>> next(iter_2)
17 1
18 >>> next(iter_2)
19 2
20 >>> next(iter_2)
21 3
22 >>> next(iter_2)
23 1
24 >>> iter_3 = itertools.repeat([1,2,3])
25 >>> next(iter_3)
26 [1, 2, 3]
27 >>> next(iter_3)
28 [1, 2, 3]
29 >>> iter_4 = itertools.accumulate([1,2,3])
30 >>> next(iter_4)
31 1
32 >>> next(iter_4)
33 3
34 >>> next(iter_4)
35 6
36 >>> next(iter_4)
37 Traceback (most recent call last):
```

```
38  File "<stdin>", line 1, in <module>
39 StopIteration
40 >>> iter_5 = itertools.chain([1,2,3],[4,5,6])
41 >>> next(iter_5)
42 1
43 >>> next(iter_5)
44 2
45 >>> next(iter_5)
46 3
47 >>> next(iter_5)
48 4
49 >>> next(iter_5)
50 5
51 >>> next(iter_5)
52 6
53 >>> next(iter_5)
54 Traceback (most recent call last):
55   File "<stdin>", line 1, in <module>
56 StopIteration
57 >>> iter_6 = itertools.product([1,2,3], 'abc')
58 >>> next(iter_6)
59 (1, 'a')
60 >>> next(iter_6)
61 (1, 'b')
62 >>> next(iter_6)
63 (1, 'c')
64 >>> next(iter_6)
65 (2, 'a')
66 >>> next(iter_6)
67 (2, 'b')
68 >>> next(iter_6)
69 (2, 'c')
70 >>>
```

Os primeiros três exemplos são iteradores que geram sequências infinitas. O quarto é um exemplo de um acumulador de somas parciais. O quinto junta todos os elementos dos iteráveis dados como argumento numa única cadeia de elementos. O sexto efectua o produto cartesiano dos argumentos. Existem muitos mais exemplos de iteradores que o módulo facilita e variantes dos exemplos apresentados. O leitor interessado deve consultar o manual da linguagem para conhecer a lista completa.

Geradores

Já estamos familiarizados com listas por comprehensão e o modo como melhoraram a legibilidade de um programa. Mas podemos juntar as ideias de listas por comprehensão à de iteradores para chegar ao novo conceito de **expressão geradora**. Sintaticamente é semelhante às listas por comprehensão com os parênteses rectos substituídos por parênteses curvos.

```

1  >>> l = [x*x for x in [1,2,3]]
2  >>> l
3  [1, 4, 9]
4  >>> type(l)
5  <class 'list'>
6  >>> type([x*x for x in [1,2,3]])
7  <class 'list'>
8  >>> e = (x*x for x in [1,2,3])
9  >>> e
10 <generator object <genexpr> at 0x108ea4d20>
11 >>> next(e)
12 1
13 >>> next(e)
14 4
15 >>> next(e)
16 9
17 >>> next(e)
18 Traceback (most recent call last):
19   File "<stdin>", line 1, in <module>
20 StopIteration
21 >>>
```

Como deve suspeitar, podem ser usadas num ciclo `for`.

```

1  >>> for i in (x*x for x in [1,2,3]):
2    ...     print(i)
3  ...
4  1
5  4
6  9
7  >>> list((x*x for x in [1,2,3,4,5] if x%2 == 0))
8  [4, 16]
9  >>>
```

O último exemplo mostra a natureza geral das expressões geradoras. As expressões geradoras não permitem múltiplos iteradores:

```

1 >>> g = (x*x for x in [1,2,3])
2 >>> g1 = iter(g)
3 >>> g2 = iter(g)
4 >>> g1 is g
5 True
6 >>> g2 is g
7 True
8 >>> next(g1)
9 1
10 >>> next(g2)
11 4
12 >>>

```

Uma questão que o leitor já se terá colocado é a de saber se é possível o programador definir ele próprio iteradores. A resposta é afirmativa. Vamos ver um método simples que faz aparecer o conceito de **função geradora**¹⁰. [Função Geradora](#) Trata-se de funções em que aparece a instrução `yield` em vez de `return`¹¹. Quando a instrução `yield` é executada é devolvido imediatamente um valor e a computação fica suspensa, com o contexto salvaguardado, até nova invocação que faz continuar o programa na instrução a seguir à instrução `yield`. O uso de funções geradoras permite a construção passo a passo de séries de valores, potencialmente infinitas. Comecemos por um problema velho conhecido mas agora com novas roupagens: a sequência de Fibonacci.

```

1 def fibonacci():
2     a,b = 0,1
3     while True:
4         yield b
5         a,b = b, a+b
6
7
8 if __name__ == '__main__':
9     fib = fibonacci()
10    print(next(fib)) # ---> 1
11    print(next(fib)) # ---> 1
12    print(next(fib)) # --> 2
13    print([next(fib) for i in range(10)]) # -->
[3,5,8,13,21,34,55,89,144,233]

```

¹⁰Como veremos mais adiante no texto existem outras formas de o fazer que se baseiam na ideia de os tipos serem implementados como classes.

¹¹Podem no entanto coexistir na mesma definição as duas instruções.

Como se pode observar não estão sendo gerados os diferentes valores da sequência. O último exemplo mostra como a sequência pode ser limitada. Um outro exemplo simples mostra como podemos gerar os sucessivos cubos, até um certo limite, dos inteiros naturais. Podemos observar que também aqui ultrapassado o limite é levantada uma exceção.

```

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 >>> def gera_cubos(n):
5     ...     for i in range(1,n):
6     ...         yield i**3
7 ...
8 >>> gc = gera_cubos(4)
9 >>> next(gc)
10 1
11 >>> next(gc)
12 8
13 >>> next(gc)
14 27
15 >>> next(gc)
16 Traceback (most recent call last):
17   File "<stdin>", line 1, in <module>
18 StopIteration
19 >>>
```

Crivo de Eratosthenes Ainda nos lembramos que os números primos são os números naturais que apenas admitem como divisores a unidade e eles próprios. Existe um método para calcular os números primos menores ou iguais a um certo número e que é atribuído ao matemático grego Eratosthenes. Ficou conhecido como o Crivo de Erastosthenes. Consiste no seguinte. Começar com a lista dos números entre 2 e o número pretendido. De seguida, eliminam-se da lista todos os números que são múltiplos do primeiro elemento da lista, neste caso 2. Passamos para o segundo elemento da lista modificada e repetimos o processo, ou seja eliminamos os seus múltiplos. Este processo repete-se até se chegar ao fim da lista. Vamos implementar este algoritmo recorrendo a funções geradoras. Temos que ter em atenção que nas funções geradoras os elementos não estão sendo produzidos um a um.

```

1 def crivo_era(lista):
```

```

1     primo = lista[0]
2     yield primo
3     nova_lista = exclui_multiplos(primo, lista)
4     for p in crivo_era_b(nova_lista):
5         yield p

```

O código é uma tradução (da explicação) do algoritmo: começamos por expor o primeiro elemento da lista que é necessariamente primo (linhas 2 e 3), depois retiramos todos os múltiplos desse numero (linha 4), para finalmente calcularmos **recursivamente** os restantes números primos (linhas 5 e 6). Testemos o programa.

```

1 if __name__ == '__main__':
2     lista = list(range(2,100))
3     crivo = crivo_era_b(lista)
4     print(next(crivo)) # chamada 1
5     for i in range(10): # chamada 2
6         print(next(crivo),end=' ')
7     print()
8     print(next(crivo)) # chamada 3
9     print(next(crivo)) # chamada 4
10
11
12 # resultado --->
13 [evaluate geradores.py]
14 2 # resultado 1
15 3 5 7 11 13 17 19 23 29 31 # resultado 2
16 37 # resultado 3
17 41 # resultado 4

```

Nesta sessão começamos por solicitar um elemento, depois dez e depois mais dois. É claro que delegámos parte da solução para a função **exclui_multiplos** que é necessário implementar. uma possibilidade, muito simples, é a da listagem seguinte.

```

1 def exclui_multiplos_b(n,lista):
2     return [num for num in lista if (num % n) != 0]

```

O leitor atento notará que esta solução não é muito satisfatória. Na realidade passamos como argumento **toda** uma lista e depois pedimos **alguns** dos primeiros números primos dessa lista. Se, por exemplo, quisermos os primeiros 25 números primos e usarmos um ciclo for para obter o resultado é gerado um erro. O ideal é então termos uma lista de números **potencialmente infinita** que permite pedir qualquer número de números primos.

Mas como o fazer? Bom . . . com geradores! Comecemos por criar o programa que nos permite retirar os múltiplos de uma lista um a um.

```

1 def exclui_multiplos(n, lista):
2     for i in lista:
3         if (i % n):
4             yield i

```

Testemos o resultado.

```

1 ex_mul = exclui_multiplos(3,range(2,20))
2 while 1:
3     try:
4         print(next(ex_mul), end=' ')
5     except StopIteration:
6         break
7
8 # resultado
9 [evaluate geradores.py]
10 2 4 5 7 8 10 11 13 14 16 17 19

```

Mas se é verdade que produzimos um gerador para excluir os múltiplos, no entanto o argumento `lista` continua sem ser potencialmente infinito. Vamos então tratar dessa questão.

```

1 def inteiros_desde(n):
2     while True:
3         yield n
4         n += 1

```

Deixamos ao leitor o cuidado de testar este último programa. Agora só é preciso juntar as peças do puzzle.

```

1 def inteiros_desde(n):
2     while True:
3         yield n
4         n += 1
5
6 def exclui_multiplos(n, lista):
7     for i in lista:
8         if (i % n):
9             yield i
10
11 def crivo_era(lista):

```

```

12     primo = next(lista)
13     yield primo
14     nao_divide_primo = exclui_multiplos(primo,lista)
15     for p in crivo_era(nao_divide_primo):
16         yield p

```

Agora pode obter a quantidade de números primos que desejar sem ter medo de o programa rebentar.

```

1 if __name__ == '__main__':
2     primos = crivo_era(inteiros_desde(2))
3     for i in range(100):
4         print(next(primos), end=' ')
5
6 # resultado dos 100 primeiros números primos
7 [evaluate geradores.py]
8 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79
   83 89 97 101 103 107 109 113 127 131 137 139 149 151 157
   163 167 173 179 181 191 193 197 199 211 223 227 229 233 239
   241 251 257 263 269 271 277 281 283 293 307 311 313 317
   331 337 347 349 353 359 367 373 379 383 389 397 401 409 419
   421 431 433 439 443 449 457 461 463 467 479 487 491 499
   503 509 521 523 541

```

Para concluir, podíamos criar uma definição geral para o problema.

```

1 def primeiros_n(func,n):
2     return [next(func) for i in range(n)]

```

Agora, se quisermos, por exemplo os 7 primeiros números primos é só chamar:

```

1 if __name__ == '__main__':
2     print(primeiros_n(inteiros_desde(2),7))

```

A solução apresentada geral e não serve apenas apenas para resolver o problema dos números primos. Ela aplica-se a qualquer sequência construída com geradores.

10.6 Funções de Ordem Superior

Funções Anónimas Os mecanismos de abstracção são um elemento fundamental de uma linguagem de programação moderna. Em Python já vimos que um mecanismo de abstracção procedural são as definições criadas com

recurso a **def**. Usamos definições quando temos várias ocorrências do mesmo código ou de código semelhante, recorrendo nesta última situação à introdução dos chamados parâmetros formais. Uma outra razão para o recurso a definições é a possibilidade de reutilização de código, e ainda o facto de introduzirem modularidade nos programas tornando-os mais legíveis e fáceis de manter. Mas em **Python** existe um outro mecanismo de abstracção procedimental conhecido por funções anónimas ou funções **lambda**¹². A sua sintaxe é muito simples:

```
1 lambda <parâmetros formais> : <expressão>
```

O que caracteriza fundamentalmente as funções **lambda** é o facto de não terem um nome associado (serem anónimas), sendo definidas em tempo de execução, e não necessitarem de uma instrução de **return**¹³. Por outro lado, existe a limitação de o corpo da função ser uma expressão, não podendo conter instruções de controlo ou a definição de objectos locais que não os parâmetros. São por isso, em geral pequenas sendo usadas, por exemplo, quando o código que implementam só vai ser usado uma vez ou quando o uso de uma **def** está sintaticamente proibido. Deve ficar claro que a existência de funções **lambda** não é uma necessidade teórica, no sentido de que podemos implementar sempre as nossas soluções sem a elas recorrer, mas quando bem usada resulta em código de melhor qualidade.

Eis alguns exemplos simples de utilização de funções **lambda**.

```
1 >>> (lambda x: x**2)(2)
2 4
3 >>> for i in range(4):
4 ...     (lambda x: x**2)(i)
5 ...
6 0
7 1
8 4
9 9
10 >>> lista = [(lambda x: x**2)(i) for i in range(4)]
11 >>> lista
12 [0, 1, 4, 9]
13 >>> lista_2 = [lambda x: x**2, lambda x: x**3, lambda x,y: x+y
]
```

¹²Este tipo de abstracção é herdeiro do cálculo lambda desenvolvido por Alonzo Church e que influenciou as linguagens ligadas ao paradigma funcional, como LISP ou Scheme.

¹³No entanto, nada impede que se use um **return**.

```

14  >>> lista_2[0](4)
15  16
16  >>> lista_2[1](4)
17  64
18  >>> lista_2[2](4,5)
19  9
20  >>> dicio = {1:lambda x: x**2, 2:lambda x: x**3,3: lambda x,y:
21      x+y}
22  >>> dicio[1](4)
23  16
24  >>> dicio[2](4)
25  64
26  >>> dicio[3](4,5)
27  9
28  >>> def aplica(n):
29      ...
30      return (lambda x: x + n)
31  ...
32  >>> aplica(4)(2)
33  6
34  >>>

```

Os dois primeiros exemplos (linhas 1 a 9) são o mais básico possível. O exemplo seguinte (linhas 10 a 12) mostra o uso conjugado com listas por comprehensão. Continuamos com dois exemplos envolvendo listas e dicionários que ilustram o conceito de tabelas de salto¹⁴. Terminamos com o caso de uma definição que devolve como resultado uma função **lambda**. Claro que são possíveis utilizações envolvendo formas simples de condicionais ou definições **lambda** imbricadas.

```

1  >>> (lambda x,y: x if x < y else y) (33,66)
2  33
3  >>> (lambda x,y: x if x < y else y) (66,33)
4  33
5  >>> (lambda x : (lambda y: x + y))(50)(100)
6  150
7  >>> def aplica_func(func, *args):
8      ...
9      return func(*args)
10 >>> aplica_func(lambda x, y: x + y,10,20)
11 30
12 >>>

```

¹⁴Do inglês *jump tables*.

O primeiro caso é possível porque se trata de uma **expressão** condicional. No último exemplo da listagem vemos que também podemos passar uma função **lambda** como argumento.

map, filter e reduce A partir do momento em que as definições são objectos de primeira classe ficamos a saber que elas podem aparecer como argumento de outras definições¹⁵. É pois possível usar o mecanismo de composição de funções a que a matemática nos habituou.

```

1 >>> from math import sqrt, sin, pi
2 >>> sin(pi/2)
3 1.0
4 >>> sqrt(sin(pi/2))
5 1.0
6 >>> sqrt(sin(pi/4))
7 0.8408964152537145
8 >>> sin(sqrt(pi))
9 0.9797359324758174
10 >>> sqrt(sqrt(16))
11 2.0
12 >>>

```

Podemos imaginar mecanismos de composição mais sofisticados.

```

1 >>> def comp_func(f,g):
2 ...     return lambda x: f(g(x))
3 ...
4 >>> new_f = comp_func(sin,cos)
5 >>> new_f(pi/4)
6 0.6496369390800625
7 >>>

```

Exploraremos agora o uso de funções como argumento. Vamos supor que queremos desenvolver um programa para calcular o somatório dos inteiros naturais até um certo limite, ou seja, queremos calcular:

$$somat = \sum_{i=1}^n i$$

Exemplos de programas simples para o fazer são dados na listagem abaixo.

```

1 def somat_int_1(n):
2     soma = 0

```

¹⁵E/ou ser devolvidas como resultado.

```

3     conta = 0
4     while conta < n:
5         conta += 1
6         soma += conta
7     return soma
8
9 def somat_int_2(n):
10    soma = 0
11    for i in range(n+1):
12        soma += i
13    return soma
14
15 def somat_int_3(n):
16    return sum(range(n+1))

```

Admitamos que agora o que pretendíamos era a soma dos quadrados dos inteiros naturais até um dado n . Podemos usar o modelo anterior para resolver a nova questão.

```

1 def somat_quad_1(n):
2     soma = 0
3     conta = 0
4     while conta < n:
5         conta += 1
6         soma += conta**2
7     return soma
8
9 def somat_quad_2(n):
10    soma = 0
11    for i in range(n+1):
12        soma += i**2
13    return soma
14
15 def somat_quad_3(n):
16    return sum([ i**2 for i in range(n+1)])

```

Como se observa foi preciso apenas uma pequena adaptação, sendo que no caso da terceira versão a solução é um pouco mais complexa. Podemos imaginar outro tipo de somatórios, de logaritmos, de senos, e muitos mais. Para cada caso teremos que fazer uma pequena adaptação. Manda o princípio da economia e da reutilização de código que se pense numa solução *geral*, que podemos descrever pela fórmula:

$$somat = \sum_{i=n}^m func(i)$$

Note-se que generalizámos a função e o limite inferior. Esta generalização resolve-se incluindo dois novos parâmetros, precisamente o que define o limite inferior e o que define a função a usar. Os programas.

```

1  import math
2
3  def somat_1(func,n,m):
4      res = func(n)
5      conta = 1
6      while (conta < (m-n+1)):
7          conta += 1
8          res += func(conta)
9      return res
10
11 def somat_2(func,n,m):
12     res = func(n)
13     for i in range(n+1,m+1):
14         res += func(i)
15     return res
16
17 def somat_3(func,n,m):
18     return sum([func(i) for i in range(m+1)])
19
20
21 if __name__ == '__main__':
22     print(somat_1(lambda x: x,1,10))
23     print(somat_2(lambda x: x,1,10))
24     print(somat_3(lambda x: x,1,10))
25
26     print(somat_1(lambda x: x**2,1,10))
27     print(somat_2(lambda x: x**2,1,10))
28     print(somat_3(lambda x: x**2,1,10))
29
30     print(somat_1(math.sqrt,1,10))
31     print(somat_2(math.sqrt,1,10))
32     print(somat_3(math.sqrt,1,10))

```

Note-se o modo como se processa a chamada das funções nos dois primeiros casos por recurso a funções anónimas. Analisando agora as três versões

o leitor não deixará de concordar que a primeira é a menos interessante e a terceira a mais elegante. Python fornece uma construção que nos permite facilmente construir soluções que passam pela aplicação da mesma função a uma sequência de objectos. Chama-se `map` e é herdeira da programação funcional.

```

1 def somat_map(func,n,m):
2     return sum(map(func, range(n,m+1)))
3
4 if __name__ == '__main__':
5     print(somat_map(lambda x: x,1,10))
6     print(somat_map(lambda x: x**2,1,10))
7     print(somat_map(math.sqrt,1,10))

```

`map` é um iterador.

```

1 >>> map(lambda x : x**2,[1,2,3])
2 <map object at 0x109298b50>
3 >>> quad = map(lambda x : x**2,[1,2,3])
4 >>> next(quad)
5 1
6 >>> next(quad)
7 4
8 >>> next(quad)
9 9
10 >>> next(quad)
11 Traceback (most recent call last):
12   File "<string>", line 1, in <fragment>
13 builtins.StopIteration:

```

Podemos user `map` para mapear funções com mais do que um argumento:

```

1 >>> sol_2 = map(pow, [1,2,3],[4,5,6])
2 >>> next(sol_2)
3 1
4 >>> next(sol_2)
5 32
6 >>> next(sol_2)
7 729

```

Neste exemplo a função `pow` aplica-se primeiro ao para (1,4), depois a (2,5) e finalmente a (3,6).

Pensemos agora num problema um pouco diferente: dada uma sequência queremos encontrar uma nova sequência a partir desta depois de filtrar os

elementos que satisfazem uma dada condição. Para tornar a questão mais concreta admitamos que pretendemos eliminar de uma lista de inteiros naturais todos os números que são múltiplos de 3 e/ou de 5. Eis três soluções possíveis.

```

1  def filtro_3_5_1(seq):
2      nova_seq = []
3      conta = 0
4      fim = len(seq) - 1
5      while conta <=fim:
6          elem = seq[conta]
7          if (elem % 3 != 0) and (elem % 5 != 0):
8              nova_seq.append(elem)
9          conta += 1
10     return nova_seq
11
12 def filtro_3_5_2(seq):
13     nova_seq = []
14     for elem in seq:
15         if (elem % 3 != 0) and (elem % 5 != 0):
16             nova_seq.append(elem)
17     return nova_seq
18
19 def filtro_3_5_3(seq):
20     return [ elem for elem in seq if (elem % 3 != 0) and (elem
21             % 5 != 0)]
22
23 def filtro_3_5_4(seq):
24     return [ elem for elem in seq if (elem % 3) and (elem % 5)
25             ]

```

O leitor reconhecerá um padrão semelhante ao caso anterior, mas onde agora é necessário usar um teste para efectuar a filtragem. A última versão ilustra como os valores de verdade são identificados em Python . Optar por esta forma simplificada é uma questão de preferência, embora na nossa opinião seja menos legível. Também nesta situação Python oferece uma função que nos resolve o problema, chamada `filter`.

```

1  def filtro(func,seq):
2      return list(filter(func,seq))
3
4  def filtro_3_5(x):
5      return (x % 3 != 0) and (x % 5 != 0)

```

A sintaxe é muito simples:

```
filter(<func>, <iteravel>)
```

Ao aplicar o filtro ao iterável obtemos apenas os elementos cujo teste `func` é verdadeiro. Caso se pretenda o inverso, isto é apenas os elementos que dão falso quando se aplica o teste, ou alteramos a função ou usamos a função `filterfalse` do módulo pré-definido `itertools`. Notar que no exemplo acima tivemos que envolver a chamada de `filter` com `list`. A razão deve-se ao facto de `filter` ser também um iterador. Podemos usar esta ideia para calcular mais uma vez os números primos até um dado limite.

```
1 def primos(n):
2     res = list(range(2, n+1))
3     for i in range(2,n//2+1):
4         res = list(filter(lambda x: (x == i) or (x % i != 0),
5                           res))
6     return res
```

Não se pode dizer, no entanto, que seja uma solução muito eficiente.

Outros Exemplos

Sabemos agora que é possível aplicar uma dada função a uma sequência ou sequências. Alguns casos são mesmo nossos conhecidos há bastante tempo, como a função `zip`.

```

1 >>> list(zip([1,2,3],['a','b','c']))
2 [(1, 'a'), (2, 'b'), (3, 'c')]

```

`zip` é um iterador e constrói tuplos com os elementos da mesma posição de cada uma das sequências dadas como argumentos. Existem outros casos de funções pré-definidas que realizam para nós operações específicas sobre todos os elementos de uma sequência. Vejamos exemplos.

```

1 >>> any([' ',' ',' '])
2 False
3 >>> any([' ','b',' '])
4 True
5 >>> all([' ','b',' '])
6 False
7 >>> all(['a','b','c'])
8 True
9 >>>

```

A listagem ilustra o uso da função `any` que devolve `False` apenas se todos os seus argumentos forem falsos, e o uso da função `all` que devolve `True` apenas se todos os seus argumentos forem verdadeiros. Os exemplos mostram também que a representação de verdadeiro e de falso em Python pode ser feita por diferentes objectos de diferentes tipos. Estas duas funções correspondem aos quantificadores existencial (`any`) e universal (`all`) usados em lógica.

Suponhamos agora que queremos efectuar a soma de uma lista com números arbitrários. Simples, dizemos nós, a partir do momento que temos a função pré-definida `sum`. Claro que se ela não existisse também podíamos escrever um pequeno programa para o fazer.

```

1 >>> def somatorio(lista_num):
2 ...     res = 0
3 ...     for elem in lista_num:
4 ...         res += elem
5 ...     return res
6 ...
7 >>> somatorio([1,2,3,4,5])
8 15

```

```
9 >>> sum([1,2,3,4,5])  
10 15  
11 >>>
```

O interessante neste exemplo é o modo como se processa o cálculo: somamos dois elementos, o resultado é somado ao elemento seguinte, e assim sucessivamente até o cálculo estar concluído. Em Python podemos fazer esta operação ainda de outro modo recorrendo à função `reduce` do módulo `functools`.

```
1 >>> import functools  
2 >>> functools.reduce(lambda x,y: x + y,[1,2,3,4,5])  
3 15  
4 >>>
```

A função que é passada como argumento a `reduce` possui dois argumentos. Pode funcionar também com a definição de um valor inicial.

```
1 >>> functools.reduce(lambda x,y: x + y,[1,2,3,4,5],10)  
2 25  
3 >>> functools.reduce(lambda x,y: x + y,[],5)  
4 5
```

No caso de não ser usado valor inicial e de a sequência ter apenas um valor é esse valor que é devolvido. Se apenas se pudesse usar `reduce` para somar não seria muito interessante. Qualquer operação binária por nós definida pode ser usada. Vimos que o podemos fazer, em casos simples, recorrendo a definições `lambda`. Mas existe um módulo denominado `operator` que exporta um conjunto de operações/operadores a que podemos recorrer. Alguns exemplos.

```
1 >>> import functools  
2 >>> import operator  
3 >>> functools.reduce(operator.add,[1,2,3,4,5])  
4 15  
5 >>> functools.reduce(operator.pow,[1,2,3,4,5])  
6 1  
7 >>> functools.reduce(operator.mul,[1,2,3,4,5])  
8 120  
9 >>> functools.reduce(operator.truediv,[1,2,3,4,5])  
10 0.008333333333333333  
11 >>>
```

Existem para todas as funções definidas neste módulo uma variante com duas linhas antes de outras duas depois como o exemplo abaixo ilustra.

```

1 >>> functools.reduce(operator.__add__,[1,2,3,4,5])
2 15
3 >>> (5).__add__(4)
4 9
5 >>>

```

O segundo exemplo mostra que se pode usar esta variante mesmo **sem** importar o módulo.

Sumário

Neste capítulo introduzimos conceitos complementares relacionados com a linguagem **Python**. Em particular abordamos os conceitos de ambiente, espaço de nomes e espaço de objectos e discutimos a regra de resolução de nomes de variáveis. Tratámos também da questão dos diferentes modos de importar um módulo, e detalhámos os diferentes tipos de argumentos que podem existir numa definição e o modo como se ligam os parâmetros formais e os reais no momento da chamada ou invocação de uma definição. De seguida introduzimos o conceito de iterador e de gerador e de como permitem uma avaliação preguiçosa¹⁶ e de como essa possibilidade permite economia no espaço de memória necessário numa computação. Falámos de funções anónimas, das suas vantagens mas também das suas limitações, e discutimos funções de ordem superior. Não terminámos sem fazer referência a funções de segunda ordem pré-definidas e das operações nos módulos **functools** e **operator**.

Teste os seus conhecimentos

- O que entende por ambiente, espaço de nomes, espaço de objectos
- O que entende por alcance de uma variável
- O que são declarações **global**, **nonlocal**
- Em que consiste a Regra **LEGB**
- O que entende por fecho e para que serve
- O que são e para que servem os decoradores
- De que modos se podem importar módulos e o que os distingue

¹⁶Do inglês *lazy evaluation*.

- Que diferença entre usar e executar um módulo
- Que código permite que um módulo esteja preparado para ser usado ou executado
- O que são parâmetros formais e parâmetros reais
- Como se processa a ligação entre os dois tipos de parâmetros
- Que cuidados se deve ter com parâmetros por defeito
- Que consequências tem usar parâmetros que estão associados a objectos mutáveis ou a objectos imutáveis
- O que são objectos iteráveis e o que são iteradores
- O que entende por funções geradoras
- O que são funções anónimas e em que situações o seu uso é obrigatório
- Que exemplos ficou a conhecer de funções de ordem superior

Exercícios

Exercício 10.1 F

Considere a sessão seguinte no interpretador.

```
1 >>> nome = 'ernesto'
2 >>> def toto():
3     ...     nome = 'costa'
4     ...     return None
5 ...
6 >>> toto()
7 >>> nome
8 ???
9 >>>
```

Diga, justificando o que vai aparecer no lugar dos pontos de interrogação.

Exercício 10.2 F

O que vai aparecer no lugar dos pontos de interrogação na sessão abaixo?
Porquê?

```

1  >>> ultima_resposta = 60
2  >>> def ultima_maquina():
3      ...     global ultima_resposta
4      ...     ultima_resposta = 'Nope!'
5      ...     return ultima_resposta
6      ...
7  >>>
8  >>> ultima_maquina()
9  ???
10 >>> ultima_maquina()
11 ???
12 >>> ultima_resposta
13 ???

```

Exercício 10.3 M

O que vai aparecer no lugar dos pontos de interrogação na sessão abaixo? Porquê?

```

1  >>> def f(t=0):
2      ...     def g(t=0):
3          ...         def h():
4              ...             nonlocal t
5              ...             t += 1
6              ...             return h, lambda:t
7              ...             h, gt = g()
8              ...             return h,gt,lambda:t
9              ...
10 >>> h,gt,ft = f()
11 >>> ft(),g()
12 ??? # <--#
13 >>> h()
14 >>>ft(),gt()
15 ??? # <--#
16 >>>

```

Exercício 10.4 M

Uma conta bancária tem como elemento fundamental o seu saldo e a sua gestão durante os movimentos de conta (depósitos e levantamentos). Pretende-se criar um programa que permita definir diversas contas e o seu saldo inicial. Esse mesmo programa é o responsável pela actualização do

saldo de cada conta criada. A listagem abaixo dá uma ideia do pretendido.

```

1 >>> conta_1 = gere_depositos(0) # cria conta com saldo...0
2 >>> conta_2 = gere_depositos(42) # cria conta com saldo 42
3 >>> print(conta_1(10))
4 Saldo insuficiente
5 >>> print(conta_2(10))
6 32

```

Exercício 10.5 D

Implemente um programa que lhe permita criar contadores. Esses contadores devem poder fazer duas acções: contar e ser reiniciados. Alistagem ilustra o pretendido.

```

1 >>> contador_1 = gere_contador(0)
2 >>> contador_2 = gere_contador(42)
3 >>> print(contador_1('conta'))
4 1
5 >>> print(contador_2('conta'))
6 43
7 >>> print(contador_2('reiniciar'))
8 0
9 >>> print(contador_1('conta'))
10 2
11 >>> print(contador_2('conta'))
12 1
13 >>> print(contador_1('oops'))
14 Acção desconhecida

```

Exercício 10.6 D

Imagine um agente que vai proferindo frases, uma atrás da outra. Use a ideia de fecho e de variáveis não locais para criar um programa que cada vez que é executado (tempo $t+1$) imprime a frase anterior do agente (tempo t).
Nota: Tem que prever que na primeira vez que é executado ainda não foi proferida nenhuma frase.

Exercício 10.7 M

Considere o código seguinte módulo.

```

1 def soma_defs(f,g,x):
2     return f(x) + g(x)
3

```

```

4 def func_1(y):
5     return y**3
6
7 def func_2(z):
8     return z + 5 # <-----
9
10 if __name__ == '__main__':
11     print(soma_defs(func_1, func_2, 4))

```

Descreva o ambiente através de um diagrama no momento da execução da instrução assinalada por uma seta. Qual o resultado final após a execução.

Exercício 10.8 MD

Diga como pode desenvolver um programa de traço de execução para algoritmos recursivos. A listagem mostra o pretendido quando aplicado às funções fibonacci e factorial.

```

1 ('|__', 'fib', 3)
2 ('| |__', 'fib', 2)
3 ('| | |__', 'fib', 1)
4 ('| | | |__', 'return', '1')
5 ('| | | |__', 'fib', 0)
6 ('| | | |__', 'return', '1')
7 ('| | | |__', 'return', '2')
8 ('| | |__', 'fib', 1)
9 ('| | | |__', 'return', '1')
10 ('| | |__', 'return', '3')
11 3
12 ('|__', 'fact', 4)
13 ('| |__', 'fact', 3)
14 ('| | |__', 'fact', 2)
15 ('| | | |__', 'fact', 1)
16 ('| | | | |__', 'fact', 0)
17 ('| | | | | |__', 'return', '1')
18 ('| | | | | |__', 'return', '1')
19 ('| | | | |__', 'return', '2')
20 ('| | | | |__', 'return', '6')
21 ('| | | |__', 'return', '24')
22 24

```

Exercício 10.9 F

Use a ideia que apresentámos para definir o iterador `range` para implementar uma versão para o caso de números reais.

Exercício 10.10 F

Implemente uma função geradora para os números naturais pares.

Exercício 10.11 M

Implemente uma função geradora genérica que permita fornecer um a um os elementos de $f(x)$ com

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

Os valores de x também eles são gerados de acordo com uma dada função. Deste modo a função geradora terá dois argumentos funcionais, um para a função f , o outro para a função que vai gerar os sucessivos valores sobre os quais se aplica a função f .

Exercício 10.12 M

Implemente uma função geradora que lhe permita gerar as letras do alfabeto. Deve ser possível indicar a letra inicial. A listagem ilustra o pretendido.

```

1 >>> gera_1 = letras('c')
2 >>> next(gera_1)
3 c
4 >>>next(gera_1)
5 d
6 >>>
```

Exercício 10.13 M

A função exponencial e^x pode ser definida pela correspondente série de Taylor:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Desenvolva um gerador que cada vez que é chamado devolve um valor mais correcto para o valos de e^x . Use o método para obter o valor aproximado de e .

Exercício 10.14 M

Maximizando o uso dos conceitos deste capítulo, implemente uma função que lhe permita determinar se uma lista de objectos está ou não ordenada.

Pode apresentar uma solução genérica que permita escolher a função de comparação.

Exercício 10.15 M

Usando funções de ordem superior implemente um filtro que permita retirar a uma sequência todos os elementos que não satisfazem um dado critério. Procure que a sua solução seja o mais genérica possível.

Exercício 10.16 F

Admita que não existe a função pré-definida `min`. Implemente-a socorrendo-se da função `reduce` e de funções anónimas.

Exercício 10.17 F

Implemente a função factorial, não recursiva, recorrendo à função `reduce`.

Exercício 10.18 M

Usando funções de ordem superior implemente um filtro que permita retirar a uma sequência todos os elementos que não satisfazem um dado critério. Procure que a sua solução seja o mais genérica possível.

Exercício 10.19 D

Pretende-se transformar uma função com vários argumentos numa cadeia de funções com apenas um argumento. Implemente um programa que permite fazer isso, para o caso de qualquer função com dois argumentos.

```
1 if __name__ == '__main__':
2     my_pow = cadeia_f(pow)
3     print(my_pow(2)(3)) # --> 8
```

Exercício 10.20 MD

Suponha que não existe pré-definido em Python o tipo dicionário. Recorrendo ao que aprendeu sobre funções anónimas e funções de segunda ordem, diga como podia definir um construtor para um tipo de dados dicionário.

Bibliografia

- [1] John E. Grayson. *Python and Tkinter Programming*. Manning Publications Co., 2000.
- [2] Kim Hamilton and Russell Miles. *Learning UML 2.0*. O'Reilly, 2006.
- [3] Mark Lutz. *Learning Python (4th edition)*. O'Reilly, 2009.
- [4] M. Stefik, D. Bobrow, and D. Winter. Object-oriented programming: themes and variations, 1986.
- [5] Peter Wegner. Dimensions of object-based language design. In *OOPSLA '87 Conference proceedings on Object-oriented programming systems, languages and applications*, 1987.