

Programação em Python: Introdução à Programação Utilizando Múltiplos Paradigmas

João Pavão Martins

Departamento de Engenharia Informática
Instituto Superior Técnico
Universidade Técnica de Lisboa

Copyright ©2013 João Pavão Martins

Este livro não pode ser reproduzido, no todo ou em parte, sob qualquer forma ou meio, electrónico ou mecânico, incluindo fotocópia, gravação, ou qualquer sistema de registo de informação, sem autorização prévia e escrita do autor:

João Pavão Martins
Departamento de Engenharia Informática
Instituto Superior Técnico
Av. Rovisco Pais
1096 Lisboa CODEX
Portugal
jpmartins@siscog.pt

Índice

1 Computadores, algoritmos e programas	1
1.1 Características de um computador	4
1.2 Algoritmos	6
1.2.1 Exemplos informais de algoritmos	6
1.2.2 Características de um algoritmo	9
1.3 Programas e algoritmos	11
1.3.1 Linguagens de programação	12
1.3.2 Exemplo de um programa	14
1.4 Sintaxe e semântica	17
1.4.1 Sintaxe	17
1.4.2 Semântica	25
1.4.3 Tipos de erros num programa	26
1.5 Notas finais	28
1.6 Exercícios	29
2 Elementos básicos de programação	33
2.1 Expressões	35
2.1.1 Constantes	35
2.1.2 Expressões compostas	37
2.2 Tipos elementares de informação	39
2.2.1 O tipo inteiro	41
2.2.2 O tipo real	41
2.2.3 O tipo lógico	45
2.3 Nomes e atribuição	46
2.4 Predicados e condições	53
2.5 Comunicação com o exterior	54
2.5.1 Leitura de dados	54

2.5.2	Escrita de dados	57
2.6	Programas, instruções e sequenciação	58
2.7	Selecção	60
2.8	Repetição	64
2.9	Notas finais	68
2.10	Exercícios	69
3	Funções	73
3.1	Definição de funções em Python	75
3.2	Aplicação de funções em Python	77
3.3	Abstracção procedural	80
3.4	Exemplos	82
3.4.1	Nota final de uma disciplina	82
3.4.2	Potência	85
3.4.3	Factorial	86
3.4.4	Máximo divisor comum	86
3.4.5	Raiz quadrada	88
3.4.6	Seno	94
3.5	Estruturação de funções	96
3.6	Módulos	109
3.7	Notas finais	113
3.8	Exercícios	113
4	Tuplos e ciclos contados	115
4.1	Tuplos	115
4.2	Ciclos contados	123
4.3	Cadeias de caracteres revisitadas	127
4.4	Notas finais	137
4.5	Exercícios	138
5	Listas	141
5.1	Listas em Python	141
5.2	Métodos de passagem de parâmetros	145
5.2.1	Passagem por valor	146
5.2.2	Passagem por referência	148
5.3	O Crivo de Eratóstenes	149
5.4	Algoritmos de procura	152

5.4.1	Procura sequencial	153
5.4.2	Procura binária	154
5.5	Algoritmos de ordenação	155
5.5.1	Ordenação por borbulhamento	158
5.5.2	Ordenação Shell	159
5.5.3	Ordenação por selecção	160
5.6	Exemplo	160
5.7	Considerações sobre eficiência	162
5.7.1	A notação do Omaiúsculo	166
5.8	Notas finais	168
5.9	Exercícios	170
6	Funções revisitadas	173
6.1	Funções recursivas	173
6.2	Funções de ordem superior	180
6.2.1	Funções como parâmetros	181
6.2.2	Funções como valor de funções	192
6.3	Programação funcional	196
6.4	Notas finais	199
6.5	Exercícios	199
7	Recursão e iteração	205
7.1	Recursão linear	206
7.2	Iteração linear	209
7.3	Recursão em processos e em funções	214
7.4	Recursão em árvore	215
7.4.1	Os números de Fibonacci	215
7.4.2	A torre de Hanói	220
7.5	Considerações sobre eficiência	224
7.6	Notas finais	226
7.7	Exercícios	226
8	Ficheiros	229
8.1	O tipo ficheiro	230
8.2	Leitura de ficheiros	231
8.3	Escrita em ficheiros	236
8.4	Notas finais	239

8.5 Exercícios	239
9 Dicionários	243
9.1 O tipo dicionário	243
9.2 Frequência de letras num texto	246
9.3 Dicionários de dicionários	250
9.4 Caminhos mais curtos em grafos	254
9.5 Notas finais	260
9.6 Exercícios	260
10 Abstracção de dados	263
10.1 A abstracção em programação	264
10.2 Motivação: números complexos	265
10.3 Tipos abstractos de informação	270
10.3.1 Identificação das operações básicas	271
10.3.2 Axiomatização	275
10.3.3 Escolha da representação	276
10.3.4 Realização das operações básicas	276
10.4 Barreiras de abstracção	278
10.5 Objectos	282
10.6 Notas finais	288
10.7 Exercícios	289
11 Programação com objectos	293
11.1 O tipo conta bancária	294
11.2 Classes, subclasses e herança	297
11.3 Objectos em Python	306
11.4 Polimorfismo	307
11.5 Notas finais	312
11.6 Exercícios	312
12 O desenvolvimento de programas	315
12.1 A análise do problema	318
12.2 O desenvolvimento da solução	319
12.3 A programação da solução	321
12.3.1 A depuração	322
12.3.2 A finalização da documentação	325
12.4 A fase de testes	327

12.5 A manutenção	329
12.6 Notas finais	330
13 Estruturas lineares	333
13.1 Pilhas	333
13.1.1 Operações básicas para pilhas	334
13.1.2 Axiomatização	339
13.1.3 Representação de pilhas	339
13.1.4 Realização das operações básicas	339
13.2 Balanceamento de parêntesis	345
13.3 Expressões em notação pós-fixa	347
13.4 Filas	351
13.4.1 Operações básicas para filas	352
13.4.2 Axiomatização	354
13.4.3 Representação de filas	355
13.4.4 A classe fila	355
13.5 Simulação de um supermercado	358
13.6 Representação gráfica	369
13.7 Notas finais	386
13.8 Exercícios	387
14 Árvores	389
14.1 Operações básicas para árvores	391
14.2 Axiomatização	394
14.3 Representação de árvores	394
14.3.1 Representação para o uso recorrendo a funções	394
14.3.2 Representação para o uso recorrendo a objectos	395
14.4 Realização das operações básicas	396
14.4.1 Árvores recorrendo a funções	397
14.4.2 A classe árvore	399
14.5 Ordenação por árvore	402
14.6 Notas finais	406
14.7 Exercícios	406
15 Ponteiros	407
15.1 A noção de ponteiro	408
15.2 Listas ligadas	410

15.3 A gestão de memória	421
15.3.1 A gestão manual do amontoado	422
15.3.2 A recolha de lixo	423
15.4 Notas finais	424
15.5 Exercícios	424
16 Epílogo	429
16.1 Programas	430
16.1.1 Algoritmos	431
16.1.2 Linguagens	433
16.1.3 Construção de abstracções	436
16.2 Programação	438
16.2.1 Arquitectura de programas	439
16.2.2 Paradigmas de programação	440
16.2.3 Técnicas usadas em programação	445
16.2.4 Sistemas operativos	449
16.3 Notas Finais	451
A Soluções de exercícios seleccionados	453
A.1 Exercícios do Capítulo 1	453
A.2 Exercícios do Capítulo 2	454
A.3 Exercícios do Capítulo 3	455
A.4 Exercícios do Capítulo 4	457
A.5 Exercícios do Capítulo 5	459
A.6 Exercícios do Capítulo 6	460
A.7 Exercícios do Capítulo 7	462
A.8 Exercícios do Capítulo 8	464
A.9 Exercícios do Capítulo 9	466
A.10 Exercícios do Capítulo 10	467
A.11 Exercícios do Capítulo 11	470

Prefácio

Este livro corresponde à matéria ensinada na disciplina semestral de Fundamentos da Programação da Licenciatura de Engenharia Informática e de Computadores do Instituto Superior Técnico. A disciplina situa-se no primeiro semestre do primeiro ano e não pressupõe conhecimentos prévios em programação. A matéria deste livro foi influenciada pelas diversas propostas internacionais relativas ao conteúdo de uma disciplina de introdução à programação a nível universitário [Koffmann et al., 1984], [Koffmann et al., 1985], [Turner, 1991], [ACM, 2000], [ACM, 2008], [ACM, 2012] tendo sido também influenciada por um dos melhores livros existentes sobre programação, [Abelson et al., 1996].

O domínio da programação transcende o conhecimento de técnicas de programação, necessitando de uma compreensão profunda dos conceitos e da natureza da computação. Com este livro fornecemos uma abordagem disciplinada à actividade de programação e, simultaneamente, transmitimos conceitos genéricos relacionados com linguagens de programação. A escolha da linguagem de programação a utilizar numa disciplina introdutória é sempre um aspecto muito controverso. Na década de 1990, a linguagem utilizada pelo autor para ensinar introdução à programação foi o Pascal, na primeira década do século XXI, foi utilizada a linguagem Scheme. No ano lectivo de 2012/13 foi decidida a adopção da linguagem Python para leccionar a disciplina de Fundamentos da Programação. A nossa escolha foi influenciada por diversos factores:

- Apresentação de uma sintaxe mínima, facilitando a aprendizagem.
- Possibilidade de utilização de diversos paradigmas de programação, incluindo a programação imperativa, a programação funcional e a programação por objectos.
- Existência de processadores da linguagem, ao nível do domínio público, para

os principais sistemas operativos, Macintosh, Windows e Unix.

Deve ser desde já clarificado que o objectivo deste livro não é o de ensinar Python (e consequentemente algumas das suas características não são abordadas), mas sim o de utilizar o Python para ensinar a programar em qualquer linguagem.

Existem vários aspectos que fazem com que este livro se distinga de outros livros que abordam a introdução à programação:

1. Apresenta a actividade de programação como uma construção de abstracções, tanto de abstracções de procedimentos, como de abstracções de dados.
2. Apresenta uma introdução geral à actividade de programação. O livro utiliza o Python como uma ferramenta para exprimir os conceitos introduzidos, sendo o seu objectivo a apresentação de conceitos essenciais de programação, tais como o desenvolvimento de algoritmos utilizando o método do topo para a base, a utilização de estruturas de informação adequadas, a abstracção procedural e a abstracção de dados, estratégias para teste e depuração de programas, a documentação correcta e o anonimato da representação.
3. Enfatiza a distinção entre a sintaxe e a semântica. Para dominar a tarefa de programação não basta saber falar com a linguagem (escrever programas), é também necessário saber falar sobre a linguagem (utilizando português ou outra linguagem como metalinguagem). Este aspecto é abordado ao longo de todo o livro.
4. A metodologia para o desenvolvimento de tipos abstractos de informação é explicada em termos claros, e exemplos de tipos abstractos de informação são discutidos pormenorizadamente.

O Python é uma linguagem de programação inventada pelo cientista holandês Guido van Rossum no final da década de 1980. A sua primeira versão surgiu em 1989, tendo a versão utilizada neste livro, o Python 3, sido lançada em 2008. Em 2007 e em 2010, o Python foi considerada a linguagem de programação do ano¹, sendo actualmente a 8^a linguagem mais popular, depois de C, Java, Objective-C, C++, C#, PHP e Visual BASIC. Entre as grandes organizações

¹www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

que utilizam o Python incluem-se² a Google, a Yahoo!, o YouTube, o Facebook, a Walt Disney Feature Animation, a Nokia, o CERN e a NASA.

Agradecimentos

Gostava de agradecer a todas as pessoas que contribuíram para a forma final deste livro através de críticas e sugestões. Entre estas pessoas realço as seguintes (por ordem alfabética do último nome): Fausto Almeida, Vasco Almeida, Luís Manuel Tavares Jordão Alves, João Romeiras Amado, Hugo Assunção, Leonor Bandeira, Ana Cardoso Cachopo, José Pedro Martins Cândido, Francisco Maria Calisto, João Carrapiço, Carlos Carvalho, João Gonçalo Ceia, Maria Teresa Coelho, João Colaço, Margarida Correia, Nuno Miguens Diogo, Adriana Domingos, Mariana Gaspar Fernandes, Alexandre Ferreira, Hugo Ferreira, Bernardo Pascoal Figueiredo, Francisco Castro Henriques, João Martins, Pedro Amaro de Matos, Francisco Melo, Diogo Filipe Panaca Mendes, Miguel Neves Pasadinhas, Pedro Pinela, Nuno Pires, Yuliya Plotka, Tiago Mendes de Almeida Rechau, João Filipe Coito Ribeiro, Vicente Rocha, Inês Santos, Márcio Santos, Anisa Shahidian, Daniel Sil, Daniel Tanque, André Tomé, João Vieira. Agradeço em particular à Prof. Maria dos Remédios Cravo, a co-autora do livro de Scheme [Martins e Cravo, 2007] pelos seus comentários e contribuição indirecta através da re-utilização de certos trechos da obra anterior. Contudo, todos os erros e imperfeições que este livro indubitavelmente contém são da exclusiva responsabilidade do autor.

²De www.infernodevelopment.com/python-becoming-most-popular-programming-language e <http://wiki.python.org/moin/OrganizationsUsingPython>.

x

ÍNDICE

Capítulo 1

Computadores, algoritmos e programas

“Take some more tea,” the March Hare said to Alice, very earnestly.

*“I’ve had nothing yet,” Alice replied in an offended tone,
“so I can’t take more.”*

“You mean you can’t take less,” said the Hatter: “it’s very easy to take more than nothing.”

Lewis Carroll, *Alice’s Adventures in Wonderland*

Uma das características de um engenheiro é a capacidade para resolver problemas técnicos. A resolução deste tipo de problemas envolve uma combinação de ciência e de arte. Por *ciência* entende-se um conhecimento dos princípios matemáticos, físicos e dos aspectos técnicos que têm de ser bem compreendidos, para que sejam aplicados correctamente. Por *arte* entende-se a avaliação correcta, a experiência, o bom senso e o conhecimento que permitem representar um problema do mundo real por um modelo ao qual o conhecimento técnico pode ser aplicado para produzir uma solução.

De um modo geral, qualquer problema de engenharia é resolvido recorrendo a uma sequência de fases: a *compreensão do problema* é a fase que corresponde a perceber e a identificar de um modo preciso o problema que tem de ser resolvido; após a compreensão do problema, entra-se na fase correspondente à *especificação do problema*, na qual o problema é claramente descrito e documentado, de modo

2 CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

a remover dúvidas e imprecisões; no *desenvolvimento da solução* (ou *modelação da solução*) utiliza-se a especificação do problema para produzir um esboço da solução do problema, identificando métodos apropriados de resolução de problemas e as suposições necessárias; durante o desenvolvimento da solução, o esboço da solução é progressivamente pormenorizado até se atingir um nível de especificação que seja adequado para a sua realização; na *concretização da solução*, as especificações desenvolvidas são concretizadas (seja num objecto físico, por exemplo, uma ponte ou um processador, seja num objecto imaterial, por exemplo, um programa de computador); finalmente, na fase de *verificações e testes*, o resultado produzido é validado, verificado e testado.

A Engenharia Informática difere das engenharias tradicionais, no sentido em que trabalha com entidades imateriais. Ao passo que as engenharias tradicionais lidam com forças físicas, directamente mensuráveis (por exemplo, a gravidade, os campos eléctricos e magnéticos) e com objectos materiais que interagem com essas forças (por exemplo, rodas dentadas, vigas, circuitos), a Engenharia Informática lida com entidades intangíveis que apenas podem ser observadas indirectamente através dos efeitos que produzem.

A *Engenharia Informática* tem como finalidade a concepção e realização de abstracções ou modelos de entidades abstractas que, quando aplicadas por um computador, fazem com que este apresente um comportamento que corresponde à solução de um dado problema.

Sob a perspectiva da Informática que apresentamos neste livro, um computador é uma máquina cuja função é manipular informação. Por *informação* entende-se qualquer coisa que pode ser transmitida ou registada e que tem um significado associado à sua representação simbólica. A informação pode ser transmitida de pessoa para pessoa, pode ser extraída directamente da natureza através de observação e de medida, pode ser adquirida através de livros, de filmes, da televisão, etc. Uma das características que distinguem o computador de outras máquinas que lidam com informação é o facto de este poder manipular a informação, para além de a armazenar e transmitir. A manipulação da informação feita por um computador segue uma sequência de instruções a que se chama um *programa*. Apesar de sabermos que um computador é uma máquina complexa, constituída por componentes electrónicos, nem os seus componentes nem a interligação entre eles serão aqui estudados. Para a finalidade que nos propomos atingir, o ensino da programação, podemos abstrair de toda a

constituição física de um computador, considerando-o uma “caixa electrónica”, vulgarmente designada pela palavra inglesa “hardware”, que tem a capacidade de compreender e de executar programas.

A *Informática* é o ramo da ciência que se dedica ao estudo dos computadores e dos processos com eles relacionados: como se desenvolve um computador, como se especifica o trabalho a ser realizado por um computador, de que forma se pode tornar mais fácil de utilizar, como se definem as suas limitações e, principalmente, como aumentar as suas capacidades e o seu domínio de aplicação.

Um dos objectivos da Informática corresponde ao estudo e desenvolvimento de entidades abstractas geradas durante a execução de programas – os processos computacionais. Um *processo computacional* é um ente imaterial que evolui ao longo do tempo, executando acções que levam à solução de um problema. Um processo computacional pode afectar objectos existentes no mundo real (por exemplo, guiar a aterragem de um avião, distribuir dinheiro em caixas multibanco, comprar e vender acções na bolsa), pode responder a perguntas (por exemplo, indicar quais as páginas da internet que fazem referência a um dado termo), entre muitos outros aspectos.

A evolução de um processo computacional é ditada por uma sequência de instruções a que se chama *programa*, e a actividade de desenvolver programas é chamada *programação*. A programação é uma actividade intelectual fascinante, que não é difícil, mas que requer muita disciplina. O principal objectivo deste livro é fornecer uma introdução à programação disciplinada, ensinando os princípios e os conceitos subjacentes, os passos envolvidos no desenvolvimento de um programa e o modo de desenvolver programas bem estruturados, eficientes e sem erros.

A programação utiliza muitas actividades e técnicas que são comuns às utilizadas em projectos nos vários ramos da engenharia: a compreensão de um problema; a separação entre a informação essencial ao problema e a informação acessória; a criação de especificações pormenorizadas para o resolver; a realização destas especificações; a verificação e os testes.

Neste primeiro capítulo definimos as principais características de um computador e introduzimos um conceito essencial para a informática, o conceito de algoritmo.

1.1 Características de um computador

Um *computador* é uma máquina cuja função é manipular símbolos. Embora os computadores difiram em tamanho, aparência e custo, eles partilham quatro características fundamentais: são automáticos, universais, electrónicos e digitais.

Um computador diz-se *automático* no sentido em que, uma vez alimentado com a informação necessária, trabalha por si só, sem a intervenção humana. Não pretendemos, com isto, dizer que o computador comece a trabalhar por si só (necessita, para isso, da intervenção humana), mas que o computador procura por si só a solução dos problemas. Ou seja, o computador é automático no sentido em que, uma vez o trabalho começado, ele será levado até ao final sem a intervenção humana. Para isso, o computador recebe um *programa*, um conjunto de instruções quanto ao modo de resolver o problema. As instruções do programa são escritas numa notação compreendida pelo computador (uma *linguagem de programação*), e especificam *exactamente* como o trabalho deve ser executado. Enquanto o trabalho está a ser executado, o programa está armazenado dentro do computador e as suas instruções estão a ser seguidas.

Um computador diz-se *universal*, porque pode efectuar qualquer tarefa cuja solução possa ser expressa através de um programa. Ao executar um dado programa, um computador pode ser considerado uma máquina orientada para um fim particular. Por exemplo, ao executar um programa para o tratamento de texto, um computador pode ser considerado como uma máquina para produzir cartas ou texto; ao executar um programa correspondente a um jogo, o computador pode ser considerado como uma máquina para jogar. A palavra “universal” provém do facto de o computador poder executar qualquer programa, resolvendo problemas em diferentes áreas de aplicação. Ao resolver um problema, o computador manipula os símbolos que representam a informação pertinente para esse problema, sem lhes atribuir qualquer significado. Devemos, no entanto, salientar que um computador não pode resolver qualquer tipo de problema. A classe dos problemas que podem ser resolvidos através de um computador foi estudada por matemáticos antes da construção dos primeiros computadores. Durante a década de 1930, matemáticos como Alonzo Church (1903–1995), Kurt Gödel (1906–1978), Stephen C. Kleene (1909–1994), Emil Leon Post (1897–1954) e Alan Turing (1912–1954) tentaram definir matematicamente a classe das funções que podiam ser calculadas mecanicamente. Embora os métodos utilizados por estes matemáticos fossem muito diferentes, todos os

formalismos desenvolvidos são equivalentes, no sentido em que todos definem a mesma classe de funções, as funções recursivas parciais. Pensa-se, hoje em dia, que as funções recursivas parciais são exactamente as funções que podem ser calculadas através de um computador. Este facto é expresso através da *tese de Church-Turing*¹.

De acordo com a tese de Church-Turing, qualquer computação pode ser baseada num pequeno número de operações elementares. Nos nossos programas, estas operações correspondem fundamentalmente às seguintes:

1. *Operações de entrada de dados*, as quais obtêm valores do exterior do programa;
2. *Operações de saída de dados*, as quais mostram valores existentes no programa;
3. *Operações matemáticas*, as quais efectuam cálculos sobre os dados existentes no programa;
4. *Execução condicional*, a qual corresponde ao teste de certas condições e à execução de instruções, ou não, dependendo do resultado do teste;
5. *Repetição*, a qual corresponde à execução repetitiva de certas instruções.

A tarefa de programação corresponde a dividir um problema grande e complexo, em vários problemas, cada vez menores e menos complexos, até que esses problemas sejam suficientemente simples para poderem ser expressos em termos de operações elementares.

Um computador é *electrónico*. A palavra “electrónico” refere-se aos componentes da máquina, componentes esses que são responsáveis pela grande velocidade das operações efectuadas por um computador.

Um computador é também *digital*. Um computador efectua operações sobre informação que é codificada recorrendo a duas grandezas discretas (tipicamente referidas como sendo 0 e 1) e não sobre grandezas que variam de um modo contínuo. Por exemplo, num computador o símbolo “J”, poderá ser representado por 1001010.

¹Uma discussão sobre a tese de Church-Turing e sobre as funções recursivas parciais está para além da matéria deste livro. Este assunto pode ser consultado em [Brainerd e Landweber, 1974], [Hennie, 1977] ou [Kleene, 1975].

1.2 Algoritmos

Ao apresentarmos as características de um computador, dissemos que durante o seu funcionamento ele segue um programa, um conjunto de instruções bem definidas que especificam exactamente o que tem que ser feito. Este conjunto de instruções é caracterizado matematicamente como um algoritmo². Os algoritmos foram estudados e utilizados muito antes do aparecimento dos computadores modernos. Um programa corresponde a um algoritmo escrito numa linguagem que é entendida pelo computador, chamada uma linguagem de programação.

Um *algoritmo* é uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita.

Antes de continuar, vamos analisar a definição de algoritmo que acabámos de apresentar. Em primeiro lugar, um algoritmo consiste numa sequência finita de instruções. Isto quer dizer que existe uma ordem pela qual as instruções aparecem no algoritmo, e que estas instruções são em número finito. Em segundo lugar, as instruções de um algoritmo são bem definidas e não ambíguas, ou seja, o significado de cada uma das instruções é claro, não havendo lugar para múltiplas interpretações do significado de uma instrução. Em terceiro lugar, cada uma das instruções pode ser executada mecanicamente, isto quer dizer que a execução das instruções não requer imaginação por parte do executante. Finalmente, as instruções devem ser executadas num período de tempo finito e com uma quantidade de esforço finita, o que significa que a execução de cada uma das instruções termina.

Um algoritmo está sempre associado com um dado objectivo, ou seja, com a solução de um dado problema. A execução das instruções do algoritmo garante que o seu objectivo é atingido.

1.2.1 Exemplos informais de algoritmos

A descrição de sequências de acções para atingir objectivos tem um papel fundamental na nossa vida quotidiana e está relacionada com a nossa facilidade

²A palavra “algoritmo” provém de uma variação fonética da pronúncia do último nome do matemático persa Abu Ja’far Mohammed ibu-Musa al-Khowarizmi (c. 780–c. 850), que desenvolveu um conjunto de regras para efectuar operações aritméticas com números decimais. Al-Khowarizmi foi ainda o criador do termo “Álgebra” (ver [Boyer, 1974], páginas 166–167).

de comunicar. Estamos constantemente a transmitir ou a seguir sequências de instruções, por exemplo, para preencher impressos, para operar máquinas, para nos deslocarmos para certo local, para montar objectos, etc.

Começamos por examinar algumas sequências de instruções utilizadas na nossa vida quotidiana. Consideremos, em primeiro lugar, a receita de “Rebuçados de ovos”³:

REBUÇADOS DE OVOS

500 g de açúcar;
2 colheres de sopa de amêndoas peladas e raladas;
5 gemas de ovos;
250 g de açúcar para a cobertura;
e farinha.

Leva-se o açúcar ao lume com um copo de água e deixa-se ferver até fazer ponto de pérola. Junta-se a amêndoas e deixa-se ferver um pouco. Retira-se do calor e adicionam-se as gemas. Leva-se o preparado novamente ao lume e deixa-se ferver até se ver o fundo do tacho. Deixa-se arrefecer completamente. Em seguida, com a ajuda de um pouco de farinha, molda-se a massa de ovos em bolas. Leva-se o restante açúcar ao lume com 1 dl de água e deixa-se ferver até fazer ponto de rebuçado. Passam-se as bolas de ovos por este açúcar e põem-se a secar sobre uma pedra untada, após o que se embrulham em papel celofane de várias cores.

Esta receita é constituída por duas partes distintas: (1) uma descrição dos objectos a manipular; (2) uma descrição das acções que devem ser executadas sobre esses objectos. A segunda parte da receita é uma sequência finita de instruções bem definidas (para uma pessoa que saiba de culinária e portanto entenda o significado de expressões como “ponto de pérola”, “ponto de rebuçado”, etc., todas as instruções desta receita são perfeitamente definidas), cada uma das quais pode ser executada mecanicamente (isto é, sem requerer imaginação por parte do executante), num período de tempo finito e com uma quantidade de esforço finita. Ou seja, a segunda parte desta receita é um exemplo informal de um algoritmo.

Consideremos as instruções para montar um papagaio voador, as quais estão associadas ao diagrama representado na Figura 1.1⁴.

³De [Modesto, 1982], página 134. Reproduzida com autorização da Editorial Verbo.

⁴Adaptado das instruções para montar um papagaio voador oferecido pela Telecom Portu-

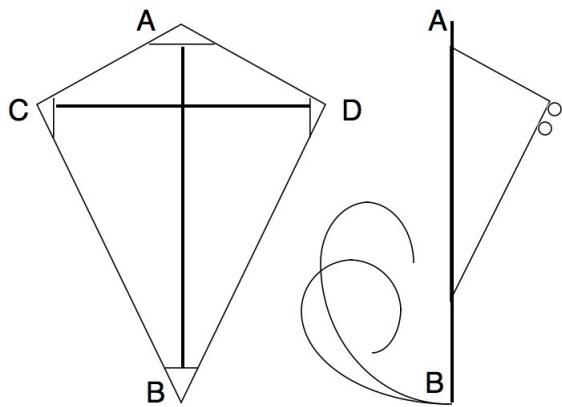


Figura 1.1: Diagrama para montar o papagaio voador.

PAPAGAIO VOADOR

1. A haste de madeira central já se encontra colocada com as pontas metidas nas bolsas A. e B.
2. Dobre ligeiramente a haste transversal e introduza as suas extremidades nas bolsas C. e D.
3. Prenda as pontas das fitas da cauda à haste central no ponto B.
4. Prenda com um nó a ponta do fio numa das argolas da aba do papagaio. Se o vento soprar forte deverá prender na argola inferior. Se o vento soprar fraco deverá prender na argola superior.

As instruções para montar o papagaio voador são constituídas por uma descrição implícita dos objectos a manipular (mostrados na Figura 1.1) e por uma sequência de passos a seguir. Tal como anteriormente, estas instruções podem ser descritas como um algoritmo informal.

Suponhamos que desejamos deslocar-nos do Instituto Superior Técnico na Avenida Rovisco Pais (campus da Alameda) para o campus do Tagus Parque (na Av. Aníbal Cavaco Silva em Oeiras). Recorrendo ao Google Maps, obtemos a descrição apresentada na Figura 1.2. Nesta figura, para além de um mapa ilustrativo, aparecem no lado esquerdo uma sequência detalhada de instruções gal.

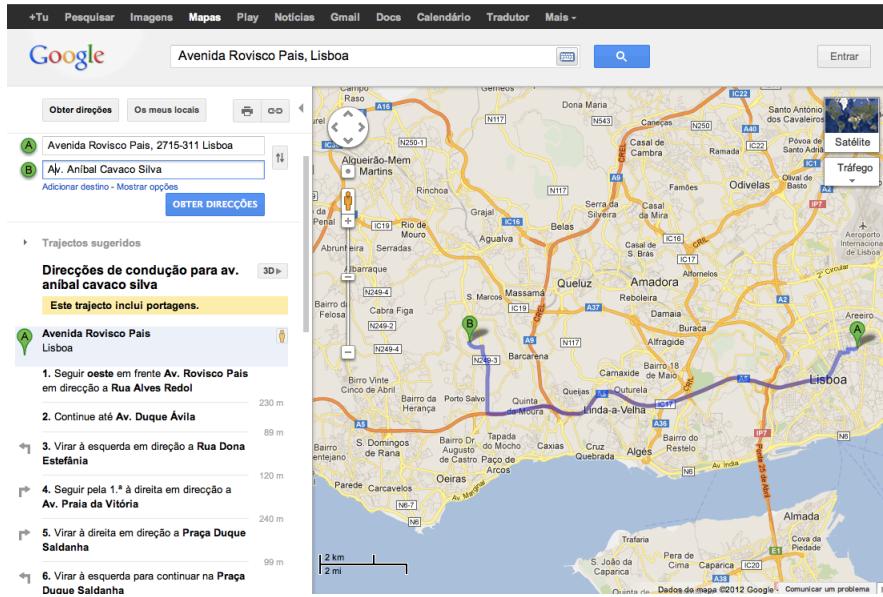


Figura 1.2: Instruções para ir do Campus da Alameda ao do Tagus Parque.

do percurso a seguir para a deslocação pretendida. Novamente, estas instruções podem ser consideradas como um algoritmo informal.

1.2.2 Características de um algoritmo

Existem inúmeros exemplos de algoritmos que utilizamos na nossa vida quotidiana (instruções para fazer uma chamada telefónica, para nos deslocarmos para um certo local, etc.). Estes exemplos mostram que usamos algoritmos desde sempre: eles correspondem a sequências finitas de instruções que devem ser seguidas de modo a atingir um determinado objectivo.

A sequência de passos de um algoritmo deve ser executada por um agente, o qual pode ser humano, mecânico, electrónico, ou qualquer outra coisa. Cada algoritmo está associado a um agente (ou a uma classe de agentes) que deve executar as suas instruções. Aquilo que representa um algoritmo para um agente pode não o ser para outro agente. Por exemplo, as instruções da receita dos rebuçados de ovos são um algoritmo para quem sabe de culinária e não o são para quem não sabe.

Embora um algoritmo não seja mais do que uma descrição da sequência de passos a seguir para atingir um determinado objectivo, nem todas as sequências de passos para atingir um dado objectivo podem ser consideradas um algoritmo, pois um algoritmo deve possuir três características, ser rigoroso, ser eficaz e ter a garantia de terminar.

1. *Um algoritmo é rigoroso.* Cada instrução do algoritmo deve especificar exacta e rigorosamente o que deve ser feito, não havendo lugar para ambiguidade. O facto de um algoritmo poder ser executado mecanicamente obriga a que cada uma das suas instruções tenha uma e só uma interpretação. Por exemplo, a instrução contida na receita dos rebuçados de ovos “leva-se o açúcar ao lume com um copo de água” pode ter várias interpretações. Uma pessoa completamente ignorante de processos culinários pode ser levada a colocar um copo de água (objecto de vidro) dentro de uma panela (ou sobre o lume, interpretando a frase à letra) juntamente com o açúcar.

Para evitar a ambiguidade inherente à linguagem utilizada pelos seres humanos (chamada linguagem natural, de que o português é um exemplo) criaram-se novas linguagens (chamadas linguagens artificiais) para exprimir os algoritmos de um modo rigoroso. Como exemplos de linguagens artificiais, já conhecemos a notação matemática, a qual permite escrever frases de um modo compacto e sem ambiguidade, por exemplo, $\forall x \exists y : y > x$, e a notação química, que permite descrever compostos e reacções químicas de um modo compacto e não ambíguo, por exemplo, $MgO + H_2 \rightarrow Mg + H_2O$. A linguagem Python, discutida neste livro, é mais um exemplo de uma linguagem artificial.

2. *Um algoritmo é eficaz.* Cada instrução do algoritmo deve ser suficientemente básica e bem compreendida de modo a poder ser executada num intervalo de tempo finito, com uma quantidade de esforço finita. Para ilustrar este aspecto, suponhamos que estávamos a consultar as instruções que apareciam na embalagem do adubo “Crescimento Gigantesco”, as quais incluíam a seguinte frase: “Se a temperatura máxima do mês de Abril for superior a 23°, misture o conteúdo de duas embalagens em 5 litros de água, caso contrário, misture apenas o conteúdo de uma embalagem”. Uma vez que não é difícil determinar qual a temperatura máxima do mês de Abril, podemos decidir se deveremos utilizar o conteúdo de duas embalagens ou

apenas o conteúdo de uma. Contudo, se o texto fosse: “Se a temperatura máxima do mês de Abril do ano de 1143 for superior a 23°, misture o conteúdo de duas embalagens em 5 litros de água, caso contrário, misture apenas o conteúdo de uma embalagem”, não seríamos capazes de determinar qual a temperatura máxima do mês de Abril de 1143 e, consequentemente, não seríamos capazes de executar esta instrução. Uma instrução como a segunda que acabamos de descrever não pode fazer parte de um algoritmo, pois não pode ser executada com uma quantidade de esforço finita, num intervalo de tempo finito.

3. *Um algoritmo deve terminar.* O algoritmo deve levar a uma situação em que o objectivo tenha sido atingido e não existam mais instruções para ser executadas. Consideremos o seguinte algoritmo para elevar a pressão de um pneu acima de 28 libras: “enquanto a pressão for inferior a 28 libras, continue a introduzir ar”. É evidente que, se o pneu estiver furado, o algoritmo anterior pode não terminar (dependendo do tamanho do furo) e, portanto, não o vamos classificar como algoritmo.

O conceito de algoritmo é fundamental em informática. Existem mesmo pessoas que consideram a informática como o estudo dos algoritmos: o estudo de máquinas para executar algoritmos, o estudo dos fundamentos dos algoritmos e a análise de algoritmos.

1.3 Programas e algoritmos

Um algoritmo, escrito de modo a poder ser executado por um computador, tem o nome de *programa*. Uma grande parte deste livro é dedicada ao desenvolvimento de algoritmos, e à sua codificação utilizando uma linguagem de programação, o Python. Os programas que desenvolvemos apresentam aspectos semelhantes aos algoritmos informais apresentados na secção anterior. Nesta secção, discutimos alguns desses aspectos.

Vimos que a receita dos rebuçados de ovos era constituída por uma descrição dos objectos a manipular (500 g de açúcar, 5 gemas de ovos) e uma descrição das acções a efectuar sobre esses objectos (leva-se o açúcar ao lume, deixa-se ferver até fazer ponto de pérola). A constituição de um programa é semelhante à de uma receita.

Num programa, existem entidades que são manipuladas pelo programa e existe uma descrição, numa linguagem apropriada, de um algoritmo que especifica as operações a realizar sobre essas entidades. Em algumas linguagens de programação, por exemplo, o C e o Java, todas as entidades manipuladas por um programa têm que ser descritas no início do programa, noutras linguagens, como é o caso do Python, isso não é necessário.

No caso das receitas de culinária, as entidades a manipular podem existir antes do início da execução do algoritmo (por exemplo, 500 g de açúcar) ou entidades que são criadas durante a sua execução (por exemplo, a massa de ovos). A manipulação destas entidades vai originar um produto que é o objectivo do algoritmo (no nosso exemplo, os rebuçados de ovos). Analogamente, nos nossos programas, iremos manipular valores de variáveis. As variáveis vão-se comportar de um modo análogo aos ingredientes da receita dos rebuçados de ovos. Tipicamente, o computador começa por receber certos valores para algumas das variáveis, após o que efectua operações sobre essas variáveis, possivelmente atribuindo valores a novas variáveis e, finalmente, chega a um conjunto de valores que constituem o resultado do programa.

As operações a efectuar sobre as entidades devem ser compreendidas pelo agente que executa o algoritmo. Essas acções devem ser suficientemente elementares para poderem ser executadas facilmente pelo agente que executa o algoritmo. É importante notar que, pelo facto de nos referirmos a estas acções como “acções elementares”, isto não significa que elas sejam operações atómicas (isto é, indecomponíveis). Elas podem referir-se a um conjunto de acções mais simples a serem executadas numa sequência bem definida.

1.3.1 Linguagens de programação

Definimos uma *linguagem de programação* como uma linguagem utilizada para escrever programas de computador. Existem muitos tipos de linguagens de programação. De acordo com as afinidades que estas apresentam com o modo como os humanos resolvem problemas, podem ser classificadas em linguagens máquina, linguagens “assembly” e linguagens de alto nível.

A *linguagem máquina* é a linguagem utilizada para comandar directamente as acções do computador. As instruções em linguagem máquina são constituídas por uma sequência de dois símbolos discretos, correspondendo à existência ou à

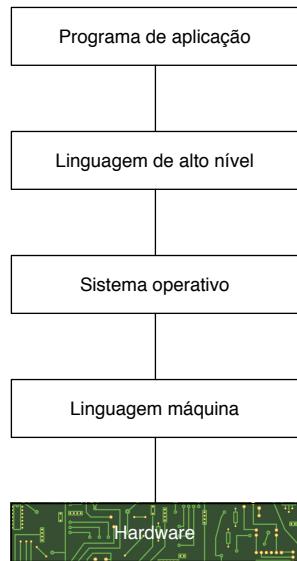


Figura 1.3: Alguns níveis de abstracção existentes num computador.

ausência de sinal (normalmente representados por 1 e por 0, respectivamente) e manipulam directamente entidades dentro do computador. A linguagem máquina é difícil de usar e de compreender por humanos e varia de computador para computador (é a sua linguagem nativa). A *linguagem “assembly”* é semelhante à linguagem máquina, diferindo desta no sentido em que usa nomes simbólicos com significado para humanos em lugar de sequências de zeros e de uns. Tal como a linguagem máquina, a linguagem “assembly” varia de computador para computador. As *linguagens de alto nível* aproximam-se das linguagens que os humanos usam para resolver problemas e, consequentemente, são muito mais fáceis de utilizar do que as linguagens máquina ou “assembly”, para além de poderem ser utilizadas em computadores diferentes. O Python é um exemplo de uma linguagem de alto nível.

Num computador, podemos identificar vários níveis de abstracção (Figura 1.3): ao nível mais baixo existem os circuitos electrónicos, o “hardware”, os quais são responsável por executar com grande velocidade as ordens dadas ao computador; o “hardware” pode ser directamente comandado através da linguagem máquina ou da linguagem “assembly”; o nível do *sistema operativo* permite-nos interagir com o computador, considerando que este contém ficheiros, organizados de

acordo com certa hierarquia, permite a manipulação desses ficheiros, e permite a interacção entre o nosso computador e o mundo exterior, o qual é composto por outros computadores e equipamento periférico, por exemplo impressoras; através do sistema operativo, podemos utilizar linguagens de programação de alto nível, de que o Python é um exemplo; finalmente, através das linguagens de alto nível, escrevemos programas de aplicação que fazem com que o computador resolva problemas específicos.

Para que os computadores possam “entender” os programas escritos numa linguagem de alto nível (recorda-se que a linguagem máquina é a linguagem que o computador comprehende), existem programas que traduzem as instruções de linguagens de alto nível em linguagem máquina, chamados *processadores da linguagem*. Existem fundamentalmente dois processos para fazer esta tradução, conhecidos por *compilação* e por *interpretação*. No caso do Python, isto é feito através de um programa chamado o *interpretador*, que recebe instruções em Python e que é capaz de executar as acções correspondentes a cada uma delas.

1.3.2 Exemplo de um programa

Apresentamos um exemplo de algoritmo para calcular a soma dos 100 primeiros inteiros positivos e o programa correspondente em Python. Para isso, vamos analisar o nosso comportamento ao resolver este problema utilizando uma calculadora. O objectivo da nossa apresentação é fornecer uma ideia intuitiva dos passos e do raciocínio envolvidos na geração de um programa e, simultaneamente, dar uma primeira ideia de um programa em Python. Podemos descrever as acções a executar para resolver este problema através da seguinte sequência de comandos a fornecer à calculadora:

Limpar o visor da calculadora

Carregar na tecla 1

Carregar na tecla +

Carregar na tecla 2

Carregar na tecla +

Carregar na tecla 3

Carregar na tecla +

...

Carregar na tecla 1

Carregar na tecla 0

Carregar na tecla 0

Carregar na tecla =

Os símbolos “...” na nossa descrição de acções indicam que existe um padrão que se repete ao longo da nossa actuação e portanto não é necessário enumerar todos os passos, porque podemos facilmente gerar e executar os que estão implícitos. No entanto, a existência destes símbolos não permite qualificar o conjunto de instruções anteriores como “algoritmo”, pois a característica do rigor deixa de se verificar. Para executar este conjunto de instruções é necessário ter a capacidade de compreender quais são os passos subentendidos por “...”.

Para transformar as instruções anteriores num algoritmo que possa ser executado por um computador, é necessário tornar explícito o que está implícito. Note-se, no entanto, que para explicitar todos os passos do algoritmo anterior teríamos mais trabalho do que se executássemos o algoritmo nós próprios, pelo que será conveniente encontrarmos uma formulação alternativa. Para isso, vamos reflectir sobre o processo de cálculo aqui descrito. Existem duas grandezas envolvidas neste processo de cálculo, a soma corrente (que aparece, em cada instante, no visor da calculadora) e o número a ser adicionado à soma (o qual é mantido na nossa cabeça). Cada vez que um número é adicionado à soma corrente, mentalmente, aumentamos em uma unidade o próximo número a ser adicionado. Se quisermos exprimir este processo de um modo rigoroso, necessitamos de recorrer a duas variáveis, uma para representar a soma corrente (à qual chamaremos *soma*), e a outra para representar o número que mantemos na nossa cabeça (a que chamaremos *numero*). Os passos repetitivos que executamos são:

A *soma* toma o valor de *soma* + *numero*

O *numero* toma o valor de *numero* + 1

O ciclo que executamos ao calcular a soma dos 100 primeiros inteiros positivos é:

enquanto o *numero* for menor ou igual a 100

 A *soma* toma o valor de *soma* + *numero*

 O *numero* toma o valor de *numero* + 1

Convém agora relembrar as operações que efectuamos antes de começar a executar esta sequência repetitiva de operações: (1) limpámos o visor da calculadora,

isto é estabelecemos que o valor inicial da variável *soma* era zero; (2) estabelecemos que o primeiro *numero* a ser adicionado à soma era um. Com estes dois aspectos em mente, poderemos dizer que a sequência de passos a seguir para calcular a soma dos 100 primeiros inteiros positivos é:

A *soma* toma o valor de 0
 O *numero* toma o valor de 1
 enquanto o *numero* for menor ou igual a 100
 A *soma* toma o valor de *soma* + *numero*
 O *numero* toma o valor de *numero* + 1

Em matemática, operações como “toma o valor de” são normalmente representadas por um símbolo (por exemplo, $=$). Em programação, esta operação é também representada por um símbolo ($=$, $\mathbf{:=}$, etc., dependendo da linguagem de programação utilizada). Se adoptarmos o símbolo utilizado em Python, $=$, o nosso algoritmo será representado por:

```
soma = 0
numero = 1
enquanto numero ≤ 100
    soma = soma + numero
    numero = numero + 1
```

Esta descrição é uma versão muito aproximada de um programa em Python para calcular a soma dos primeiros 100 inteiros positivos, o qual é o seguinte:

```
def prog_soma():
    soma = 0
    numero = 1
    while numero <= 100:
        soma = soma + numero
        numero = numero + 1
    print('O valor da soma é: ', soma)
```

É importante notar que existe uma fórmula que nos permite calcular a soma dos primeiros 100 inteiros positivos, se os considerarmos como uma progressão

aritmética de razão um, a qual é dada por:

$$soma = \frac{100 \cdot (1 + 100)}{2}$$

e, consequentemente, poderíamos utilizar esta fórmula para obter o valor desejado da soma, o que poderá ser representado pelo seguinte programa em Python⁵:

```
def prog_soma():
    soma = (100 * (1 + 100)) // 2
    print('O valor da soma é: ', soma)
```

Um aspecto importante a reter a propósito deste exemplo é o facto de normalmente não existir apenas um algoritmo (e consequentemente apenas um programa) para resolver um dado problema. Estes algoritmos podem ser muito diferentes entre si.

1.4 Sintaxe e semântica

O Python, como qualquer linguagem, apresenta dois aspectos distintos: as frases da linguagem e o significado associado às frases. Estes aspectos são denominados, respectivamente, *sintaxe* e *semântica* da linguagem.

Ao estudar uma linguagem de programação, é fundamental uma perfeita compreensão da sua sintaxe e da sua semântica: a sintaxe determina qual a constituição das frases que podem ser fornecidas ao computador, e a semântica vai determinar o que o computador vai fazer ao seguir cada uma dessas frases.

1.4.1 Sintaxe

A *sintaxe* de uma linguagem é o conjunto de regras que definem quais as relações válidas entre os componentes da linguagem, tais como as palavras e as frases. A sintaxe nada diz em relação ao significado das frases da linguagem.

Em linguagem natural, a sintaxe é conhecida como a gramática. Analogamente, em linguagens de programação, a sintaxe também é definida através

⁵Em Python, “*” representa a multiplicação e “//” representa a divisão inteira.

de gramáticas. Como exemplo da gramática de uma linguagem natural, vamos considerar um fragmento do português. Em português, uma frase simples é constituída por um sintagma nominal seguido de um sintagma verbal (um *sintagma* é uma subdivisão de uma frase). Por exemplo, a frase “O João acendeu o cachimbo” é constituída pelo sintagma nominal “O João” e pelo sintagma verbal “acendeu o cachimbo”. Uma das possíveis constituições de um sintagma nominal consiste num determinante (por exemplo, um artigo) seguido de um nome. No exemplo apresentado, o sintagma nominal “O João” é constituído pelo determinante “O”, seguido do nome “João”. Um sintagma verbal pode ser definido como sendo um verbo seguido de um sintagma nominal. No nosso exemplo, o sintagma verbal “acendeu o cachimbo” é constituído pelo verbo “acendeu” e pelo sintagma nominal “o cachimbo”. De acordo com o que temos vindo a apresentar sobre a estrutura de uma frase em português, a frase “O cachimbo acendeu o João” é uma frase sintacticamente correcta, embora sem qualquer significado. Isto apoia a ideia de que a sintaxe não se preocupa com o significado associado às frases, mas apenas com a definição da sua estrutura.

Como a sintaxe apenas se preocupa com o processo de combinação dos símbolos de uma linguagem, ela pode ser, na maior parte dos casos, facilmente formalizada. Os linguistas e os matemáticos estudaram as propriedades sintácticas das linguagens, e grande parte deste trabalho é aplicável às linguagens de programação.

Para descrever a sintaxe do Python, utilizaremos uma notação conhecida por notação BNF, a qual utiliza as seguintes regras:

1. Para designar um componente da linguagem escrevemo-lo entre parênteses angulares, “⟨” e “⟩”. Por exemplo, ⟨expressão⟩ representa uma expressão em Python. Não estamos, aqui, a falar de nenhuma expressão em particular, mas sim a falar de um componente genérico da linguagem. Por esta razão, os nomes que aparecem entre “⟨” e “⟩” são designados por *símbolos não terminais*. Um símbolo não terminal está sempre associado a um conjunto de entidades da linguagem, no nosso exemplo, todas as expressões.
2. Ao descrever a linguagem, teremos também de falar das entidades que aparecem explicitamente nas frases da linguagem, por exemplo o símbolo “+”. Os símbolos que aparecem nas frases da linguagem são chamados *símbolos terminais* e são escritos, em notação BNF, sem qualquer símbolo

especial à sua volta.

3. O símbolo “|” (lido “ou”) representa possíveis alternativas.
4. O símbolo “::=” (lido “é definido como”) serve para definir componentes da linguagem.
5. A utilização do carácter “+” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido uma ou mais vezes.
6. A utilização do carácter “*” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido zero ou mais vezes.
7. A utilização de chavetas, “{” e “}”, englobando símbolos terminais ou não terminais, significa que esses símbolos são opcionais.

Para aumentar a facilidade de leitura das nossas gramáticas, vamos usar dois tipos de letra, respectivamente para os **símbolos terminais** e para os **símbolos não terminais**: os símbolos terminais são escritos utilizando uma letra correspondente ao tipo máquina de escrever (como é feito nas palavras “símbolos terminais”, em cima); os símbolos não terminais são escritos usando um tipo helvética (como é feito nas palavras “símbolos não terminais”, em cima). Note-se, contudo, que esta convenção apenas serve para aumentar a facilidade de leitura das expressões e não tem nada a ver com as propriedades formais das nossas gramáticas.

Como primeiro exemplo, vamos considerar uma gramática em notação BNF para o subconjunto de português descrito no início desta secção. Dissemos que uma frase é constituída por um sintagma nominal seguido de um sintagma verbal, utilizando $\langle F \rangle$, $\langle SN \rangle$ e $\langle SV \rangle$ para representar, respectivamente, uma frase, um sintagma nominal e um sintagma verbal. A regra da gramática em notação BNF que define a estrutura de uma frase é

$$\langle F \rangle ::= \langle SN \rangle \langle SV \rangle$$

Esta regra é lida do seguinte modo: uma frase é definida como (“é definida como” é representado pelo símbolo “::=”) um sintagma nominal seguido por um sintagma verbal (“seguido por” está implícito quando usamos um símbolo não terminal seguido de outro símbolo não terminal).

Um sintagma nominal é um determinante seguido de um nome. Isto pode ser expresso através da seguinte regra em notação BNF:

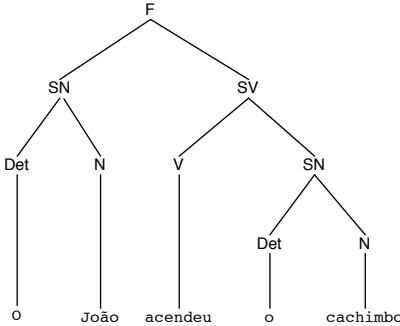


Figura 1.4: Estrutura da frase “O João acendeu o cachimbo”.

$\langle \text{SN} \rangle ::= \langle \text{Det} \rangle \langle \text{N} \rangle$

em que $\langle \text{Det} \rangle$ representa um determinante e $\langle \text{N} \rangle$ representa um nome. Finalmente, um sintagma verbal é definido como um verbo seguido de um sintagma nominal, o que pode ser escrito através da seguinte expressão em notação BNF:

$\langle \text{SV} \rangle ::= \langle \text{V} \rangle \langle \text{SN} \rangle$

em que $\langle \text{V} \rangle$ representa um verbo.

Podemos ainda dizer que **o** é um determinante, **João** e **cachimbo** são nomes e que **acendeu** é um verbo:

$\langle \text{Det} \rangle ::= \text{o}$

$\langle \text{N} \rangle ::= \text{João} \mid \text{cachimbo}$

$\langle \text{V} \rangle ::= \text{acendeu}$

A segunda regra é lida do seguinte modo: um nome é definido como **João** ou **cachimbo**, em que “ou” corresponde ao símbolo “|”.

Note-se que existe uma relação directa entre as regras da gramática em notação BNF e a estrutura das frases da linguagem. Na Figura 1.4 apresentamos a estrutura da frase “O João acendeu o cachimbo”. De facto, cada regra da gramática pode ser considerada como definindo uma estrutura em que o símbolo à sua esquerda está imediatamente acima dos símbolos à sua direita (dizemos que o símbolo à sua esquerda *domina* os símbolos à sua direita). Na Figura 1.4, $\langle \text{F} \rangle$ domina tanto $\langle \text{SN} \rangle$ como $\langle \text{SV} \rangle$, correspondendo à regra $\langle \text{F} \rangle ::= \langle \text{SN} \rangle \langle \text{SV} \rangle$; $\langle \text{SN} \rangle$ domina tanto $\langle \text{Det} \rangle$ como $\langle \text{N} \rangle$, correspondendo à regra $\langle \text{SN} \rangle ::= \langle \text{Det} \rangle \langle \text{N} \rangle$,

e assim sucessivamente. Nesta figura não representamos os parênteses angulares nos símbolos não terminais, pois a distinção entre símbolos terminais e símbolos não terminais é óbvia, os símbolos terminais não dominam quaisquer outros símbolos.

Como segundo exemplo, consideremos uma gramática em notação BNF para definir números binários. Informalmente, um número binário é apenas constituído pelos dígitos binários 0 e 1, podendo apresentar qualquer quantidade destes dígitos ou qualquer combinação entre eles. A seguinte gramática define números binários:

```
 $\langle \text{número binário} \rangle ::= \langle \text{dígito binário} \rangle |$ 
 $\quad \langle \text{dígito binário} \rangle \langle \text{número binário} \rangle$ 
 $\langle \text{dígito binário} \rangle ::= 0 | 1$ 
```

Nesta gramática os símbolos terminais são 0 e 1 e os símbolos não terminais são $\langle \text{número binário} \rangle$ e $\langle \text{dígito binário} \rangle$. A gramática tem duas regras. A primeira define a classe dos números binários, representados pelo símbolo não terminal $\langle \text{número binário} \rangle$, como sendo um $\langle \text{dígito binário} \rangle$, ou um $\langle \text{dígito binário} \rangle$ seguido de um $\langle \text{número binário} \rangle$ ⁶. A segunda parte desta regra diz simplesmente que um número binário é constituído por um dígito binário seguido por um número binário. Sucessivas aplicações desta regra levam-nos a concluir que um número binário pode ter tantos dígitos binários quantos queiramos (ou seja, podemos aplicar esta regra tantas vezes quantas desejarmos). Podemos agora perguntar quando é que paramos a sua aplicação. Note-se que a primeira parte desta mesma regra diz que um número binário é um dígito binário. Portanto, sempre que utilizamos a primeira parte desta regra, terminamos a sua aplicação. A segunda regra de produção define um dígito binário, representado pelo símbolo não terminal dígito binário, como sendo ou 0 ou 1. Na Figura 1.5 mostramos a estrutura do número binário 101 de acordo com a gramática apresentada.

Como último exemplo, consideremos a seguinte gramática que define um maior subconjunto do português do que o apresentado no primeiro exemplo (nesta gramática, $\langle \text{SP} \rangle$ representa um sintagma preposicional e $\langle \text{Prep} \rangle$ representa uma preposição):

⁶É importante compreender bem esta regra. Ela representa o primeiro contacto com uma classe muito importante de definições chamadas definições *recursivas* (ou definições por *recorrência*), nas quais uma entidade é definida em termos de si própria. As definições recursivas são discutidas em pormenor no Capítulo 6.

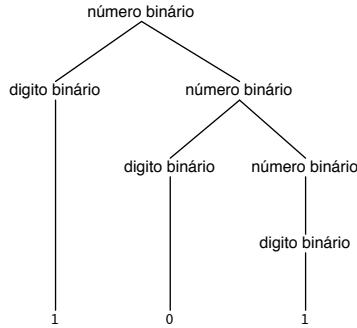


Figura 1.5: Estrutura do número binário 101.

$$\begin{aligned}
 \langle F \rangle &::= \langle SN \rangle \langle SV \rangle \mid \langle SN \rangle \langle SV \rangle \langle SP \rangle \\
 \langle SN \rangle &::= \langle N \rangle \mid \langle Det \rangle \langle N \rangle \mid \langle N \rangle \langle SP \rangle \mid \langle Det \rangle \langle N \rangle \langle SP \rangle \\
 \langle SV \rangle &::= \langle V \rangle \mid \langle V \rangle \langle SN \rangle \mid \langle V \rangle \langle SP \rangle \mid \langle V \rangle \langle SN \rangle \langle SP \rangle \\
 \langle SP \rangle &::= \langle Prep \rangle \langle SN \rangle \\
 \langle Det \rangle &::= \text{o} \mid \text{um} \\
 \langle N \rangle &::= \text{Eu} \mid \text{homem} \mid \text{telescópio} \mid \text{monte} \\
 \langle V \rangle &::= \text{vi} \\
 \langle Prep \rangle &::= \text{em} \mid \text{com}
 \end{aligned}$$

É importante dizer que uma gramática completa para o português é extremamente complicada e não pode ser formalizada utilizando apenas a notação BNF.

A gramática que apresentámos permite a construção de frases como “Eu vi o homem no monte com um telescópio”. Notemos que esta frase pode ser gerada, pelo menos, de dois modos diferentes (as estruturas correspondentes estão representadas nas figuras 1.6 e 1.7). Nestas estruturas, as palavras “em o” são contraídas para gerar a palavra “no”.

Uma gramática que pode originar a mesma frase com estruturas diferentes é denominada *ambígua*. As gramáticas para as linguagens naturais são inerentemente ambíguas. Um problema associado às gramáticas ambíguas é que às diferentes estruturas da mesma frase estão associados significados diferentes, e portanto, dada uma frase isolada, não é possível determinar univocamente o seu

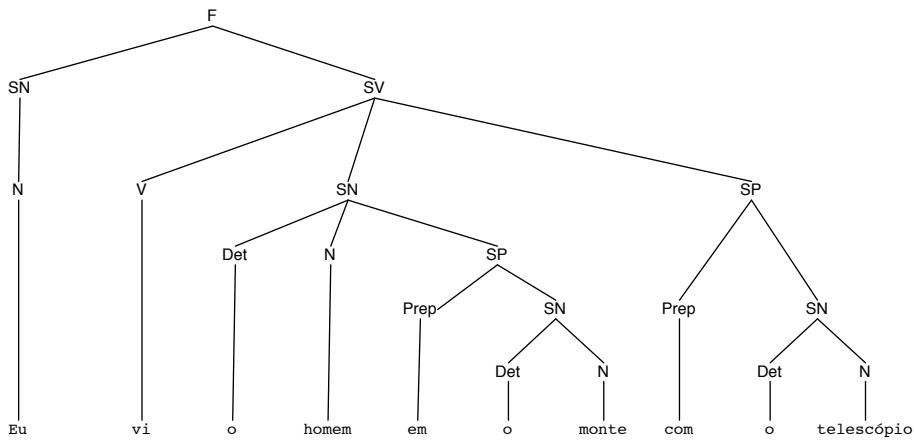


Figura 1.6: Possível estrutura da frase “Eu vi o homem no monte com um telescópio”.

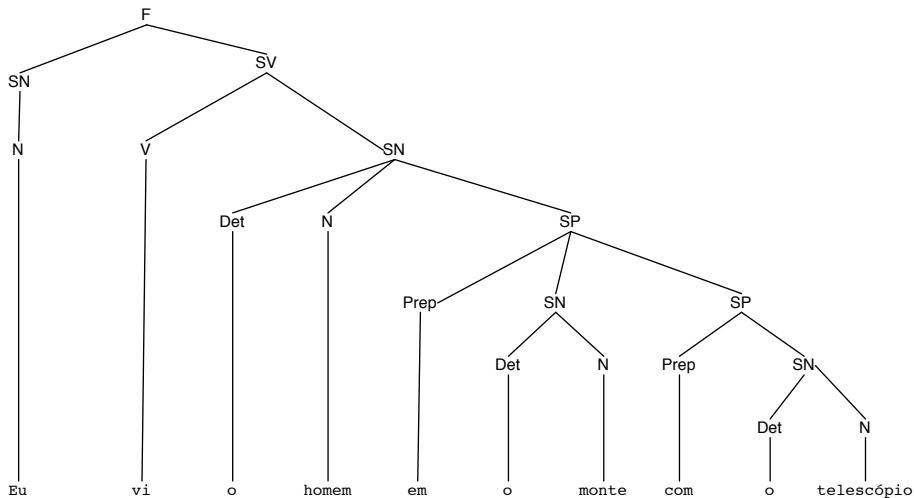


Figura 1.7: Estrutura alternativa da frase “Eu vi o homem no monte com um telescópio”.

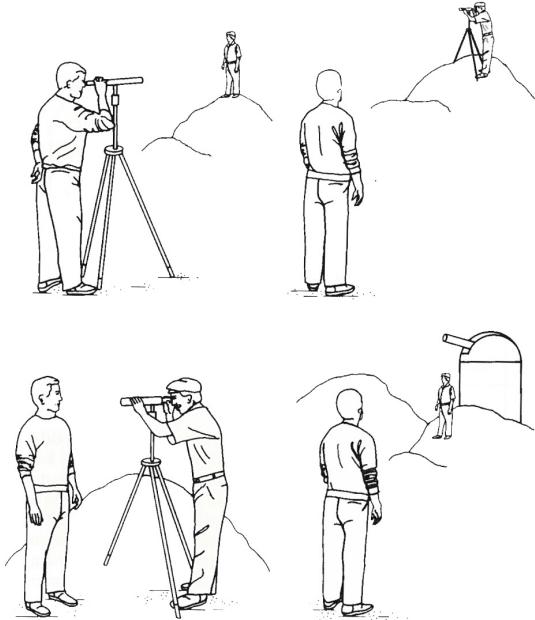


Figura 1.8: Possíveis significados da frase “Eu vi o homem no monte com um telescópio”.

significado. Na Figura 1.8⁷ apresentamos alguns dos possíveis significados da frase “Eu vi o homem no monte com um telescópio”. Em linguagem natural, a ambiguidade das frases é normalmente resolvida tendo em atenção o contexto da conversa em que a frase aparece. Por exemplo, a frase “Eu vi o homem no monte com um telescópio”, seguida de uma das frases “Limpei a lente para o ver melhor” ou “Falámos sobre astronomia”, selecciona uma das possíveis interpretações da Figura 1.8 (ver a Figura 1.9⁸).

Nas linguagens de programação, a ambiguidade sintáctica não pode existir. Ao analisar uma frase de uma linguagem de programação, o computador deve ser capaz de determinar exactamente a estrutura dessa frase e, consequentemente, o seu significado. Uma das preocupações na definição de uma linguagem de programação é a de evitar qualquer tipo de ambiguidade.

A notação utilizada para definir formalmente uma linguagem, no caso da notação

⁷ Adaptada de [Lenhart e Ringle, 1982], página 21, Figura 1.5. Reproduzida com autorização de Lawrence Erlbaum Associates, Inc. e dos autores.

⁸ *ibid.*



Figura 1.9: Resolução de ambiguidade na frase “Eu vi o homem no monte com um telescópio”.

BNF, “⟨”, “⟩”, “|”, “::=”, “{”, “}”, “+”, “*”, os símbolos não terminais e os símbolos terminais, é denominada *metalinguagem*, visto ser a linguagem que utilizamos para falar acerca de outra linguagem (ou a linguagem que está para além da linguagem). Um dos poderes da formalização da sintaxe utilizando metalinguagem é tornar perfeitamente clara a distinção entre “falar *acerca* da linguagem” e “falar *com* a linguagem”. A confusão entre linguagem e metalinguagem pode levar a paradoxos de que é exemplo a frase “esta frase é falsa”.

1.4.2 Semântica

“Then you should say what you mean,” the March Hare went on.

“I do,” Alice hastily replied; “at least—at least I mean what I say—that’s the same thing, you know.”

“Not the same thing a bit!” said the Hatter. “You might just as well say that “I see what I eat” is the same thing as “I eat what I see”!”

Lewis Carroll, *Alice’s Adventures in Wonderland*

A *semântica* de uma linguagem define qual o significado de cada frase da linguagem. A semântica nada diz quanto ao processo de geração das frases da linguagem. A descrição da semântica de uma linguagem de programação é muito mais difícil do que a descrição da sua sintaxe. Um dos processos de descrever a semântica de uma linguagem consiste em fornecer uma descrição em língua natural (por exemplo, em português) do significado, ou seja, das ações

que são realizadas pelo computador, de cada um dos possíveis componentes da linguagem. Este processo, embora tenha os inconvenientes da informalidade e da ambiguidade associadas às línguas naturais, tem a vantagem de fornecer uma perspectiva intuitiva da linguagem.

Cada frase em Python tem uma semântica, a qual corresponde às acções tomadas pelo Python ao executar essa frase, ou seja, o significado que o Python atribui à frase. Esta semântica é definida por regras para extrair o significado de cada frase, as quais são descritas neste livro de um modo incremental, à medida que novas frases são apresentadas. Utilizaremos o português para exprimir a semântica do Python.

1.4.3 Tipos de erros num programa

De acordo com o que dissemos sobre a sintaxe e a semântica de uma linguagem, deverá ser evidente que um programa pode apresentar dois tipos distintos de erros: erros de natureza sintáctica e erros de natureza semântica.

Os erros de natureza sintáctica, ou *erros sintácticos* resultam do facto de o programador não ter escrito as frases do seu programa de acordo com as regras da gramática da linguagem de programação utilizada. A detecção destes erros é feita pelo processador da linguagem, o qual fornece normalmente um diagnóstico sobre o que provavelmente está errado. Todos os erros de natureza sintáctica têm que ser corrigidos antes da execução das instruções, ou seja, o computador não executará nenhuma instrução sintaticamente incorrecta. Os programadores novatos passam grande parte do seu tempo a corrigir erros sintácticos, mas à medida que se tornam mais experientes, o número de erros sintácticos que originam é cada vez menor e a sua origem é detectada de um modo cada vez mais rápido.

Os erros de natureza semântica, ou *erros semânticos* (também conhecidos por *erros de lógica*) são erros em geral muito mais difíceis de detectar do que os erros de carácter sintáctico. Estes erros resultam do facto de o programador não ter expressado correctamente, através da linguagem de programação, as acções a serem executadas (o programador queria dizer uma coisa mas disse outra). Os erros semânticos podem-se manifestar pela geração de uma mensagem de erro durante a execução de um programa, pela produção de resultados errados ou pela geração de ciclos que nunca terminam. Neste livro apresentaremos

técnicas de programação que permitem minimizar os erros semânticos e, além disso, discutiremos métodos a utilizar para a detecção e correcção dos erros de natureza semântica de um programa.

Ao processo de detecção e correcção, tanto dos erros sintácticos como dos erros semânticos, dá-se o nome de *depuração* (do verbo depurar, tornar puro, limpar). Em inglês, este processo é denominado “*debugging*” e aos erros que existem num programa, tanto sintácticos como semânticos, chamam-se “*bugs*”⁹. O termo “*bug*” foi criado pela pioneira da informática Grace Murray Hopper (1906–1992). Em Agosto de 1945, Hopper e alguns dos seus associados estavam a trabalhar em Harvard com um computador experimental, o Mark I, quando um dos circuitos deixou de funcionar. Um dos investigadores localizou o problema e, com auxílio de uma pinça, removeu-o: uma traça com cerca de 5 cm. Hopper colou a traça, com fita gomada, no seu livro de notas e disse: “A partir de agora, sempre que um computador tiver problemas direi que ele contém insectos (*bugs*)”. A traça ainda hoje existe, juntamente com os registos das experiências, no “U.S. Naval Surface Weapons Center” em Dahlgren, Virginia, nos Estados Unidos da América¹⁰.

Para desenvolver programas, são necessárias duas competências fundamentais, a capacidade de *resolução de problemas* que corresponde à competência para formular o problema que deve ser resolvido pelo programa, criar uma solução para esse problema, através da sua divisão em vários subproblemas mais simples, e expressar essa solução de um modo rigoroso recorrendo a uma linguagem de programação e a capacidade de *depuração* que consiste em, através de uma análise rigorosa, perceber quais os erros existentes no programa e corrigi-los adequadamente. A depuração é fundamentalmente um trabalho de detective em que se analisa de uma forma sistemática o que está a ser feito pelo programa, formulando hipóteses sobre o que está mal e testando essas hipóteses através da modificação do programa. A depuração semântica é frequentemente uma tarefa difícil, requerendo espírito crítico e persistência.

⁹Do inglês, insectos.

¹⁰Ver [Taylor, 1984], página 44.

1.5 Notas finais

Neste capítulo apresentámos alguns conceitos básicos em relação à programação. Um computador, como uma máquina cuja função é a manipulação de símbolos, e as suas características fundamentais, ser automático, universal, electrónico e digital. Uma apresentação informal muito interessante sobre as origens dos computadores e dos matemáticos ligados à sua evolução pode ser consultada em [Davis, 2004].

Introduzimos a noção de programa, uma sequência de instruções escritas numa linguagem de programação, e o resultado originado pela execução de um programa, um processo computacional.

Apresentámos o conceito de algoritmo, uma sequência finita de instruções bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num período de tempo finito com uma quantidade de esforço finita, bem como as suas características, ser rigoroso, eficaz e dever terminar. O aspecto de um algoritmo ter que terminar é de certo modo controverso. Alguns autores, por exemplo [Hennie, 1977] e [Hermes, 1969], admitem que um algoritmo possa não terminar. Para estes autores, um algoritmo apenas apresenta as características de rigor e de eficácia. Outros autores, por exemplo [Brainerd e Landweber, 1974] e [Hopcroft e Ullman, 1969], distinguem entre procedimento mecânico – uma sequência finita de instruções que pode ser executada mecanicamente – e um algoritmo – um procedimento mecânico que é garantido terminar. Neste livro, adoptamos a segunda posição.

Finalmente apresentámos os dois aspectos associados a uma linguagem, a sintaxe e a semântica, e introduzimos a notação BNF para definir a sintaxe de uma linguagem. Como a sintaxe apenas se preocupa com o processo de combinação dos símbolos de uma dada linguagem, ela pode ser, na maior parte dos casos, facilmente formalizada. Os linguistas e os matemáticos estudaram as propriedades sintáticas das linguagens, e grande parte deste trabalho é aplicável às linguagens de programação. É particularmente importante o trabalho de Noam Chomsky ([Chomsky, 1957] e [Chomsky, 1959]), que classifica as linguagens em grupos. Grande parte das linguagens de programação pertence ao grupo 2, ou grupo das linguagens livres de contexto (do inglês, “context-free languages”).

A notação BNF foi inventada por John Backus e Peter Naur, e a sua primeira utilização significativa foi na definição da sintaxe da linguagem Algol 60. O

termo BNF significa “Backus-Naur Form”. Alguns autores, por exemplo [Hopcroft e Ullman, 1969] e [Ginsburg, 1966], atribuem ao termo “BNF” o significado “Backus Normal Form”.

1.6 Exercícios

1. Escreva uma gramática em notação BNF para definir um número inteiro.
Um número inteiro é um número, com ou sem sinal, constituído por um número arbitrário de dígitos.
2. Escreva uma gramática em notação BNF para definir um número real, o qual pode ser escrito quer em notação decimal quer em notação científica. Um real em notação decimal pode ou não ter sinal, e tem que ter ponto decimal, o qual é rodeado por dígitos. Por exemplo, +4.0, -4.0 e 4.0 são números reais em notação decimal. Um real em notação científica tem uma mantissa, a qual é um inteiro ou um real, o símbolo “e” e um expoente inteiro, o qual pode ou não ter sinal. Por exemplo, 4.2e-5, 2e4 e -24.24e+24 são números reais em notação científica.
3. Considere a seguinte gramática em notação BNF:

$$\begin{aligned}\langle S \rangle &::= \langle A \rangle a \\ \langle A \rangle &::= a \langle B \rangle \\ \langle B \rangle &::= \langle A \rangle a \mid b\end{aligned}$$

- (a) Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.
- (b) Quais das frases pertencem ou não à linguagem definida pela gramática.
Justifique a sua resposta.

aabaa

abc

abaa

aaaabaaaa

4. Considere a seguinte gramática em notação BNF:

$$\langle \text{idt} \rangle ::= \langle \text{letras} \rangle \langle \text{numeros} \rangle$$

$$\begin{aligned}\langle \text{letras} \rangle &::= \langle \text{letra} \rangle \mid \\ &\quad \langle \text{letra} \rangle \langle \text{letras} \rangle \\ \langle \text{numeros} \rangle &::= \langle \text{num} \rangle \mid \\ &\quad \langle \text{num} \rangle \langle \text{numeros} \rangle \\ \langle \text{letra} \rangle &::= A \mid B \mid C \mid D \\ \langle \text{num} \rangle &::= 1 \mid 2 \mid 3 \mid 4\end{aligned}$$

- (a) Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.
- (b) Quais das seguintes frases pertencem à linguagem definida pela gramática? Justifique a sua resposta.
- ABCD
1CD
A123CD
AAAAB12
- (c) Descreva informalmente as frases que pertencem à linguagem.
5. Escreva uma gramática em notação BNF que defina frases da seguinte forma: (1) as frases começam por **c**; (2) as frases acabam em **r**; (3) entre o **c** e o **r** podem existir tantos **a**'s e **d**'s quantos quisermos, mas tem que existir pelo menos um deles. São exemplos de frases desta linguagem: **car**, **cadar**, **cdr** e **cdddddrr**.
6. Considere a representação de tempo utilizada em relógios digitais, na qual aparecem as horas (entre 0 e 23), minutos e segundos. Por exemplo 10:23:45.

- (a) Descreva esta representação utilizando uma gramática em notação BNF.
- (b) Quais são os símbolos terminais e quais são os símbolos não terminais da sua gramática?
7. Considere a seguinte gramática em notação BNF:

$$\begin{aligned}\langle \text{Blop} \rangle &::= c \langle \text{Gulp} \rangle s \\ \langle \text{Gulp} \rangle &::= a \mid b \langle \text{Gulp} \rangle \mid x \langle \text{Bat} \rangle y \\ \langle \text{Bat} \rangle &::= x \mid \langle \text{Bat} \rangle y\end{aligned}$$

- (a) Diga quais são os símbolos terminais e quais são os símbolos não terminais da gramática.
- (b) Para cada uma das seguintes frases, diga se ela pertence ou não à linguagem definida pela gramática. Se não pertence, indique o primeiro símbolo que impossibilita a pertença.

cas

cGulps

cxxayys

cxayys

abxy

bay

cass

cbbbbbxxy

cxxy

x

8. Dada a seguinte gramática em notação BNF:

$\langle S \rangle ::= b \langle B \rangle$

$\langle B \rangle ::= b \langle C \rangle | a \langle B \rangle | b$

$\langle C \rangle ::= a$

- (a) Diga quais os símbolos terminais e quais são os símbolos não terminais desta gramática.
- (b) Diga, justificando, se as seguintes frases pertencerem ou não à linguagem definida pela gramática:

baaab

aabb

bba

baaaaaaba

Capítulo 2

Elementos básicos de programação

*The White Rabbit put on his spectacles.
‘Where shall I begin, please your Majesty?’ he asked.
‘Begin at the beginning,’ the king said, very gravely, ‘and go on till you come to the end: then stop.’*

Lewis Carroll, *Alice’s Adventures in Wonderland*

No capítulo anterior, vimos que um programa corresponde a um algoritmo escrito numa linguagem de programação. Dissemos também que qualquer programa é composto por instruções, as quais são construídas a partir de um pequeno número de operações elementares, entre as quais se encontram as operações de entrada de dados, as operações de saída de dados, operações matemáticas, a execução condicional e a repetição.

Neste capítulo apresentamos algumas noções básicas associadas a um programa, nomeadamente, algumas das operações correspondentes às operações elementares que acabámos de listar. No final deste capítulo, estaremos em condições de escrever alguns programas muito simples.

Um programa manipula variáveis usando um conjunto de operações. Apresentamos neste capítulo alguns dos tipos de valores que podem ser associados a variáveis, o modo de os combinar, juntamente com algumas instruções existentes em Python.

O Python é uma linguagem de programação, ou seja, corresponde a um formalismo para escrever programas. Um programa em Python pode ser introduzido e executado interactivamente num ambiente em que exista um interpretador do Python – uma “caixa electrónica” que compreenda as frases da linguagem Python.

A interacção entre um utilizador e o Python é feita através de um teclado e de um ecrã. O utilizador escreve frases através do teclado (aparecendo estas também no ecrã), e o computador responde, mostrando no ecrã o resultado de efectuar as acções indicadas na frase. Após efectuar as acções indicadas na frase, o utilizador fornece ao computador outra frase, e este ciclo repete-se até o utilizador terminar o trabalho. A este modo de interacção dá-se o nome de processamento interactivo. Em *processamento interactivo*, o utilizador dialoga com o computador, fornecendo uma frase de cada vez e esperando pela resposta do computador, antes de fornecer a próxima frase.

Ao iniciar uma sessão com o Python recebemos uma mensagem semelhante à seguinte:

```
Python 3.2.2 (v3.2.2:137e45f15c0b, Sep 3 2011, 16:48:10)
[GCC 4.0.1 (Apple Inc. build 5493)]
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

O símbolo “`>>>`” é uma indicação de que o Python está pronto para receber uma frase. Este símbolo é chamado o *carácter de pronto*¹. A utilização interactiva do Python corresponde à repetição de um ciclo em que o Python lê uma frase, efectua as acções indicadas na frase e escreve o resultado dessas acções. Este ciclo é chamado *ciclo lê-avalia-escreve*². Uma sessão em Python corresponde a um ciclo, tão longo quanto o utilizador desejar, de leitura de frases, execução das acções indicadas na frase e apresentação dos resultados.

Ao longo do livro apresentamos exemplos de interacções com o Python. Ao apresentar uma interacção, o que aparece depois do símbolo “`>>>`” corresponde à informação que é fornecida ao Python, e o que aparece na linha, ou linhas, seguintes corresponde à resposta que é fornecida pelo Python.

¹Do inglês, “prompt character”.

²Do inglês “read-eval-print loop”.

Uma frase em Python é designada por um *comando*. Um comando pode ser uma expressão, uma instrução ou uma definição. Em notação BNF, um comando é definido do seguinte modo:

$$\langle \text{comando} \rangle ::= \langle \text{expressão} \rangle \mid \langle \text{instrução} \rangle \mid \langle \text{definição} \rangle$$

Começamos por analisar as expressões em Python, após o que consideramos algumas instruções elementares. O conceito de *definição* é introduzido no próximo capítulo.

2.1 Expressões

Um dos tipos de entidades que utilizamos nos nossos programas corresponde a expressões. Por definição, uma *expressão* é uma entidade computacional que tem um valor. Usamos o termo *entidade computacional* para designar, de um modo genérico, uma entidade que existe dentro de um programa.

Uma expressão em Python pode ser uma constante, uma expressão composta, um nome ou uma aplicação de função. Em notação BNF, uma expressão é definida do seguinte modo:

$$\langle \text{expressão} \rangle ::= \langle \text{constante} \rangle \mid \langle \text{expressão composta} \rangle \mid \langle \text{nome} \rangle \mid \langle \text{aplicação de função} \rangle$$

Nesta secção, vamos considerar expressões correspondentes a constantes e expressões compostas utilizando algumas operações básicas. A aplicação de função é abordada no Capítulo 3

2.1.1 Constantes

Para os efeitos deste capítulo, consideramos que as constantes em Python podem ser números, valores lógicos ou cadeias de caracteres. Sempre que é fornecida uma constante ao Python, este devolve a constante como resultado da avaliação. Ou seja, o valor de uma constante é a própria constante (o Python mostra a representação externa da constante). A *representação externa* de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do modo como esta é representada internamente no computador (a *representação interna*), a qual, como sabemos, é feita recorrendo apenas aos símbolos 0 e 1. Por exemplo, a representação externa do inteiro 1958 é 1958 ao

passo que a sua representação interna é 11110100110.

A seguinte interacção mostra a resposta do Python quando lhe fornecemos algumas constantes:

```
>>> 1958
1958
>>> -1
-1
>>> +2
2
>>> 655484877641416186376754588877162243232221200091999228887333
655484877641416186376754588877162243232221200091999228887333
>>> 3.5
3.5
>>> 65397518517617656541959.888666225224423331
6.539751851761766e+22
>>> 0.0000000000000000000000000000000000000000000000000000000000000001
1e-38
>>> True
True
>>> False
False
>>> 'Bom dia'
'Bom dia'
```

Da interacção anterior, podemos verificar que existem em Python os seguintes tipos de constantes:

1. *Números inteiros*. Estes correspondem a números sem parte decimal (com ou sem sinal) e podem ser arbitrariamente grandes.
2. *Números reais*. Estes correspondem a números com parte decimal (com ou sem sinal) e podem ser arbitrariamente grandes ou arbitrariamente pequenos. Os números reais com valores absolutos muito pequenos ou muito grandes são apresentados (eventualmente arredondados) em notação científica. Em notação científica, representa-se o número, com ou sem sinal, através de uma mantissa (que pode ser inteira ou real) e de uma

potência inteira de dez (o expoente) que multiplicada pela mantissa produz o número. A mantissa e o expoente são separados pelo símbolo “e”. São exemplos de números reais utilizando a notação científica: `4.2e+5` (= 420000.0), `-6e-8` (= -0.00000006).

3. *Valores lógicos.* Os quais são representados por `True` (*verdadeiro*) e `False` (*falso*).
4. *Cadeias de caracteres*³. As quais correspondem a sequências de caracteres. As constantes das cadeias de caracteres são representadas em Python delimitadas por plicas. O *conteúdo* da cadeia de caracteres corresponde a todos os caracteres da cadeia, com a excepção das plicas; o *comprimento* da cadeia é o número de caracteres do seu conteúdo. Por exemplo `'bom dia'` é uma cadeia de caracteres com 7 caracteres, `b`, `o`, `m`, ⁴, `d`, `i`, `a`.

2.1.2 Expressões compostas

Para além das constantes, em Python existe também um certo número de operações, as operações embutidas. Por *operações embutidas*⁵, também conhecidas por operações *pré-definidas* ou por operações *primitivas*, entendem-se operações que o Python conhece, independentemente de qualquer indicação que lhe seja dada por um programa. Em Python, para qualquer uma destas operações, existe uma indicação interna (um algoritmo) daquilo que o Python deve fazer quando surge uma expressão com essa operação.

As operações embutidas podem ser utilizadas através do conceito de expressão composta. Informalmente, uma *expressão composta* corresponde ao conceito de aplicação de uma operação a operandos. Uma expressão composta é constituída por um operador e por um certo número de operandos. Os operadores podem ser *unários* (se apenas têm um operando, por exemplo, o operador lógico `not` ou o operador `-` representando o simétrico) ou *binários* (se têm dois operandos, por exemplo, `+` ou `*`).

Em Python, uma *expressão composta* é definida sintaticamente do seguinte modo⁶:

³Uma cadeia de caracteres é frequentemente designada pelo seu nome em inglês, “string”.

⁴Este carácter corresponde ao espaço em branco.

⁵Do inglês, “built-in” operations.

⁶Iremos ver outros modos de definir expressões compostas.

```

⟨expressão composta⟩ ::= ⟨operador⟩ ⟨expressão⟩ |
    ⟨operador⟩ ((⟨expressão⟩)) |
    ⟨expressão⟩ ⟨operador⟩ ⟨expressão⟩ |
    ((⟨expressão⟩) ⟨operador⟩ ⟨expressão⟩)

```

As duas primeiras linhas correspondem à utilização de operadores unários e as duas últimas à utilização de operadores binários.

Entre o operador e os operandos podemos inserir espaços em branco para aumentar a legibilidade da expressão, os quais são ignorados pelo Python.

Para os efeitos da apresentação nesta secção, consideramos que um operador corresponde a uma operação embutida.

Utilizando expressões compostas com operações embutidas, podemos originar a seguinte interacção (a qual utiliza operadores cujo significado é óbvio, exceptuando o operador * que representa a multiplicação):

```

>>> 2012 - 1958
54
>>> 3 * (24 + 12)
108
>>> 3.0 * (24 + 12)
108.0
>>> 7 > 12
False
>>> 23 / 7 * 5 + 12.5
28.928571428571427

```

Uma questão que surge imediatamente quando consideramos expressões compostas diz respeito à ordem pela qual as operações são efectuadas. Por exemplo, qual o denominador da última expressão apresentada? 7? 7*5? 7*5+12.5? É evidente que o valor da expressão será diferente para cada um destes casos.

Para evitar ambiguidade em relação à ordem de aplicação dos operadores numa expressão, o Python utiliza duas regras que especificam a ordem de aplicação dos operadores. A primeira regra, associada a uma *lista de prioridades de operadores*, especifica que os operadores com maior prioridade são aplicados antes dos operadores com menor prioridade; a segunda regra especifica qual a ordem de aplicação dos operadores quando se encontram dois operadores com a mesma

Prioridade	Operador
Máxima	Aplicação de funções not, - (simétrico) *, /, //, % +, - (subtração) <, >, ==, >=, <=, != and or
Mínima	

Tabela 2.1: Prioridade dos operadores em Python.

prioridade. Na Tabela 2.1 apresentamos a lista de prioridades dos operadores em Python (estas prioridades são, de modo geral, adoptadas em todas as linguagens de programação). Quando existem dois (ou mais) operadores com a mesma prioridade, eles são aplicados da esquerda para a direita. A utilização de parêntesis permite alterar a ordem de aplicação dos operadores.

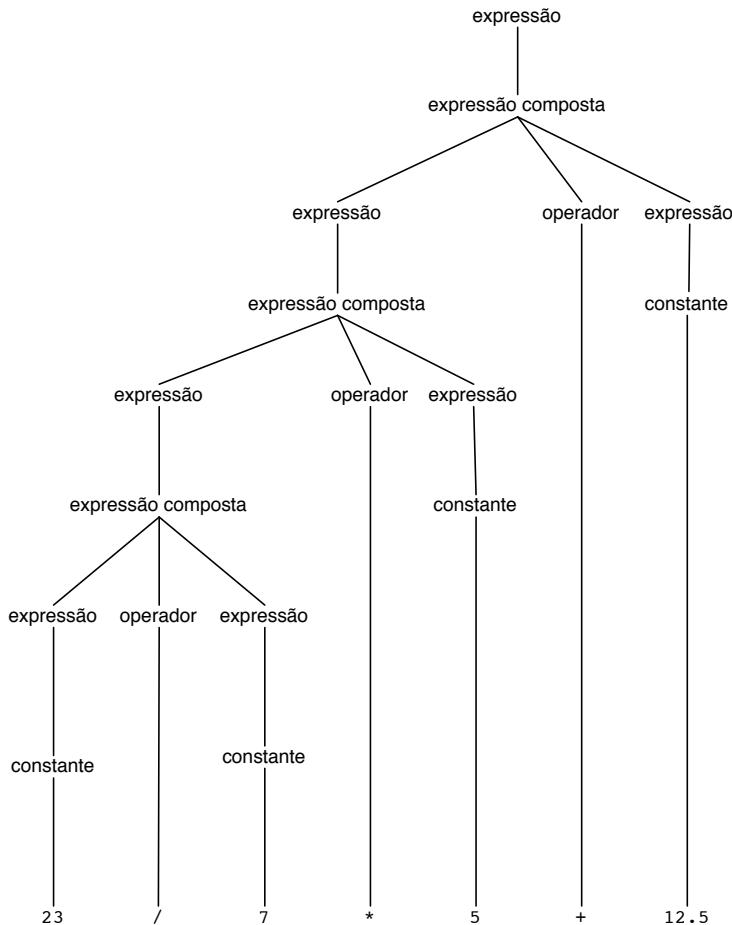
Na Figura 2.1 apresentamos a estrutura da expressão $23 / 7 * 5 + 12.5$, tendo em atenção as regras de aplicação de operadores utilizadas pelo Python.

2.2 Tipos elementares de informação

Em Matemática, é comum classificar as grandezas de acordo com certas características importantes. Existe uma distinção clara entre grandezas reais, grandezas complexas e grandezas do tipo lógico, entre grandezas representando valores individuais e grandezas representando conjuntos de valores, etc. De modo análogo, em programação, cada entidade computacional correspondente a um valor pertence a um certo tipo. Este tipo vai caracterizar a possível gama de valores da entidade computacional e as operações a que pode ser sujeita.

A utilização de tipos para caracterizar entidades que correspondem a dados é muito importante em programação. Um *tipo de informação* é caracterizado por um *conjunto de entidades* (valores) e um *conjunto de operações* aplicáveis a essas entidades. Ao conjunto de entidades dá-se nome de *domínio do tipo*. Cada uma das entidades do domínio do tipo é designada por *elemento do tipo*.

Os tipos de informação disponíveis variam de linguagem de programação para linguagem de programação. De um modo geral, podemos dizer que os tipos de

Figura 2.1: Estrutura da expressão $23 / 7 * 5 + 12.5$.

informação se podem dividir em dois grandes grupos: os tipos elementares e os tipos estruturados. Os *tipos elementares* são caracterizados pelo facto de as suas constantes (os elementos do tipo) serem tomadas como indecomponíveis (ao nível da utilização do tipo). Como exemplo de um tipo elementar podemos mencionar o tipo lógico, que possui duas constantes, “verdadeiro” e “falso”. Em contraste, os *tipos estruturados* são caracterizados pelo facto de as suas constantes serem constituídas por um agregado de valores.

Em Python, como tipos elementares, existem, entre outros, o tipo inteiro, o tipo real e o tipo lógico.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$e_1 + e_2$	Inteiros	A soma dos valores de e_1 e e_2 .
$e_1 - e_2$	Inteiros	O resultado de subtrair e_2 a e_1 .
$-e$	Inteiro	O simétrico de e .
$e_1 * e_2$	Inteiros	O produto dos valores de e_1 por e_2 .
$e_1 // e_2$	Inteiros	A divisão inteira de e_1 por e_2 .
$e_1 \% e_2$	Inteiros	O resto da divisão inteira de e_1 por e_2 .
$\text{abs}(e)$	Inteiro	O valor absoluto de e .

Tabela 2.2: Operações sobre números inteiros.

2.2.1 O tipo inteiro

Os números inteiros, em Python designados por `int`⁷, são números sem parte decimal, podendo ser positivos, negativos ou zero. Sobre expressões de tipo inteiro podemos realizar as operações apresentadas na Tabela 2.2. Por exemplo,

```
>>> -12
-12
>>> 7 // 2
3
>>> 7 % 2
1
>>> 5 * (7 // 2)
15
>>> abs(-3)
3
```

2.2.2 O tipo real

Os números reais, em Python designados por `float`⁸, são números com parte decimal. Em Python, e na maioria das linguagens de programação, existem dois métodos para a representação das constantes do tipo real, a notação decimal e a notação científica.

⁷Do inglês, “integer”.

⁸Designação que está associada à representação de números reais dentro de um computador, a representação em *virgula flutuante* (em inglês, “floating point representation”).

1. A *notação decimal*, em que se representa o número, com ou sem sinal, por uma parte inteira, um ponto (correspondente à vírgula), e uma parte decimal. São exemplos de números reais em notação decimal, -7.236 , 7.0 e 0.76752 . Se a parte decimal ou a parte inteira forem zero, estas podem ser omitidas, no entanto a parte decimal e a parte inteira não podem ser omitidas simultaneamente. Assim, $7.$ e $.1$ correspondem a números reais em Python, respectivamente 7.0 e 0.1 .
2. A *notação científica* em que se representa o número, com ou sem sinal, através de uma *mantissa* (que pode ser inteira ou real) e de uma potência inteira de dez (o *expoente*) que multiplicada pela mantissa produz o número. A mantissa e o expoente são separados pelo símbolo “e”. São exemplos de números reais utilizando a notação científica, $4.2\text{e}5$ ($=420000.0$), $-6\text{e}-8$ ($=-0.0000006$). A notação científica é utilizada principalmente para representar números muito grandes ou muito pequenos.

A seguinte interacção mostra algumas constantes reais em Python:

```
>>> 7.7
7.7
>>> 7.
7.0
>>> .4
0.4
>>> 2000000000000000000000000000000000000000000.
2e+35
>>> .00000000000000000000000000000000000000000000000000001
1e-53
```

Sobre os números reais, podemos efectuar as operações apresentadas na Tabela 2.3. Por exemplo,

```
>>> 2.7 + 3.9
6.6
>>> 3.4 / 5.9
0.5762711864406779
```

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$e_1 + e_2$	Reais	A soma dos valores de e_1 e e_2 .
$e_1 - e_2$	Reais	O resultado de subtrair e_2 a e_1 .
$-e$	Real	O simétrico de e .
$e_1 * e_2$	Reais	O produto dos valores de e_1 por e_2 .
e_1 / e_2	Reais	A divisão de e_1 por e_2 .
$\text{abs}(e)$	Real	O valor absoluto de e .

Tabela 2.3: Operações sobre números reais.

Notemos que existem operações aparentemente em comum entre os números inteiros e os números reais, por exemplo, a adição `+` e a multiplicação `*`. Dentro do computador, os números inteiros e os números reais são representados de modos diferentes. Ou seja, o inteiro 1 e o real 1.0 não correspondem, dentro do computador, à mesma entidade computacional. As relações existentes em Matemática entre o conjunto dos inteiros e o conjunto dos reais, $\mathbb{Z} \subset \mathbb{R}$, não existem deste modo claro em relação à representação de números, num computador os inteiros não estão contidos nos reais. Estes tipos formam conjuntos disjuntos no que respeita à representação das suas constantes. No entanto, as operações definidas sobre números sabem lidar com estas diferentes representações, originando os resultados que seriam de esperar em termos de operações aritméticas.

O que na realidade se passa dentro do computador é que cada operação sobre números (por exemplo, a operação de adição, `+`) corresponde a duas operações internas, uma para cada um dos tipos numéricos (por exemplo, a operação `+z` que adiciona números inteiros produzindo um número inteiro e a operação `+r` que adiciona números reais produzindo um número real). Estas operações estão associadas à mesma representação externa (`+`). Quando isto acontece, ou seja, quando a mesma representação externa de uma operação está associada a mais do que uma operação dentro do computador, diz-se que a operação está *sobre carregada*⁹.

Quando o Python tem de aplicar uma operação sobre carregada, determina o tipo de cada um dos operandos. Se ambos forem inteiros, aplica a operação `+z`, se ambos forem reais, aplica a operação `+r`, se um for inteiro e o outro real, converte o número inteiro para o real correspondente e aplica a operação

⁹Do inglês “overloaded”.

<i>Operação</i>	<i>Tipo do argumento</i>	<i>Tipo do valor</i>	Operação
<code>round(<i>e</i>)</code>	Real	Inteiro	O inteiro mais próximo do real <i>e</i> .
<code>int(<i>e</i>)</code>	Real	Inteiro	A parte inteira do real <i>e</i> .
<code>float(<i>e</i>)</code>	Inteiro	Real	O número real correspondente a <i>e</i> .

Tabela 2.4: Transformações entre reais e inteiros.

`+R`. Esta conversão tem o nome de *coerção*¹⁰, sendo demonstrada na seguinte interacção:

```
>>> 2 + 3.5
5.5
>>> 7.8 * 10
78.0
>>> 1 / 3
0.3333333333333333
```

Note-se que na última expressão, fornecemos dois inteiros à operação `/` que é definida sobre números reais, pelo que o Python converte ambos os inteiros para reais antes de aplicar a operação, sendo o resultado um número real.

O Python fornece operações embutidas que transformam números reais em inteiros e vice-versa. Algumas destas operações são apresentadas na Tabela 2.4. A seguinte interacção mostra a utilização destas operações:

```
>>> round(3.3)
3
>>> round(3.6)
4
>>> int(3.9)
3
>>> float(3)
3.0
```

¹⁰Do inglês, “coercion”.

e_1	e_2	$e_1 \text{ and } e_2$	$e_1 \text{ or } e_2$
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Tabela 2.5: Operações de conjunção e disjunção.

2.2.3 O tipo lógico

O tipo lógico, em Python designado por `bool`¹¹, apenas pode assumir dois valores, `True` (*verdadeiro*) e `False` (*falso*).

As operações que se podem efectuar sobre valores lógicos, produzindo valores lógicos, são de dois tipos, as operações unárias e as operações binárias.

- As operações unárias produzem um valor lógico a partir de um valor lógico. Existe uma operação unária em Python, `not`. A operação `not` muda o valor lógico, de um modo semelhante ao papel desempenhado pela palavra “não” em português. Assim, `not(True)` tem o valor `False` e `not(False)` tem o valor `True`.
- As operações binárias aceitam dois argumentos do tipo lógico e produzem um valor do tipo lógico. Entre estas operações encontram-se as operações lógicas tradicionais correspondentes à conjunção e à disjunção. A conjunção, representada por `and`, tem o valor `True` apenas se ambos os seus argumentos têm o valor `True` (Tabela 2.5). A disjunção, representada por `or`, tem o valor `False` apenas se ambos os seus argumentos têm o valor `False` (Tabela 2.5).

¹¹Em honra ao matemático inglês George Boole (1815–1864).

2.3 Nomes e atribuição

'Don't stand there chattering to yourself like that,' Humpty Dumpty said, looking at her for the first time,' but tell me your name and your business.'

'My NAME is Alice, but -'

'It's a stupid name enough!' Humpty Dumpty interrupted impatiently. 'What does it mean?'

'MUST a name mean something?' Alice asked doubtfully.

'Of course it must,' Humpty Dumpty said with a sort laugh: 'MY name means the shape I am – and a good handsome shape it is, too. With a name like yours, you might be any shape, almost.'

Lewis Carroll, *Through the Looking Glass*

Um dos aspectos importantes em programação corresponde à possibilidade de usar nomes para designar entidades computacionais. A utilização de nomes corresponde a um nível de abstracção no qual deixamos de nos preocupar com a indicação directa da entidade computacional, referindo-nos a essa entidade pelo seu nome. A associação entre um nome e um valor é realizada através da *instrução de atribuição*, a qual tem uma importância fundamental numa classe de linguagens de programação chamadas *linguagens imperativas* (de que é exemplo o Python).

A instrução de atribuição em Python recorre à operação embutida `=`, o *operador de atribuição*. Este operador recebe dois operandos, o primeiro corresponde ao nome que queremos usar para nomear o valor resultante da avaliação do segundo operando, o qual é uma expressão. Em notação BNF, a instrução de atribuição é definida do seguinte modo:

$$\langle \text{instrução de atribuição} \rangle ::= \langle \text{nome} \rangle = \langle \text{expressão} \rangle \mid \\ \langle \text{nome} \rangle, \langle \text{instrução de atribuição} \rangle, \langle \text{expressão} \rangle$$

Antes de apresentar a semântica da instrução de atribuição e exemplos da sua utilização, teremos de especificar o que é um nome.

Os nomes são utilizados para representar entidades usadas pelos programas. Como estas entidades podem variar durante a execução do programa, os nomes são também conhecidos por *variáveis*. Um *nome* é definido formalmente através das seguintes expressões em notação BNF:

$$\langle \text{nome} \rangle ::= \langle \text{nome simples} \rangle \mid \\ \langle \text{nome indexado} \rangle \mid \\ \langle \text{nome composto} \rangle$$

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

Tabela 2.6: Nomes reservados em Python.

Neste capítulo, apenas consideramos $\langle\text{nome simples}\rangle$, sendo $\langle\text{nome indexado}\rangle$ introduzido na Secção 4.1 e $\langle\text{nome composto}\rangle$ introduzido na Secção 3.6.

Em Python, um $\langle\text{nome simples}\rangle$ é uma sequência de caracteres que começa por uma letra ou pelo carácter $_$:

$\langle\text{nome simples}\rangle ::= \langle\text{inicial}\rangle \langle\text{subsequente}\rangle^*$

$\langle\text{inicial}\rangle ::= \text{A} | \text{B} | \text{C} | \text{D} | \text{E} | \text{F} | \text{G} | \text{H} | \text{I} | \text{J} | \text{K} | \text{L} | \text{M} | \text{N} | \text{O} | \text{P} | \text{Q} | \text{R}$
 $| \text{S} | \text{T} | \text{U} | \text{V} | \text{X} | \text{Y} | \text{W} | \text{Z} | \text{a} | \text{b} | \text{c} | \text{d} | \text{e} | \text{f} | \text{g} | \text{h} | \text{i} |$
 $\text{j} | \text{k} | \text{l} | \text{m} | \text{n} | \text{o} | \text{p} | \text{q} | \text{r} | \text{s} | \text{t} | \text{u} | \text{v} | \text{x} | \text{y} | \text{w} | \text{z} | -$

$\langle\text{subsequente}\rangle ::= \text{A} | \text{B} | \text{C} | \text{D} | \text{E} | \text{F} | \text{G} | \text{H} | \text{I} | \text{J} | \text{K} | \text{L} | \text{M} | \text{N} | \text{O} | \text{P} |$
 $\text{Q} | \text{R} | \text{S} | \text{T} | \text{U} | \text{V} | \text{X} | \text{Y} | \text{W} | \text{Z} | \text{a} | \text{b} | \text{c} | \text{d} | \text{e} | \text{f} |$
 $\text{g} | \text{h} | \text{i} | \text{j} | \text{k} | \text{l} | \text{m} | \text{n} | \text{o} | \text{p} | \text{q} | \text{r} | \text{s} | \text{t} | \text{u} | \text{v} |$
 $\text{x} | \text{y} | \text{w} | \text{z} | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | -$

Estas expressões em notação BNF dizem-nos que um $\langle\text{nome simples}\rangle$ pode ter tantos caracteres quantos queiramos, tendo necessariamente de começar por uma letra ou pelo carácter $_$. São exemplos de nomes `Taxa_de_juros`, `Numero`, `def`, `factorial`, `_757`. Não são exemplos de nomes simples `5A` (começa por um dígito), `turma 10101` (tem um carácter, “ ”, que não é permitido) e `igual?` (tem um carácter, “?”, que não é permitido). Para o Python os nomes `xpto`, `Xpto` e `XPTO` são nomes diferentes. Alguns nomes são usados pelo Python, estando reservados pela linguagem para seu próprio uso. Estes nomes, chamados *nomes reservados* mostram-se na Tabela 2.6.

A instrução de atribuição apresenta duas formas distintas. Começamos por discutir a primeira forma, a qual corresponde à primeira linha da expressão BNF que define $\langle\text{instrução de atribuição}\rangle$, a que chamamos *atribuição simples*.

Ao encontrar a instrução $\langle\text{nome}\rangle = \langle\text{expressão}\rangle$, o Python começa por avaliar a $\langle\text{expressão}\rangle$ após o que associa $\langle\text{nome}\rangle$ ao valor da $\langle\text{expressão}\rangle$. A execução de

uma instrução de atribuição não devolve nenhum valor, mas sim altera o valor de um nome.

A partir do momento em que associamos um nome a um valor (ou nomeamos o valor), o Python passa a “conhecer” esse nome, mantendo uma memória desse nome e do valor que lhe está associado. Esta memória correspondente à associação de nomes a valores (ou, de um modo mais geral, à associação de nomes a entidades computacionais – de que os valores são um caso particular) tem o nome de *ambiente*. Um ambiente (também conhecido por *espaço de nomes*¹²) contém associações para todos os nomes que o Python conhece. Isto significa que no ambiente existem também associações para os nomes de todas as operações embutidas do Python, ou seja, as operações que fazem parte do Python.

Ao executar uma instrução de atribuição, se o nome não existir no ambiente, o Python insere o nome no ambiente, associando-o ao respectivo valor; se o nome já existir no ambiente, o Python substitui o seu valor pelo valor da expressão. Deste comportamento, podemos concluir que, num ambiente, o mesmo nome não pode estar associado a dois valores diferentes.

Consideremos a seguinte interacção com o Python:

```
>>> nota = 17
>>> nota
17
```

Na primeira linha surge uma instrução de atribuição. Ao executar esta instrução, o Python avalia a expressão 17 (uma constante) e atribui o seu valor à variável **nota**. A partir deste momento, o Python passa a “conhecer” o nome, **nota**, o qual tem o valor 17. A segunda linha da interacção anterior corresponde à avaliação de uma expressão e mostra que se fornecermos ao Python a expressão **nota** (correspondente a um nome), este diz que o seu valor é 17. Este resultado resulta de uma regra de avaliação de expressões que afirma que o valor de um nome é a entidade associada com o nome no ambiente em questão. Na Figura 2.2 mostramos a representação do ambiente correspondente a esta interacção. Um ambiente é representado por um rectângulo a cinzento, dentro do qual aparecem associações de nomes a entidades computacionais. Cada associação contém um nome, apresentado no lado esquerdo, ligado por uma seta ao seu valor.

¹²Do inglês “namespace”.

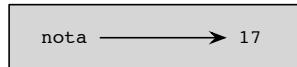


Figura 2.2: Representação de um ambiente.

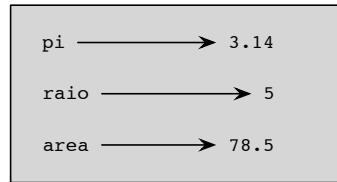
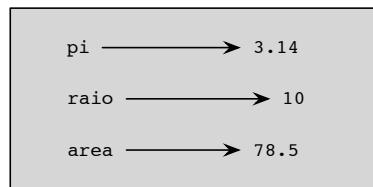
Figura 2.3: Ambiente resultante da execução de `nota = nota + 1`.

Consideremos agora a seguinte interacção com o Python, efectuada depois da interacção anterior:

```
>>> nota = nota + 1
>>> nota
18
>>> soma
NameError: name 'soma' is not defined
```

Segundo a semântica da instrução de atribuição, para a instrução apresentada na primeira linha, o Python começa por avaliar a expressão `nota + 1`, a qual tem o valor 18, em seguida associa o nome `nota` a este valor, resultando no ambiente apresentado na Figura 2.3. A instrução de atribuição `nota = nota + 1` tem o efeito de atribuir à variável `nota` o valor anterior de `nota` mais um. Este último exemplo mostra o carácter dinâmico da operação de atribuição: em primeiro lugar, a expressão à direita do símbolo `=` é avaliada, e, em segundo lugar, o valor resultante é atribuído à variável à esquerda deste símbolo. Isto mostra que uma operação de atribuição não corresponde a uma equação matemática, mas sim a um processo de atribuir o valor da expressão à direita do operador de atribuição à variável à sua esquerda. Na interacção anterior, mostramos também que se fornecermos ao Python um nome que não existe no ambiente (`soma`), o Python gera um erro, dizendo que não conhece o nome.

Na seguinte interacção com o Python, começamos por tentar atribuir o valor 3 ao nome `def`, o qual corresponde a um nome reservado do Python (ver Tabela 2.6). O Python reage com um erro. Seguidamente, definimos valores para as variáveis `pi`, `raio` e `area`, resultando no ambiente apresentado na Figura 2.4.

Figura 2.4: Ambiente depois da definição de `pi`, `raio` e `area`.Figura 2.5: Ambiente depois da alteração do valor de `raio`.

```

>>> def = 3
Syntax Error: def = 3: <string>, line 15
>>> pi = 3.14
>>> raio = 5
>>> area = pi * raio * raio
>>> raio
5
>>> area
78.5
>>> raio = 10
>>> area
78.5
  
```

A interacção anterior também mostra que se mudarmos o valor da variável `raio` o valor de `area`, embora tenha sido calculado a partir do nome `raio`, não se altera (Figura 2.5).

Consideremos agora a segunda forma da instrução de atribuição, a qual é conhecida por *atribuição múltipla* e que corresponde à segunda linha da expressão BNF que define *(instrução de atribuição)*. Ao encontrar uma instrução da forma

$\langle \text{nome}_1 \rangle, \langle \text{nome}_2 \rangle, \dots, \langle \text{nome}_n \rangle = \langle \text{exp}_1 \rangle, \langle \text{exp}_2 \rangle, \dots, \langle \text{exp}_n \rangle,$

o Python começa por avaliar as expressões $\langle \text{exp}_1 \rangle, \langle \text{exp}_2 \rangle, \dots, \langle \text{exp}_n \rangle$ (a ordem da avaliação destas expressões é irrelevante), após o que associa $\langle \text{nome}_1 \rangle$ ao valor da expressão $\langle \text{exp}_1 \rangle, \langle \text{nome}_2 \rangle$ ao valor da expressão $\langle \text{exp}_2 \rangle, \dots, \langle \text{nome}_n \rangle$ ao valor da expressão $\langle \text{exp}_n \rangle$.

O funcionamento da instrução de atribuição múltipla é ilustrado na seguinte interacção:

```
>>> nota_teste1, nota_teste2, nota_projecto = 15, 17, 14
>>> nota_teste1
15
>>> nota_teste2
17
>>> nota_projecto
14
```

Consideremos agora a seguinte interacção

```
>>> nota_1, nota_2 = 17, nota_1 + 1
NameError: name 'nota_1' is not defined
```

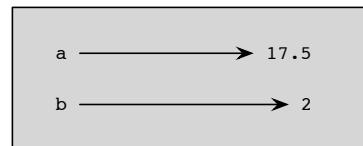
e analisemos a origem do erro. Dissemos que o Python começa por avaliar as expressões à direita do símbolo `=`, as quais são 17 e `nota_1 + 1`. O valor da constante 17 é 17. Ao avaliar a expressão `nota_1 + 1`, o Python não encontra o valor de `nota_1` no ambiente, pelo que gera um erro semelhante ao apresentado na página 49.

Consideremos agora a seguinte interacção:

```
>>> a = 2
>>> b = 17.5
>>> a
2
>>> b
17.5
>>> a, b = b, a
>>> a
```



Figura 2.6: Ambiente.

Figura 2.7: Ambiente depois da execução de `a, b = b, a`.

```
17.5
>>> b
2
```

Ao executar a instrução `a, b = b, a`, o Python começa por avaliar as expressões `b` e `a`, cujos valores são, respectivamente `17.5` e `2` (Figura 2.6). Em seguida, o valor `17.5` é atribuído à variável `a` e o valor `2` é atribuído à variável `b` (Figura 2.7). Ou seja a instrução `a, b = b, a` tem o efeito de trocar os valores das variáveis `a` e `b`.

A instrução de atribuição é a primeira instrução do Python que considerámos. Ao passo que uma expressão tem um valor, e consequentemente quando nos referimos às acções realizadas pelo Python para calcular o valor de uma expressão dizemos que a expressão é *avaliada*, uma instrução não tem um valor mas causa a realização de certas acções, por exemplo a atribuição de um nome a uma variável. Por esta razão, quando nos referimos às acções efectuadas pelo Python associadas a uma instrução dizemos que a *instrução é executada*.

Operação	Tipo dos argumentos	Valor
$e_1 == e_2$	Números	Tem o valor <code>True</code> se e só se os valores das expressões e_1 e e_2 são iguais.
$e_1 != e_2$	Números	Tem o valor <code>True</code> se e só se os valores das expressões e_1 e e_2 são diferentes.
$e_1 > e_2$	Números	Tem o valor <code>True</code> se e só se o valor da expressão e_1 é maior do que o valor da expressão e_2 .
$e_1 >= e_2$	Números	Tem o valor <code>True</code> se e só se o valor da expressão e_1 é maior ou igual ao valor da expressão e_2 .
$e_1 < e_2$	Números	Tem o valor <code>True</code> se e só se o valor da expressão e_1 é menor do que o valor da expressão e_2 .
$e_1 <= e_2$	Números	Tem o valor <code>True</code> se e só se o valor da expressão e_1 é menor ou igual ao valor da expressão e_2 .

Tabela 2.7: Operadores relacionais.

2.4 Predicados e condições

Uma operação que produz resultados do tipo lógico chama-se um *predicado*. Uma expressão cujo valor é do tipo lógico chama-se uma *condição*¹³. As condições podem ser combinadas através de operações lógicas. Entre outros, no Python existem, como operações embutidas, os operadores relacionais que se apresentam na Tabela 2.7. As operações lógicas foram apresentadas na Secção 2.2.3.

A seguinte interacção com o Python mostra a utilização de operadores relacionais e de operações lógicas (para compreender a última expressão fornecida ao Python, recorde-se a prioridade das operações apresentada na Tabela 2.1.):

```
>>> nota = 17
>>> 3 < nota % 2
False
>>> 3 < nota // 2
True
>>> 4 > 5 or 2 < 3
True
```

¹³Na realidade, o Python trata qualquer expressão como uma condição. Se o valor da expressão for zero ou `False`, esta é considerada como *falsa*, em caso contrário, é considerada como *verdadeira*. Neste livro tomamos uma atitude menos permissiva, considerando que uma condição é uma expressão cujo valor é ou `True` ou `False`.

O Python permite simplificar algumas operações envolvendo operadores relacionais. Consideremos a expressão $1 < 3 < 5$ que normalmente é usada em Matemática. Considerando a sua tradução directa para Python, $1 < 3 < 5$, e a prioridade dos operadores, esta operação deverá ser avaliada da esquerda para a direita ($1 < 3$) < 5 , dando origem a `True < 5` que corresponde a uma expressão que não faz sentido avaliar. Na maioria das linguagens de programação, esta expressão deverá ser traduzida para `(1 < 3) and (3 < 5)`. O Python oferece-nos uma notação simplificada para escrever condições com operadores relacionais, assim a expressão `(1 < 3) and (3 < 5)` pode ser simplificada para `1 < 3 < 5`, como o mostram os seguintes exemplos:

```
>>> 2 < 4 < 6 < 9
True
>>> 2 < 4 > 3 > 1 < 12
True
```

Uma alternativa para tornar a notação mais simples, como a que acabámos de apresentar relativa aos operadores relacionais, é vulgarmente designada por *açúcar sintático*¹⁴.

2.5 Comunicação com o exterior

2.5.1 Leitura de dados

Durante a execução de um programa, o computador necessita normalmente de obter valores do exterior para efectuar a manipulação da informação. A obtenção de valores do exterior é feita através das operações de leitura de dados. As *operações de leitura* de dados permitem transmitir informação do exterior para o programa. Por “exterior” entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador. Neste capítulo apenas consideramos operações de leitura de dados em que os dados são fornecidos através do teclado. No Capítulo 8, consideraremos operações de leitura de dados localizados em ficheiros.

¹⁴Do inglês, “syntactic sugar”.

O Python fornece uma operação de leitura de dados, a função `input`. Esta função tem a seguinte sintaxe:

```
<leitura de dados> ::= input() |  
                      input(<informação>)
```

```
<informação> ::= <cadeia de caracteres>
```

Ao encontrar a função `input(<informação>)` o Python mostra no ecrã o conteúdo da cadeia de caracteres correspondente a `<informação>`, após o que lê todos os símbolos introduzidos no teclado até que o utilizador carregue na tecla “Return” (ou, em alguns computadores, a tecla “Enter”). O valor da função `input` é a cadeia de caracteres cujo conteúdo é a sequência de caracteres encontrada durante a leitura.

Consideremos as seguintes interacções:

```
>>> input('-> ')  
-> 5  
'5'
```

Neste caso o Python mostra o conteúdo da cadeia de caracteres `'-> '`, o qual corresponde a `->`, e lê o que é fornecido através do teclado, neste caso, 5 seguido de “Return”, sendo devolvida a cadeia de caracteres `'5'`.

```
>>> input()  
estou a escrever sem carácter de pronto  
'estou a escrever sem carácter de pronto'
```

Neste caso não é mostrada nenhuma indicação no ecrã do computador, ficando o Python à espera que seja escrita qualquer informação. Escrevendo no teclado “estou a escrever sem carácter de pronto” seguido de “Return”, a função `input` devolve a cadeia de caracteres `'estou a escrever sem carácter de pronto'`, e daí a aparente duplicação das duas últimas linhas na interacção anterior.

```
>>> input('Por favor escreva qualquer coisa\n-> ')  
Por favor escreva qualquer coisa  
-> 554 umas palavras 3.14  
'554 umas palavras 3.14'
```

Carácter escape	Significado
\\\	Barra ao contrário (\)
\'	Plica (')
\"	Aspas ("")
\a	Toque de campainha
\b	Retrocesso de um espaço
\f	Salto de página
\n	Salto de linha
\r	“Return”
\t	Tabulação horizontal
\v	Tabulação vertical

Tabela 2.8: Alguns caracteres de escape em Python.

O exemplo anterior, introduz algo de novo. Na cadeia de caracteres que é fornecida à função `input` aparece algo que não vimos até agora, a sequência `\n`. A isto chama-se um carácter de escape¹⁵. Um *carácter de escape* é um carácter não gráfico com um significado especial para um meio de escrita, por exemplo, uma impressora ou o ecrã do computador. Em Python, um carácter de escape corresponde a um carácter precedido por uma barra ao contrário, “\”. Na Tabela 2.8 apresentam-se alguns caracteres de escape existentes em Python.

Finalmente, a seguinte interacção mostra a atribuição do valor lido a uma variável, `a`.

```
>>> a = input('-> ')
-> teste
>>> a
'teste'
```

Sendo o valor da função `input` uma cadeia de caracteres, poderemos questionar como ler valores inteiros ou reais. Para isso teremos que recorrer à função embutida, `eval`, chamada a *função de avaliação*, a qual tem a seguinte sintaxe:

$\langle \text{função de avaliação} \rangle ::= \text{eval}(\langle \text{cadeia de caracteres} \rangle)$

A função `eval` recebe uma cadeia de caracteres e devolve o resultado de avaliar essa cadeia de caracteres como sendo uma expressão. Por exemplo:

¹⁵Do inglês “escape character”.

```
>>> eval('3 + 5')
8
>>> eval('2')
2
>>> eval('5 * 3.2 + 10')
26.0
>>> eval('fundamentos da programação')
Syntax Error: fundamentos da programação: <string>, line 114
```

Combinando a função de avaliação com a função `input` podemos obter o seguinte resultado:

```
>>> b = eval(input('Escreva um número: '))
Escreva um número: 4.56
>>> b
4.56
```

2.5.2 Escrita de dados

Após efectuar a manipulação da informação, é importante que o computador possa comunicar ao exterior os resultados a que chegou. Isto é feito através das operações de escrita de dados. As *operações de escrita de dados* permitem transmitir informação do programa para o exterior. Por “exterior” entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador. Neste capítulo apenas consideramos operações de escrita de dados em que os dados são escritos no ecrã. No Capítulo 8, consideramos operações de escrita de dados para ficheiros.

Em Python existe a função embutida, `print`, com a sintaxe definida pelas seguintes expressões em notação BNF:

```
(escrita de dados) ::= print() |
                     print(<expressões>)

<expressões> ::= <expressão> |
                  <expressão>,  <expressões>
```

Ao encontrar a função `print()`, o Python escreve uma linha em branco no

écrã. Ao encontrar a função `print(<exp1n>)`, o Python começa por avaliar cada uma das expressões $\langle \text{exp}_1 \rangle \dots \langle \text{exp}_n \rangle$, após o que escreve os seus valores, todos na mesma linha do ecrã, separados por um espaço em branco.

A seguinte interacção ilustra o uso da função `print`:

```
>>> a = 10
>>> b = 15
>>> print('a =', a, 'b =', b)
a = 10 b = 15
>>> print('a =', a, '\nb =', b)
a = 10
b = 15
```

Sendo `print` uma função, estamos à espera que esta devolva um valor. Se imediatamente após a interacção anterior atribuirmos à variável `c` o valor devolvido por `print` constatamos o seguinte¹⁶:

```
>>> c = print('a =', a, '\nb =', b)
>>> print(c)
None
```

Ou seja, `print` é uma função que não devolve nada! Em Python, Existem algumas funções cujo objectivo não é a produção de um valor, mas sim a produção de um *efeito*, a alteração de qualquer coisa. A função `print` é uma destas funções. Depois da avaliação da função `print`, o conteúdo do ecrã do nosso computador muda, sendo esse efeito a única razão da existência desta função.

2.6 Programas, instruções e sequenciação

Até agora a nossa interacção com o Python correspondeu a fornecer comandos (através de expressões e de uma instrução) ao Python, imediatamente a seguir ao carácter de pronto, e a receber a resposta do Python ao comando fornecido. De um modo geral, para instruirmos o Python a realizar uma dada tarefa fornecemos-lhe um programa, uma sequência de comandos, que o Python executa, comando a comando.

¹⁶No capítulo 14 voltamos a considerar a constante `None`.

Um *programa* em Python corresponde a uma sequência de zero ou mais definições seguida por instruções. Em notação BNF, um programa em Python, também conhecido por um *guião*¹⁷, é definido do seguinte modo:

$\langle \text{programa em Python} \rangle ::= \langle \text{definição} \rangle^* \langle \text{instruções} \rangle$

Um programa não contém directamente expressões, aparecendo estas associadas a definições e a instruções.

O conceito de $\langle \text{definição} \rangle$ é apresentado nos capítulos 3 e 10, pelo que os nossos programas neste capítulo apenas contêm instruções. Nesta secção, começamos a discutir os mecanismos através dos quais se pode especificar a ordem de execução das instruções de um programa e apresentar algumas das instruções que podemos utilizar num programa em Python. No fim do capítulo, estaremos aptos a desenvolver alguns programas em Python, extremamente simples mas completos.

O controle da sequência de instruções a executar durante um programa joga um papel essencial no funcionamento de um programa. Por esta razão, as linguagens de programação fornecem estruturas que permitem especificar qual a ordem de execução das instruções do programa.

Ao nível da linguagem máquina, existem dois tipos de estruturas de controle: a sequenciação e o salto. A sequenciação especifica que as instruções de um programa são executadas pela ordem em que aparecem no programa. O salto especifica a transferência da execução para qualquer ponto do programa. As linguagens de alto nível, de que o Python é um exemplo, para além da sequenciação e do salto (a instrução de salto pode ter efeitos perniciosos na execução de um programa e não será considerada neste livro), fornecem estruturas de controle mais sofisticadas, nomeadamente a selecção e a repetição.

A utilização de estruturas de controle adequadas contribui consideravelmente para a facilidade de leitura e manutenção de programas. De facto, para dominar a compreensão de um programa é crucial que as suas instruções sejam estruturadas de acordo com processos simples, naturais e bem compreendidos.

A *sequenciação* é a estrutura de controle mais simples, e consiste na especificação de que as instruções de um programa são executadas sequencialmente, pela ordem em que aparecem no programa. Sendo $\langle \text{inst1} \rangle$ e $\langle \text{inst2} \rangle$ símbolos não

¹⁷Do inglês “script”.

terminais que correspondem a instruções, a indicação de que a instrução $\langle \text{inst2} \rangle$ é executada imediatamente após a execução da instrução $\langle \text{inst1} \rangle$ é especificada, em Python escrevendo a instrução $\langle \text{inst2} \rangle$ numa linha imediatamente a seguir à linha em que aparece a instrução $\langle \text{inst1} \rangle$. Ou seja, o fim de linha representa implicitamente o operador de sequenciação. O conceito de sequência de instruções é definido através da seguinte expressão em notação BNF:

$$\begin{aligned}\langle \text{instruções} \rangle ::= & \langle \text{instrução} \rangle \boxed{\text{CR}} \mid \\ & \langle \text{instrução} \rangle \boxed{\text{CR}} \langle \text{instruções} \rangle\end{aligned}$$

Nesta definição, $\boxed{\text{CR}}$ é um símbolo terminal que corresponde ao símbolo obtido carregando na tecla “Return” do teclado (ou, em alguns computadores, a tecla “Enter”), ou seja, $\boxed{\text{CR}}$ corresponde ao fim de uma linha,

Consideremos o seguinte programa (sequência de instruções):

```
nota_1 = 17
nota_2 = 18
media = (nota_1 + nota_2) / 2
print('A média é', media)
```

Se instruirmos o Python para executar este programa, este escreve

A média é 17.5

o que mostra um exemplo da execução sequencial de instruções.

2.7 Selecção

Para desenvolver programas complexos, é importante que possamos especificar a execução condicional de instruções: devemos ter a capacidade de decidir se uma instrução ou grupo de instruções deve ou não ser executado, dependendo do valor de uma condição. Esta capacidade é por nós utilizada constantemente. Repare-se, por exemplo, nas instruções para o papagaio voador, apresentadas no Capítulo 1, página 8: “se o vento soprar forte deverá prender na argola inferior. Se o vento soprar fraco deverá prender na argola superior.”.

A instrução **if**¹⁸ permite a selecção entre duas ou mais alternativas. Depen-

¹⁸A palavra “if” traduz-se, em português, por “se”.

dendo do valor de uma condição, esta instrução permite-nos seleccionar uma de duas ou mais instruções para serem executadas. A sintaxe da instrução `if` é definida pelas seguintes expressões em notação BNF¹⁹:

```

⟨instrução if⟩ ::= if ⟨condição⟩: [CR]
    ⟨instrução composta⟩
    ⟨outras alternativas⟩*
    {⟨alternativa final⟩}

⟨outras alternativas⟩ ::= elif ⟨condição⟩: [CR]
    ⟨instrução composta⟩

⟨alternativa final⟩ ::= else: [CR]
    ⟨instrução composta⟩

⟨instrução composta⟩ ::= [TAB+] ⟨instruções⟩ [TAB-]

⟨condição⟩ ::= ⟨expressão⟩

```

Nesta definição, `[TAB+]` corresponde ao símbolo obtido carregando na tecla que efectua a tabulação e `[TAB-]` corresponde ao símbolo obtido desfazendo a acção correspondente a `[TAB+]`. Numa instrução composta, o efeito de `[TAB+]` aplica-se a cada uma das suas instruções, fazendo com que estas começem todas na mesma coluna. Este aspecto é conhecido por *paragrafação*. Uma instrução `if` estende-se por várias linhas, não deixando de ser considerada *uma única* instrução.

Na definição sintáctica da instrução `if`, a `⟨condição⟩` representa qualquer expressão do tipo lógico. Este último aspecto, a imposição de que expressão seja do tipo lógico, não pode ser feita recorrendo a expressões em notação BNF.

De acordo com esta definição,

```
if nota > 15:
    print('Bom trabalho')
```

corresponde a instruções `if`, mas

```
if nota > 15:
    print('Bom trabalho')
```

não corresponde a uma instrução `if`, pois falta o símbolo terminal `[TAB+]` antes da instrução `print('Bom trabalho')`.

¹⁹Existem outras alternativas para a instrução `if` que não consideramos neste livro.

Para apresentar a semântica da instrução `if` vamos considerar cada uma das suas formas:

1. Ao encontrar uma instrução da forma

```
if <cond>:  
    <instruções>
```

o Python começa por avaliar a expressão `<cond>`. Se o seu valor for `True`, o Python executa as instruções correspondentes a `<instruções>`; se o valor da expressão `<cond>` for `False`, o Python não faz mais nada relativamente a esta instrução `if`. Os efeitos desta semântica são mostrados na seguinte interacção:

```
>>> if 3 < 5:  
...     print('3 < 5')  
...  
3 < 5  
>>> if 3 > 5:  
...     print('3 < 5')  
...  
>>>
```

Nesta interacção com o Python, surge um novo *carácter de pronto*, correspondente a “...”. Esta nova forma do carácter de pronto corresponde à indicação do Python que está à espera de mais informação para terminar de ler o comando que está a ser fornecido. Esta informação termina quando o utilizador escreve uma linha em branco.

A última linha, apenas com o sinal de pronto, indica que a instrução na linha anterior foi executada, estando o Python à espera de receber novo comando.

2. Ao encontrar uma instrução da forma (a qual não usa `<outras alternativas>`, utilizando apenas a `<alternativa final>`)

```
if <cond>:  
    <instruções1>  
else:  
    <instruções2>
```

o Python começa por avaliar a expressão $\langle \text{cond} \rangle$. Se o seu valor for `True`, as instruções correspondentes a $\langle \text{instruções}_1 \rangle$ são executadas e as instruções correspondentes a $\langle \text{instruções}_2 \rangle$ não o são; se o seu valor for `False`, as instruções correspondentes a $\langle \text{instruções}_2 \rangle$ são executadas e as instruções correspondentes a $\langle \text{instruções}_1 \rangle$ não o são.

A seguinte interacção mostra a semântica da instrução `if`:

```
>>> nota = 15
>>> if nota > 16:
...     print('Bom trabalho')
... else:
...     print('Podes fazer melhor')
...     print('Estuda mais')
...
Podes fazer melhor
Estuda mais
>>> nota = 17
>>> if nota > 16:
...     print('Bom trabalho')
... else:
...     print('Podes fazer melhor')
...     print('Estuda mais')
...
Bom trabalho
```

3. De um modo geral, ao encontrar uma instrução da forma:

```
if <cond1>:
    <instruções1>
elif <cond2>:
    <instruções2>
elif <cond3>:
    <instruções3>
:
else:
    <instruçõesf>
```

o Python começa por avaliar a expressão $\langle \text{cond}_1 \rangle$. Se o seu valor for `True`, as instruções correspondentes a $\langle \text{instruções}_1 \rangle$ são executadas e a execução

da instrução `if` termina; se o seu valor for `False`, o Python avalia a expressão $\langle \text{cond}_2 \rangle$. Se o seu valor for `True`, as instruções correspondentes a $\langle \text{instruções}_2 \rangle$ são executadas e a execução da instrução `if` termina. Em caso contrário, o Python avalia a expressão $\langle \text{cond}_3 \rangle$, e assim sucessivamente. Se todas as condições forem falsas, o Python executa as instruções correspondentes a $\langle \text{instruções}_f \rangle$.

Consideremos a seguinte instrução `if`, a qual para um valor quantitativo de uma `nota` escreve o seu valor qualitativo equivalente, usando a convenção de que uma nota entre zero e 4 corresponde a mau, uma nota entre 5 e 9 corresponde a medíocre, uma nota entre 10 e 13 corresponde a suficiente, uma nota entre 14 e 17 corresponde a bom e uma nota superior a 17 corresponde a muito bom. Como exercício, o leitor deve convencer-se que embora o limite inferior das gamas de notas não seja verificado, esta instrução `if` realiza adequadamente o seu trabalho para qualquer valor de `nota`.

```
if nota <= 4:
    print('Mau')
elif nota <= 9:
    print('Mediocre')
elif nota <= 13:
    print('Suficiente')
elif nota <= 17:
    print('Bom')
else :
    print('Muito bom')
```

2.8 Repetição

Em programação é frequente ser preciso repetir a execução de um grupo de instruções, ou mesmo repetir a execução de todo o programa, para diferentes valores dos dados. Repare-se, por exemplo, na receita para os rebuçados de ovos, apresentada no Capítulo 1, página 7: “Leva-se o açúcar ao lume com um copo de água e deixa-se ferver até fazer ponto de pérola”, esta instrução corresponde a dizer que “enquanto não se atingir o ponto de pérola, deve-se deixar o açúcar com água a ferver”.

Em programação, uma sequência de instruções executada repetitivamente é chamada um *ciclo*. Um ciclo é constituído por uma sequência de instruções, o *corpo do ciclo*, e por uma estrutura que controla a execução dessas instruções, especificando quantas vezes o corpo do ciclo deve ser executado. Os ciclos são muito comuns em programação, sendo raro encontrar um programa sem um ciclo.

Cada vez que as instruções que constituem o corpo do ciclo são executadas, dizemos que se efectuou uma *passagem pelo ciclo*. As instruções que constituem o corpo do ciclo podem ser executadas qualquer número de vezes (eventualmente, nenhuma) mas esse número tem de ser finito. Há erros semânticos que podem provocar a execução interminável do corpo do ciclo, caso em que se diz que existe um *ciclo infinito*²⁰. Em Python, existem duas instruções que permitem a especificação de ciclos, a instrução `while`, que apresentamos nesta secção, e a instrução `for`, que apresentamos no Capítulo 4.

A instrução `while` permite especificar a execução repetitiva de um conjunto de instruções enquanto uma determinada expressão do tipo lógico tiver o valor verdadeiro. A sintaxe da instrução `while` é definida pela seguinte expressão em notação BNF²¹:

```
(instrução while) ::= while <condição>: [CR]
                           <instrução composta>
```

Na definição sintáctica da instrução `while`, a `<condição>` representa uma expressão do tipo lógico. Este último aspecto não pode ser explicitado recorrendo a expressões em notação BNF.

Na instrução `while <cond>: [CR] <inst>`, o símbolo não terminal `<inst>` corresponde ao corpo do ciclo e o símbolo `<cond>`, uma expressão do tipo lógico, representa a condição que controla a execução do ciclo.

A semântica da instrução `while` é a seguinte: ao encontrar a instrução `while <cond>: [CR] <inst>`, o Python calcula o valor de `<cond>`. Se o seu valor for `True`, o Python efectua uma passagem pelo ciclo, executando as instruções correspondentes a `<inst>`. Em seguida, volta a calcular o valor de `<cond>` e o processo repete-se enquanto o valor de `<cond>` for `True`. Quando o valor de `<cond>` for `False`, a execução do ciclo termina.

²⁰O conceito de ciclo infinito é teórico, pois, na prática ciclo acabará por terminar ou porque nós o interrompemos (fartamo-nos de esperar) ou porque os recursos computacionais se esgotam.

²¹A palavra “while” traduz-se em português por “enquanto”.

É evidente que a instrução que constitui o corpo do ciclo deve modificar o valor da expressão que controla a execução do ciclo, caso contrário, o ciclo pode nunca terminar (se o valor desta expressão é inicialmente `True` e o corpo do ciclo não modifica esta expressão, então estamos na presença de um ciclo infinito).

No corpo do ciclo, pode ser utilizada uma outra instrução existente em Python, a instrução `break`. A instrução `break` apenas pode aparecer dentro do corpo de um ciclo. A sintaxe da instrução `break` é definida pela seguinte expressão em notação BNF²²:

`<instrução break> ::= break`

Ao encontrar uma instrução `break`, o Python termina a execução do ciclo, independentemente do valor da condição que controla o ciclo.

Como exemplo da utilização de uma instrução `while`, vulgarmente designada por um ciclo `while`, consideremos a seguinte sequência de instruções:

```
soma = 0
num = eval(input('Escreva um inteiro\n(-1 para terminar): '))

while num != -1:
    soma = soma + num
    num = eval(input('Escreva um inteiro\n(-1 para terminar): '))

print('A soma é:', soma)
```

Estas instruções começam por estabelecer o valor inicial da variável `soma` como sendo zero²³, após o que efectuam a leitura de uma sequência de números inteiros positivos, calculando a sua soma. A quantidade de números a ler é desconhecida à partida. Para assinalar que a sequência de números terminou, fornecemos ao Python um número especial, no nosso exemplo, o número `-1`. Este valor é chamado o *valor sentinel*, porque assinala o fim da sequência a ser lida.

Neste programa inserimos linhas em branco para separar partes do programa, a inicialização de variáveis, o ciclo, e a escrita do resultado. As linhas em branco podem ser inseridas entre instruções, aumentando a facilidade de leitura do programa, mas não tendo qualquer efeito sobre a sua execução. As linhas em

²²A palavra “`break`” traduz-se em português por “fuga”, “quebra”, “pausa”, ou “ruptura”.

²³Esta operação é conhecida por *inicialização da variável*.

branco podem ser consideradas como a *instrução vazia*, a qual é definida pela seguinte sintaxe:

`(instrução vazia) ::=`

Ao encontrar uma instrução vazia, o Python não toma nenhuma acção.

Este programa utiliza uma instrução `while`, a qual é executada enquanto o número fornecido pelo utilizador for diferente de `-1`. O corpo deste ciclo é constituído por uma instrução composta, que é, por sua vez, constituída por duas instruções `soma = soma + num` (que actualiza o valor da soma) e `num = eval(input('Escreva um inteiro\n(-1 para terminar): '))` (que lê o número fornecido). Note-se que este ciclo pressupõe que já foi fornecido um número ao computador, e que o corpo do ciclo inclui uma instrução (`num = eval(input('Escreva um inteiro\n(-1 para terminar): '))`) que modifica o valor da condição que controla o ciclo (`num != -1`).

Existem dois aspectos importantes a lembrar relativamente à instrução `while`:

1. De um modo geral, o número de vezes que o corpo do ciclo é executado não pode ser calculado antecipadamente: a condição que especifica o término do ciclo é testada durante a execução do próprio ciclo, sendo impossível saber de antemão como vai prosseguir a avaliação.
2. Pode acontecer que o corpo do ciclo não seja executado nenhuma vez. Com efeito, a semântica da instrução `while` especifica que o valor da expressão que controla a execução do ciclo é calculado antes do início da execução do ciclo. Se o valor inicial desta expressão é `False`, o corpo do ciclo não é executado.

Consideremos agora um programa que lê um inteiro positivo e calcula a soma dos seus dígitos. Este programa lê um número inteiro, atribuindo-o à variável `num` e inicializa o valor da soma dos dígitos (variável `soma`) para zero.

Após estas duas acções, o programa executa um ciclo enquanto o número não for zero. Neste ciclo, adiciona sucessivamente cada um dos algarismos do número à variável `soma`. Para realizar esta tarefa o programa tem que obter cada um dos dígitos do número. Notemos que o dígito das unidades corresponde ao resto da divisão inteira do número por 10 (`num % 10`). Após obter o algarismo das unidades, o programa tem que “remover” este algarismo do número, o que pode

ser feito através da divisão inteira do número por 10 (`num // 10`). Depois do ciclo, o programa escreve o valor da variável `soma`.

O seguinte programa realiza a tarefa desejada:

```
num = eval(input('Escreva um inteiro positivo\n? '))
soma = 0

while num > 0:
    digito = num % 10  # obtém o algarismo das unidades
    num = num // 10    # remove o algarismo das unidades
    soma = soma + digito

print('Soma dos dígitos:', soma)
```

Este programa contém anotações, comentários, que explicam ao seu leitor algumas das instruções do programa. *Comentários* são frases em linguagem natural que aumentam a facilidade de leitura do programa, explicando o significado das variáveis, os objectivos de zonas do programa, certos aspectos do algoritmo, etc. Os comentários podem também ser expressões matemáticas provendo propriedades sobre o programa. Em Python, um comentário é qualquer linha, ou parte de linha, que se encontra após o símbolo “#”. Tal como as linhas em branco, os comentários são ignorados pelo computador.

2.9 Notas finais

Começámos por apresentar alguns tipos existente em Python, os tipos inteiro, real, lógico, e as cadeias de caracteres. Vimos que um tipo corresponde a um conjunto de valores, juntamente com um conjunto de operações aplicáveis a esses valores. Definimos alguns dos componentes de um programa, nomeadamente, as expressões e algumas instruções elementares. As expressões são entidades computacionais que têm um valor, ao calcular o valor de uma expressão dizemos que a expressão foi avaliada. As instruções correspondem a indicações fornecidas ao computador para efectuar certas acções. Ao contrário das expressões, as instruções não têm um valor mas produzem um efeito. Quando o computador efectua as acções correspondentes a uma instrução, diz-se que a instrução foi executada.

Neste capítulo, apresentámos a estrutura de um programa em Python. Um programa é constituído, opcionalmente por uma sequência de definições (que ainda não foram abordadas), seguido por uma sequência de instruções. Estudámos três instruções em Python, a instrução de atribuição (que permite a atribuição de um valor a um nome), a instrução `if` (que permite a selecção de grupos de instruções para serem executadas) e a instrução `while` (que permite a execução repetitiva de um conjunto de instruções). Apresentámos também duas operações para efectuarem a entrada e saída de dados.

2.10 Exercícios

1. Escreva um programa em Python que pede ao utilizador que lhe forneça um inteiro correspondente a um número de segundos e que calcula o número de dias correspondentes a esse número de segundos. O número de dias é fornecido como um real. O programa termina quando o utilizador fornece um número negativo. O seu programa deve possibilitar a seguinte interacção:

```
Escreva um número de segundos
(um número negativo para terminar)
? 45
O número de dias correspondente é  0.0005208333333333333
Escreva um número de segundos
(um número negativo para terminar)
? 6654441
O número de dias correspondente é  77.01899305555555
Escreva um número de segundos
(um número negativo para terminar)
? -1
>>>
```

2. Escreva um programa em Python que lê três números e que diz qual o maior dos números lidos.
3. A conversão de temperatura em graus Farenheit (F) para graus centígrados

(C) é dada através da expressão

$$C = \frac{5}{9} \cdot (F - 32).$$

Escreva um programa em Python que produz uma tabela com as temperaturas em graus centígrados, equivalentes às temperaturas em graus Farenheit entre $-40^{\circ} F$ e $120^{\circ} F$.

4. Escreva um programa em Python que lê uma sequência de dígitos, sendo cada um dos dígitos fornecido numa linha separada, e calcula o número inteiro composto por esses dígitos, pela ordem fornecida. Para terminar a sequência de dígitos é fornecido ao programa o inteiro -1 . Por exemplo, lendo os dígitos 1 5 4 5 8, o programa calcula o número inteiro 15458.
5. Escreva um programa em Python que lê um número inteiro positivo e calcula a soma dos seus dígitos pares. Por exemplo,

```
Escreva um inteiro positivo
? 234567
Soma dos dígitos pares: 12
```

6. Escreva um programa em Python que lê um número inteiro positivo e produz o número correspondente a inverter a ordem dos seus dígitos. Por exemplo,

```
Escreva um inteiro positivo
? 7633256
O número invertido é 6523367
```

7. Escreva um programa em Python que calcula o valor da série.

$$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

para um dado valor de x e de n . O seu programa deve ter em atenção que cada termo da série pode ser obtido a partir do anterior, multiplicando-o por x/n . O seu programa deve permitir a interacção:

```
Qual o valor de x?
> 2
```

2.10. EXERCÍCIOS

71

Qual o valor de n?

> 3

O valor da soma é 6.333333333333333

Capítulo 3

Funções

'Can you do Addition?' the White Queen asked. 'What's one and one?'

'I don't know,' said Alice. 'I lost count.'

'She ca'n't do Addition,' the Red Queen interrupted. 'Can you do Subtraction? Take nine from eight.'

'Nine from eight I ca'n't, you know,' Alice replied very readily: 'but —'

'She ca'n't do Subtraction,' said the White Queen. 'Can you do Division? Divide a loaf by a knife — what's the answer to that?'

Lewis Carroll, *Alice's Adventures in Wonderland*

No capítulo anterior considerámos apenas operações embutidas, as operações que já estão incluídas no Python. Como é evidente, não é possível que uma linguagem de programação forneça todas as operações que são necessárias para o desenvolvimento de uma certa aplicação. Por isso, durante o desenvolvimento de um programa, é necessário utilizar operações que não estão previamente definidas na linguagem de programação utilizada. As linguagens de programação fornecem mecanismos para a criação de novas operações e para a sua utilização num programa.

A possibilidade de agrupar informação e de tratar o grupo de informação como um todo, dando-lhe um nome que passará então a identificar o grupo, é parte fundamental da aquisição de conhecimento. Quando mais tarde se pretender utilizar a informação que foi agrupada e nomeada, ela poderá ser referida pelo

seu nome, sem que seja preciso enumerar exaustivamente os pedaços individuais de informação que a constituem. Em programação, este aspecto está ligado aos conceitos de *função*, *procedimento* e *subprograma*. O Python utiliza a designação “função”.

Antes de abordar a definição de funções em Python, recordemos o modo de definir e utilizar funções em matemática. Uma *função* (de uma variável) é um conjunto de pares ordenados que não contém dois pares distintos com o mesmo primeiro elemento. Ao conjunto de todos os primeiros elementos dos pares dá-se o nome de *domínio* da função, e ao conjunto de todos os segundos elementos dos pares dá-se o nome de *contradomínio* da função. Por exemplo, o conjunto $\{(1,3), (2,4), (3,5)\}$ corresponde a uma função cujo domínio é $\{1, 2, 3\}$ e cujo contradomínio é $\{3, 4, 5\}$.

A definição de uma função listando todos os seus elementos corresponde à *definição por extensão* (ou enumeração). Normalmente, as funções interessantes têm um número infinito de elementos, pelo que não é possível defini-las por extensão. Neste caso é necessário definir a função por *compreensão* (ou abstracção), apresentando uma propriedade comum aos seus elementos. Um dos modos de definir uma função por compreensão corresponde a descrever o processo de cálculo dos segundos elementos dos pares a partir dos primeiros elementos dos pares, fornecendo uma expressão designatória em que a variável que nela aparece pertence ao domínio da função. Uma *expressão designatória* é uma expressão que se transforma numa designação quando a variável que nela aparece é substituída por uma constante.

Para calcular o valor da função para um dado elemento do seu domínio, basta substituir o valor deste elemento na expressão designatória que define a função. Por exemplo, definindo a função natural de variável natural, f , pela expressão $f(x) = x * x$, estamos a definir o seguinte conjunto (infinito) de pares ordenados $\{(1,1), (2,4), (3,9), \dots\}$.

Note-se que $f(y) = y * y$ e $f(z) = z * z$ definem a mesma função (o mesmo conjunto de pares ordenados) que $f(x) = x * x$. As variáveis, tais como x , y e z nas expressões anteriores, que ao serem substituídas em todos os lugares que ocupam numa expressão dão origem a uma expressão equivalente, chamam-se *variáveis mudas*. Com a expressão $f(x) = x * x$, estamos a definir uma função cujo nome é f ; os elementos do domínio desta função (ou *argumentos* da função) são designados pelo nome x , e a regra para calcular o valor da função para um

dado elemento do seu domínio corresponde a multiplicar esse elemento por si próprio.

Dada uma função, designa-se por *conjunto de partida* o conjunto a que pertencem os elementos do seu domínio, e por *conjunto de chegada* o conjunto a que pertencem os elementos do seu contradomínio. Dada uma função f e um elemento x pertencente ao seu conjunto de partida, se o par (x, y) pertence à função f , diz-se que o *valor* de $f(x)$ é y . É frequente que alguns dos elementos tanto do conjunto de chegada como do conjunto de partida da função não apareçam no conjunto de pares que correspondem à função. Se existe um valor z , pertencente ao conjunto de partida da função, para o qual não existe nenhum par cujo primeiro elemento é z , diz-se que a função é *indefinida* para z .

Note-se que a utilização de funções tem dois aspectos distintos: a definição da função e a aplicação da função.

1. A *definição da função* é feita fornecendo um nome (ou uma designação) para a função, uma indicação das suas variáveis (ou argumentos) e a indicação de um processo de cálculo para os valores da função, por exemplo, $f(x) = x * x$.
2. A *aplicação da função* é feita fornecendo o nome da função e um elemento do seu domínio para o qual se pretende calcular o valor, por exemplo, $f(5)$.

Analogamente ao processo de definição de funções em Matemática, em Python, o processo de utilização de funções comprehende dois aspectos distintos: a *definição da função*, que, de um modo semelhante ao que se faz com a definição de funções em Matemática, é feita fornecendo o nome da função, os argumentos da função e um processo de cálculo para os valores da função (processo esse que é descrito por um algoritmo), e a *aplicação da função* a um valor, ou valores, do(s) seu(s) argumento(s). Esta aplicação corresponde à execução do algoritmo associado à função para valores particulares dos seus argumentos e em programação é vulgarmente designada por *chamada à função*.

3.1 Definição de funções em Python

Para definir funções em Python, é necessário indicar o nome da função, os seus argumentos (designados por *parâmetros formais*) e o processo de cálculo

(algoritmo) dos valores da função (designado por *corpo da função*). Em notação BNF, uma função em Python é definida do seguinte modo¹:

```

⟨definição de função⟩ ::= def ⟨nome⟩ ⟨parâmetros formais⟩ : [CR]
    [TAB+] ⟨corpo⟩ [TAB-]

⟨parâmetros formais⟩ ::= ⟨nada⟩ | ⟨nomes⟩

⟨nomes⟩ ::= ⟨nome⟩ |
    ⟨nome⟩, ⟨nomes⟩

⟨nada⟩ ::=

⟨corpo⟩ ::= ⟨definição de função⟩* ⟨instruções em função⟩

⟨instruções em função⟩ ::= ⟨instrução em função⟩ [CR] |
    ⟨instrução em função⟩ [CR] ⟨instruções em função⟩

⟨instrução em função⟩ ::= ⟨instrução⟩ |
    ⟨instrução return⟩

⟨instrução return⟩ ::= return |
    return ⟨expressão⟩

```

Nestas expressões, o símbolo não terminal ⟨parâmetros formais⟩, correspondente a zero ou mais nomes, especifica os parâmetros da função e o símbolo não terminal ⟨corpo⟩ especifica o algoritmo para o cálculo do valor da função. No ⟨corpo⟩ de uma função pode ser utilizada uma instrução adicional, a instrução `return`. Para já, não vamos considerar a possibilidade de utilizar a ⟨definição de função⟩ dentro do corpo de uma função.

Por exemplo, em Python, a função, `quadrado`, para calcular o quadrado de um número arbitrário, é definida do seguinte modo:

```
def quadrado(x):
    return x * x
```

Nesta definição:

1. O nome `x` representa o argumento da função, o qual é designado por *parâmetro formal*. O parâmetro formal vai indicar o nome pelo qual o argumento da função é representado dentro do corpo da função;
2. A instrução `return x * x` corresponde ao *corpo da função*.

¹A ⟨definição de função⟩ corresponde em Python a uma ⟨definição⟩.

A semântica da definição de uma função é a seguinte: ao encontrar a definição de uma função, o Python cria um nome, correspondente ao nome da função, e associa esse nome com um processo de cálculo para os argumentos da função. Ao definir uma função, o Python adiciona ao ambiente o nome correspondente ao nome da função, associando-o ao algoritmo para calcular o valor da função.

Consideremos a seguinte interacção:

```
>>> quadrado
NameError: name 'quadrado' is not defined
>>> def quadrado(x):
...     return x * x
...
>>> quadrado
<function quadrado at 0x10f4618>
```

Na primeira linha, ao fornecermos ao Python o nome `quadrado` este indica-nos que não conhece o nome. Nas três linhas seguintes, definimos a função `quadrado`. Recorde-se que as linhas que apresentam ... como carácter de pronto correspondem a linhas de continuação da definição da função e uma linha em branco indica ao Python que terminámos a sua definição. Quando, na linha seguinte fornecemos ao Python o nome `quadrado` este indica-nos que `quadrado` corresponde a uma função e qual a posição de memória em que esta se encontra armazenada.

3.2 Aplicação de funções em Python

Uma vez definida, uma função pode ser usada do mesmo modo que as funções embutidas, fornecendo ao Python, numa expressão, o nome da função seguido do número apropriado de argumentos (os *parâmetros concretos*). A partir do momento em que uma função é definida, passa a existir em Python uma nova expressão, correspondente a uma aplicação de função (ver a definição apresentada na página 35), a qual é definida do seguinte modo:

```
⟨aplicação de função⟩ ::= ⟨nome⟩(⟨parâmetros concretos⟩)
⟨parâmetros concretos⟩ ::= ⟨nada⟩ | ⟨expressões⟩
```

```
 $\langle \text{expressões} \rangle ::= \langle \text{expressão} \rangle \mid$ 
 $\quad \langle \text{expressão} \rangle, \langle \text{expressões} \rangle$ 
```

Nesta definição, $\langle \text{nome} \rangle$ corresponde ao nome da função e o número de expressões em $\langle \text{parâmetros concretos} \rangle$ é igual ao número de parâmetros formais da função. Este último aspecto não pode ser definido em notação BNF. Quando uma função é aplicada é comum dizer-se que a função foi *chamada*.

Usando a função `quadrado` podemos originar a interacção:

```
>>> quadrado(7)
49
>>> quadrado(2, 3)
TypeError: quadrado() takes exactly 1 argument (2 given)
```

A interacção anterior mostra que se fornecermos à função `quadrado` o argumento 7, o valor da função é 49. Se fornecermos dois argumentos à função, o Python gera um erro indicando-nos que esta função recebe apenas um argumento.

Consideremos, de um modo mais detalhado, os passos seguidos pelo Python ao calcular o valor de uma função. Para calcular o valor da aplicação de uma função, o Python utiliza a seguinte regra:

1. Avalia os parâmetros concretos (por qualquer ordem).
2. Associa os parâmetros formais da função com os valores dos parâmetros concretos calculados no passo anterior. Esta associação é feita com base na posição dos parâmetros, isto é, o primeiro parâmetro concreto é associado ao primeiro parâmetro formal, e assim sucessivamente.
3. Cria um novo ambiente, o *ambiente local* à função, definido pela associação entre os parâmetros formais e os parâmetros concretos. No ambiente local executa as instruções correspondentes ao corpo da função. O ambiente local apenas existe enquanto a função estiver a ser executada e é apagado pelo Python quando termina a execução da função.

Com a definição de funções, surge uma nova instrução em Python, a instrução `return`, a qual apenas pode ser utilizada dentro do corpo de uma função, e cuja sintaxe foi definida na página 76. Ao encontrar a instrução `return`, o Python calcula o valor da expressão associada a esta instrução (se ela existir),

e termina a execução do corpo da função, sendo o valor da função o valor desta expressão. Este valor é normalmente designado pelo *valor devolvido pela função*. Se a instrução `return` não estiver associada a uma expressão, então a função não devolve nenhum valor (note-se que já vimos, na página 58, que podem existir funções que não devolvem nenhum valor). Por outro lado, se todas as instruções correspondentes ao corpo da função forem executadas sem encontrar uma instrução `return`, a execução do corpo da função termina e a função não devolve nenhum valor.

A avaliação de uma função introduz o conceito de *ambiente local*. Para distinguir um ambiente local do ambiente que considerámos no Capítulo 2, este é designado por ambiente global. O *ambiente global* está associado a todas as operações efectuadas directamente após o carácter de pronto. Todos os nomes que são definidos pela operação de atribuição ou através de definições, directamente executadas após o carácter de pronto, pertencem ao ambiente global. O ambiente global é criado quando uma sessão com o Python é iniciada e existe enquanto essa sessão durar. A partir de agora, todos os ambientes que apresentamos contém, para além da associação entre nomes e valores, uma designação do tipo de ambiente a que correspondem, o ambiente global, ou um ambiente local a uma função.

Consideremos agora a seguinte interacção, efectuada depois da definição da função `quadrado`:

```
>>> x = 5  
>>> y = 7  
>>> quadrado(y)
```

49

As duas primeiras linhas dão origem ao ambiente apresentado na Figura 3.1 (neste ambiente, o nome `quadrado` já era conhecido, estando associado com o processo de cálculo desta função). Ao encontrar a expressão `quadrado(y)` na terceira linha da nossa interacção, o Python calcula o valor de `y` e cria um novo ambiente, representado na Figura 3.2 com o título *Ambiente local a quadrado*, no qual a variável `x`, o parâmetro formal de `quadrado`, está associada ao valor do parâmetro concreto 7. A seta que na Figura 3.2 liga o ambiente local ao ambiente global indica, entre outras coisas, qual o ambiente que será considerado pelo Python quando o ambiente local desaparecer. É então executado o corpo

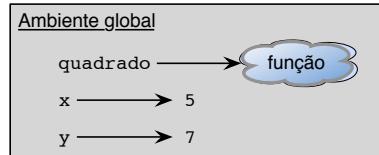


Figura 3.1: Ambiente global com os nomes `quadrado`, `x` e `y`.

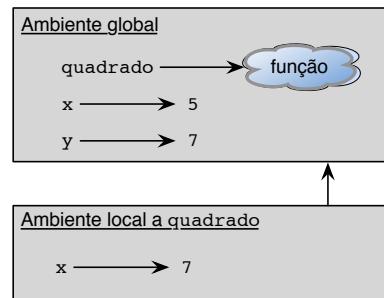


Figura 3.2: Ambiente criado durante a avaliação de `quadrado(y)`.

da função `quadrado`, o qual apenas contém uma instrução `return x * x`. A semântica da instrução `return` leva à avaliação da expressão `x * x`, cujo valor é 49. Este valor corresponde ao valor da aplicação função `quadrado(y)`. A instrução `return` tem também como efeito fazer desaparecer o ambiente que foi criado com a avaliação da função, pelo que quando o valor 49 é devolvido o Python volta a ter o ambiente apresentado na Figura 3.1.

Podemos então dizer que quando uma função é chamada, é criado um *ambiente local*, o qual corresponde a uma associação entre os parâmetros formais e os parâmetros concretos. Este ambiente local desaparece no momento em que termina a avaliação da função que deu origem à sua criação.

3.3 Abstracção procedural

A criação de funções corresponde a uma forma de abstracção, a qual consiste em nomear uma sequência de operações que permitem atingir um determinado objectivo (a definição da função). Uma vez definida uma função, para a aplicar a determinados parâmetros concretos basta escrever o seu nome seguido desses

parâmetros (a aplicação da função), tal como se de uma função embutida se tratasse. A introdução dos parâmetros formais permite que uma função represente um número potencialmente infinito de operações (idênticas), dependendo apenas dos parâmetros concretos que lhe são fornecidos.

A *abstracção procedural* consiste em dar um nome à sequência de acções que serve para atingir um objectivo (e consequentemente abstrair do modo como as funções realizam as suas tarefas), e em utilizar esse nome, sempre que desejarmos atingir esse objectivo sem termos de considerar explicitamente cada uma das acções individuais que a constituem.

A associação de um nome a uma sequência de acções, a qual é referida pelo seu nome em vez de enumerar todas as instruções que a constituem, é por nós utilizada diariamente. Suponhamos que desejamos que alguém abra uma determinada porta. Ao dizermos “abra a porta”, estamos implicitamente a referir uma determinada sequência de instruções: “mova-se em direcção à porta, determine para onde a porta se abre, rode o manípulo, empurre ou puxe a porta, etc.”. Sem a capacidade de nomear sequências de instruções, seria impossível comunicarmos, uma vez que teríamos de explicitar tudo ao mais ínfimo pormenor.

A *abstracção procedural* consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam. Ou seja, a separação do “como” de “o que”.

A abstracção procedural permite-nos desenvolver programas complexos, manipulando entidades complexas, sem nos perdermos em pormenores em relação à especificação minuciosa do algoritmo que manipula essas entidades. Usando a abstracção procedural consideramos funções como caixas pretas – sabemos *o que elas fazem*, mas não nos interessa saber *como o fazem*. Para ilustrar este aspecto, recordemos que a função `quadrado` foi definida do seguinte modo:

```
def quadrado(x):
    return x * x
```

No entanto, para a finalidade da sua utilização, os pormenores da realização da função `quadrado` são irrelevantes. Poderíamos ter definido a função `quadrado` do seguinte modo (esta função calcula o quadrado de um número somando o número consigo próprio um número de vezes igual ao número):



Figura 3.3: O conceito de caixa preta.

```

def quadrado(x):
    res = 0
    cont = 0
    while cont != x:
        res = res + x
        cont = cont + 1
    return res
  
```

e obtínhamos exactamente o mesmo comportamento. O que nos interessa saber, quando usamos a função `quadrado`, é que esta recebe como argumento um número e produz o valor do quadrado desse número — ou seja, interessa-nos saber *o que* esta função faz, *como* o faz não nos interessa (Figura 3.3). É a isto que corresponde a abstracção procedimental.

A abstracção procedural permite-nos considerar as funções como ferramentas que podem ser utilizadas na construção de outras funções. Ao considerarmos uma destas ferramentas, sabemos quais os argumentos que recebe e qual o resultado produzido, mas não sabemos (ou não nos preocupamos em saber) como ela executa a sua tarefa — apenas em que consiste essa tarefa.

3.4 Exemplos

3.4.1 Nota final de uma disciplina

Suponhamos que numa determinada disciplina existiam três provas de avaliação, e que a nota da disciplina era calculada como a média aritmética das três classificações obtidas.

Pretendemos escrever uma função que recebe três argumentos, correspondentes às notas das provas realizadas, e que devolve a classificação final obtida na disciplina. Se a média das provas realizadas for inferior a 9.5 a classificação será “Reprovado”, em caso contrário a classificação será o número inteiro que

corresponde ao arredondamento da média da disciplina.

A seguinte função realiza o cálculo da nota final da disciplina:

```
def calcula_nota(n1, n2, n3):
    media = (n1 + n2 + n3) / 3
    if media < 9.5:
        return 'Reprovado'
    else:
        return round(media)
```

Com esta função podemos gerar a interacção:

```
>>> calcula_nota(14, 16, 13)
14
>>> calcula_nota(3, 9, 12)
'Reprovado'
```

Recorrendo a esta função, podemos escrever o seguinte programa:

```
def calcula_nota(n1, n2, n3):
    media = (n1 + n2 + n3) / 3
    if media < 9.5:
        return 'Reprovado'
    else:
        return round(media)

soma = 0
cont = 0
num = input('Qual o número do aluno\n(zero para terminar)\n? ')
while num != '0':
    n_1 = eval(input('Introduza a primeira nota\n? '))
    n_2 = eval(input('Introduza a segunda nota\n? '))
    n_3 = eval(input('Introduza a terceira nota\n? '))
    nota = calcula_nota(n_1, n_2, n_3)
    print('A nota obtida pelo aluno', num, 'é', nota)
    if nota != 'Reprovado' :
        soma = soma + nota
```

```
    cont = cont + 1
    num = input('Qual o número do aluno\n(zero para terminar)\n? ')
if cont > 0:
    print('A média dos alunos aprovados é', soma/cont)
else:
    print('Não existem alunos aprovados')
```

Recorde-se da página 59, que um programa é constituído por zero ou mais definições seguidas de uma ou mais instruções. No nosso caso, o programa é constituído pela definição da função `calcula_nota`, seguida das instruções que, através de um ciclo, calculam as notas de vários alunos. O programa vai acumulando a somas das notas correspondentes às aprovações e do número de alunos aprovados, escrevendo no final a média das notas obtidas. Com este programa podemos obter a interacção:

```
Qual o número do aluno
(zero para terminar)
? 12345
Introduza a primeira nota
? 12
Introduza a segunda nota
? 7
Introduza a terceira nota
? 14
A nota obtida pelo aluno 12345 é 11
Qual o número do aluno
(zero para terminar)
? 12346
Introduza a primeira nota
? 17
Introduza a segunda nota
? 20
Introduza a terceira nota
? 16
A nota obtida pelo aluno 12346 é 18
Qual o número do aluno
(zero para terminar)
```

```
? 12347
Introduza a primeira nota
? 9
Introduza a segunda nota
? 11
Introduza a terceira nota
? 6
A nota obtida pelo aluno 12347 é Reprovado
Qual o número do aluno
(zero para terminar)
? 0
A média dos alunos aprovados é 14.5
>>>
```

3.4.2 Potência

Uma operação comum em Matemática é o cálculo da potência de um número. Dado um número qualquer, x (a base), e um inteiro não negativo, n (o expoente), define-se potência da base x ao expoente n , escrito x^n , como sendo o produto de x por si próprio n vezes. Por convenção, $x^0 = 1$.

Com base nesta definição podemos escrever a seguinte função em Python:

```
def potencia(x, n):
    res = 1
    while n != 0:
        res = res * x
        n = n - 1
    return res
```

Com esta função, podemos gerar a seguinte interacção:

```
>>> potencia(3, 2)
9
>>> potencia(2, 8)
256
>>> potencia(3, 100)
515377520732011331036461129765621272702107522001
```

3.4.3 Factorial

Em matemática, o *factorial*² de um inteiro não negativo n , representado por $n!$, é o produto de todos os inteiros positivos menores ou iguais a n . Por exemplo, $5! = 5 * 4 * 3 * 2 * 1 = 120$. Por convenção, o valor de $0!$ é 1.

Com base na definição anterior, podemos escrever a seguinte função em Python para calcular o factorial de um inteiro::

```
def factorial(n):
    fact = 1
    while n != 0:
        fact = fact * n
        n = n - 1
    return fact
```

Com esta função obtemos a interacção:

```
>>> factorial(3)
6
>>> factorial(21)
51090942171709440000
```

3.4.4 Máximo divisor comum

O máximo divisor comum entre dois inteiros m e n diferentes de zero, escrito $mdc(m, n)$, é o maior inteiro positivo p tal que tanto m como n são divisíveis por p . Esta descrição define uma função matemática no sentido em que podemos reconhecer quando um número é o máximo divisor comum de dois inteiros, mas, dados dois inteiros, esta definição não nos indica como calcular o seu máximo divisor comum. Este é um dos aspectos em que as funções em informática diferem das funções matemáticas. Embora tanto as funções em informática como as funções matemáticas especifiquem um valor que é determinado por um

²Segundo [Biggs, 1979], a função factorial já era conhecida por matemáticos indianos no início do Século XII, tendo a notação $n!$ sido introduzida pelo matemático francês Christian Kramp (1760–1826) em 1808.

ou mais parâmetros, as funções em informática devem ser *eficazes*, no sentido de que têm de especificar, de um modo claro, como calcular os valores.

Um dos primeiros algoritmos a ser formalizado corresponde ao cálculo do máximo divisor comum entre dois inteiros e foi enunciado, cerca de 300 anos a.C., por Euclides no seu Livro VII. A descrição originalmente apresentada por Euclides é complicada (foi enunciada há 2300 anos) e pode ser hoje expressa de um modo muito mais simples:

1. O máximo divisor comum entre um número e zero é o próprio número.
2. Quando dividimos um número por um menor, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor.

Com base na descrição anterior, podemos enunciar o seguinte algoritmo para calcular o máximo divisor comum entre m e n , $\text{mdc}(m, n)$, utilizando o algoritmo de Euclides:

1. Se $n = 0$, então o máximo divisor comum corresponde a m . Note-se que isto corresponde ao facto de o máximo divisor comum entre um número e zero ser o próprio número.
2. Em caso contrário, calculamos o resto da divisão entre m e n , o qual utilizando a função embutida do Python (ver Tabela 2.2) será dado por $r = m \% n$ e repetimos o processo com m igual ao valor de n (o divisor) e n igual ao valor de r (o resto da divisão).

Os passos descritos serão repetidos enquanto n não for zero. Na Tabela 3.1 apresentamos os passos seguidos no cálculo do máximo divisor comum entre 24 e 16.

Podemos então escrever a seguinte função em Python para calcular o máximo divisor comum entre os inteiros m e n :

```
def mdc(m, n):
    while m % n != 0:
        m, n = n, m % n
    return n
```

m	n	$m \% n$
24	16	8
16	8	0
8	0	8

Tabela 3.1: Passos no cálculo do máximo divisor comum entre 24 e 16.

Uma vez definida a função `mdc`, podemos gerar a seguinte interacção:

```
>>> mdc(24, 16)
8
>>> mdc(16, 24)
8
>>> mdc(35, 14)
7
```

3.4.5 Raiz quadrada

Vamos supor que queríamos escrever uma função em Python para calcular a raiz quadrada de um número positivo. Como poderíamos calcular o valor de \sqrt{x} para um dado x ?

Comecemos por considerar a definição matemática de raiz quadrada: \sqrt{x} é o y tal que $y^2 = x$. Esta definição, típica da Matemática, diz-nos o que é uma raiz quadrada, mas não nos diz nada sobre o processo de cálculo de uma raiz quadrada. Com esta definição, podemos determinar se um número corresponde à raiz quadrada de outro, podemos provar propriedades sobre as raízes quadradas mas não temos qualquer pista sobre o modo de calcular raízes quadradas.

Iremos utilizar um algoritmo para o cálculo de raízes quadradas, que foi introduzido no início da nossa era pelo matemático grego Heron de Alexandria (c. 10–75 d.C.)³. Este algoritmo, corresponde a um método de iterações sucessivas, em que a partir de um “palpite” inicial para o valor da raiz quadrada de x , digamos p_0 , nos permite melhorar sucessivamente o valor aproximado de \sqrt{x} .

Em cada iteração, partindo de um valor aproximado, p_i , para a raiz quadrada

³Ver [Kline, 1972].

Número da tentativa	Aproximação para $\sqrt{2}$	Nova aproximação
0	1	$\frac{1+\frac{2}{1}}{2} = 1.5$
1	1.5	$\frac{1.5+\frac{2}{1.5}}{2} = 1.4167$
2	1.4167	$\frac{1.4167+\frac{2}{1.4167}}{2} = 1.4142$
3	1.4142	...

Tabela 3.2: Sucessivas aproximações para $\sqrt{2}$.

de x podemos calcular uma aproximação melhor, p_{i+1} , para a raiz quadrada de x , através da seguinte fórmula:

$$p_{i+1} = \frac{p_i + \frac{x}{p_i}}{2}.$$

Suponhamos, por exemplo, que desejamos calcular $\sqrt{2}$. Sabemos que $\sqrt{2}$ é *um vírgula qualquer coisa*. Seja então 1 o nosso palpite inicial, $p_0 = 1$. A Tabela 3.2 mostra-nos a evolução das primeiras aproximações calculadas para $\sqrt{2}$.

Um aspecto que distingue este exemplo dos apresentados nas secções anteriores é o facto de o cálculo da raiz quadrada ser realizado por um método aproximado. Isto significa que não vamos obter o valor exacto da raiz quadrada⁴, mas sim uma aproximação que seja suficientemente boa para o fim em vista.

Podemos escrever a seguinte função em Python que corresponde ao algoritmo apresentado:

```
def calcula_raiz(x, palpito):
    while not bom_palpite(x, palpito):
        palpito = novo_palpite(x, palpito)
    return palpito
```

Esta função é bastante simples: fornecendo-lhe um valor para o qual calcular a raiz quadrada (**x**) e um palpite (**palpito**), enquanto estivermos perante um palpite que não é bom, deveremos calcular um novo palpite; se o palpite for um bom palpite, esse palpite será o valor da raiz quadrada.

Esta função pressupõe que existem outras funções para decidir se um dado pal-

⁴Na realidade, este é um número irracional.

pite para o valor da raiz quadrada de x é bom (a função `bom_palpite`) e para calcular um novo palpite (a função `novo_palpite`). A isto chama-se *pensamento positivo!* Note-se que esta técnica de pensamento positivo nos permitiu escrever uma versão da função `calcula_raiz` em que os problemas principais estão identificados, versão essa que é feita em termos de outras funções. Para podermos utilizar a função `calcula_raiz` teremos de desenvolver funções para decidir se um dado palpite é bom e para calcular um novo palpite.

Esta abordagem do pensamento positivo corresponde a um método largamente difundido para limitar a complexidade de um problema, a que se dá o nome de *abordagem do topo para a base*⁵. Ao desenvolver a solução de um problema utilizando a abordagem do topo para a base, começamos por dividir esse problema noutras mais simples. Cada um destes problemas recebe um nome, e é então desenvolvida uma primeira aproximação da solução em termos dos principais subproblemas identificados e das relações entre eles. Seguidamente, aborda-se a solução de cada um destes problemas utilizando o mesmo método. Este processo termina quando se encontram subproblemas para os quais a solução é trivial.

O cálculo do novo palpite é bastante fácil, bastando recorrer à fórmula apresentada pelo algoritmo, a qual é traduzida pela função:

```
def novo_palpite(x, palpite):
    return (palpite + x / palpite) / 2
```

Resta-nos decidir quando estamos perante um bom palpite. É evidente que a satisfação com dado palpite vai depender do objectivo para o qual estamos a calcular a raiz quadrada: se estivermos a fazer cálculos que exijam alta precisão, teremos de que exigir uma maior precisão do que a que consideraríamos em cálculos grosseiros.

A definição matemática de raiz quadrada, \sqrt{x} é o y tal que $y^2 = x$, dá-nos uma boa pista para determinar se um palpite p_i é ou não bom. Para que o palpite p_i seja um bom palpite, os valores $(p_i)^2$ e x deverão ser *suficientemente* próximos. Uma das medidas utilizadas em Matemática para decidir se dois números são “quase iguais”, chamada *erro absoluto*, corresponde a decidir se o valor absoluto da sua diferença é menor do que um certo limiar⁶. Utilizando

⁵Do inglês, “top-down design”.

⁶Uma outra alternativa corresponde a considerar o *erro relativo*, o quociente entre o erro absoluto e o valor correcto, $| (p_i)^2 - x | / x$.

esta medida, diremos que p_i é uma boa aproximação de \sqrt{x} se $|(p_i)^2 - x| < \delta$, em que δ é um valor suficientemente pequeno. Supondo que δ corresponde ao nome `delta`, o qual define o limiar de aproximação exigido pela nossa aplicação, podemos escrever a função:

```
def bom_palpite(x, palpate):
    return abs(x - palpate * palpate) < delta
```

Note-se que poderíamos ser tentados a escrever a seguinte função para decidir se uma dada aproximação é um bom palpite:

```
def bom_palpite(x, palpate):
    if abs(x - palpate * palpate) < delta:
        return True
    else:
        return False
```

Esta função avalia a expressão `abs(x - palpate * palpate) < delta`, devolvendo `True` se esta for verdadeira e `False` se a expressão for falsa. Note-se no entanto que o valor devolvido pela função é exactamente o valor da expressão `abs(x - palpate * palpate) < delta`, daí a simplificação que introduzimos na nossa primeira função.

Com as funções apresentadas, e definindo `delta` como sendo 0.0001, podemos gerar a interacção

```
>>> delta = 0.0001
>>> calcula_raiz(2, 1)
1.4142156862745097
```

Convém notar que o cálculo da raiz quadrada através da função `calcula_raiz` não é natural, pois obriga-nos a fornecer um palpite (o que não acontece quando calculamos a raiz quadrada com uma calculadora). Sabendo que 1 pode ser o palpite inicial para o cálculo da raiz quadrada de qualquer número, e que apenas podemos calcular raízes de números não negativos, podemos finalmente escrever a função `raiz` que calcula a raiz quadrada de um número não negativo:

```
def raiz(x):
```

Nome	Situação correspondente ao erro
<code>AttributeError</code>	Referência a um atributo não existente num objecto.
<code>ImportError</code>	Importação de uma biblioteca não existente.
<code>IndexError</code>	Erro gerado pela referência a um índice fora da gama de um tuplo ou de uma lista.
<code>KeyError</code>	Referência a uma chave inexistente num dicionário.
<code>NameError</code>	Referência a um nome que não existe.
<code>SyntaxError</code>	Erro gerado quando uma das funções <code>eval</code> ou <code>input</code> encontram uma expressão com a sintaxe incorrecta.
<code>ValueError</code>	Erro gerado quando uma função recebe um argumento de tipo correcto mas cujo valor não é apropriado.
<code>ZeroDivisionError</code>	Erro gerado pela divisão por zero.

Tabela 3.3: Alguns dos identificadores de erros em Python.

```

delta = 0.001
if x >= 0:
    return calcula_raiz (x, 1)
else:
    raise ValueError ('raiz, argumento negativo')

```

Na função `raiz` usámos a instrução `raise` que força a geração de um erro de execução. A razão da nossa decisão de gerar um erro em lugar de recorrer à geração de uma mensagem através da função `print` resulta do facto que quando fornecemos à função `raiz` um argumento negativo não queremos que a execução desta função continue, alertando o utilizador que algo de errado aconteceu.

A instrução `raise` tem a seguinte sintaxe em notação BNF:

$\langle \text{instrução raise} \rangle ::= \text{raise } \langle \text{nome} \rangle \left(\langle \text{mensagem} \rangle \right)$

$\langle \text{mensagem} \rangle ::= \langle \text{cadeia de caracteres} \rangle$

O símbolo não terminal $\langle \text{nome} \rangle$ foi definido na página 76. A utilização de $\langle \text{nome} \rangle$ nesta instrução diz respeito a nomes que correspondem à identificação de vários tipos de erros que podem surgir num programa, alguns dos quais são apresentados na Tabela 3.3. Note-se que várias das situações correspondentes a erros apresentados na Tabela 3.3 ainda não foram apresentadas neste livro. A $\langle \text{mensagem} \rangle$ corresponde à mensagem que é mostrada pelo Python com a indicação do erro.

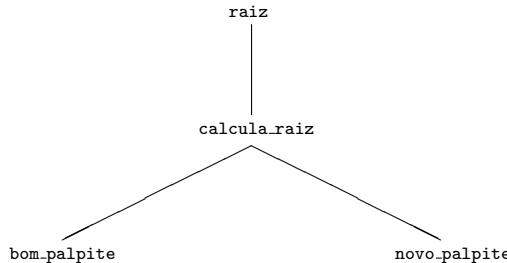


Figura 3.4: Funções associadas ao cálculo da raiz.

Assim, com a função `raiz`, podemos obter a seguinte interacção:

```

>>> raiz(2)
1.4142156862745097
>>> raiz(-1)
ValueError: raiz, argumento negativo
  
```

A função `raiz` é a nossa primeira função definida à custa de um certo número de outras funções. Cada uma destas funções aborda um problema específico: como determinar se um palpito é suficientemente bom; como calcular um novo palpito; etc. Podemos olhar para a função `raiz` como sendo definida através de um agrupamento de outras funções (Figura 3.4), o que corresponde à abstracção procedural.

Sob esta perspectiva, a tarefa de desenvolvimento de um programa corresponde a definir várias funções, cada uma resolvendo um dos subproblemas do problema a resolver, e “ligá-las entre si” de modo apropriado.

Antes de terminar esta secção convém discutir a relação entre funções matemáticas e as funções em Python que correspondem a essas funções. As funções que temos definido até aqui comportam-se basicamente como funções matemáticas: ao receberem um valor correspondente a um argumento do seu domínio, devolvem o valor correspondente da função. No entanto, as funções em Python devem estar associadas a um algoritmo que calcula o valor da função, ao passo que no caso das funções matemáticas basta definir o que elas são. Em resumo, a informática baseia-se em *conhecimento procedural*, ou seja *como* fazer as coisas, ao passo que a matemática se baseia em *conhecimento declarativo*, ou seja, *o que* as coisas são.

Número de termos	Aproximação a $\sin(1.57)$
1	1.57
2	$1.57 - \frac{1.57^3}{3!} = 0.92501$
3	$1.57 - \frac{1.57^3}{3!} + \frac{1.57^5}{5!} = 1.00450$
4	$1.57 - \frac{1.57^3}{3!} + \frac{1.57^5}{5!} - \frac{1.57^7}{7!} = 0.99983$

Tabela 3.4: Sucessivas aproximações ao cálculo de $\sin(1.57)$.

3.4.6 Seno

Apresentamos um procedimento para calcular o valor do *seno*. Este exemplo partilha com o cálculo da raiz o facto de ser realizado por um método aproximado.

O *seno* é uma função trigonométrica. Dado um triângulo rectângulo com um de seus ângulos internos igual a α , define-se $\sin(\alpha)$ como sendo a razão entre o cateto oposto e a hipotenusa deste triângulo. Para calcular o valor de *seno* podemos utilizar o desenvolvimento em série de Taylor o qual fornece o valor de $\sin(x)$ para um valor de x em radianos:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

A série anterior fornece-nos um processo de cálculo para o $\sin(x)$, mas, aparentemente, não nos serve de muito, uma vez que, para calcular o valor da função para um dado argumento, teremos de calcular a soma de um número infinito de termos. No entanto, podemos utilizar a fórmula anterior para calcular uma aproximação à função *seno*, considerando apenas um certo número de termos. Na Tabela 3.4 apresentamos alguns exemplos de aproximações ao cálculo de $\sin(1.57)$ (note-se que $1.57 \approx \pi/2$, cujo valor do *seno* é 1).

Usando uma técnica semelhante à da secção anterior, consideremos o seguinte algoritmo para calcular $\sin(x)$, o qual começa com uma aproximação ao valor do *seno* (correspondente ao primeiro termo da série):

1. Se o termo a adicionar for suficientemente pequeno, então a aproximação considerada será o valor da série.

2. Em caso contrário, calculamos uma aproximação melhor, por adição de mais um termo da série.

Com base no que dissemos, podemos esboçar uma função em Python para calcular o *seno*:

```
def sin(x):
    n = 0    # termo da série em consideração
    termo = calc_termo(x, n)
    seno = termo

    while not suf_pequeno(termo):
        n = n + 1
        termo = calc_termo(x, n)
        seno = seno + termo

    return seno
```

Esta função pressupõe que existem outras funções para decidir se um termo é suficientemente pequeno (a função `suf_pequeno`) e para calcular o termo da série que se encontra na posição `n` (a função `calc_termo`). Com base na discussão apresentada na secção anterior, a função `suf_pequeno` é definida trivialmente do seguinte modo:

```
def suf_pequeno(valor):
    return abs(valor) < delta

delta = 0.0001
```

Para calcular o termo na posição n da série, notemos que o factor $(-1)^n$, em cada um dos termos na série de Taylor, na realidade define qual o sinal do termo: o sinal é positivo se n for par, e é negativo em caso contrário. Por esta razão, a função `sinal` não é definida em termos de potências mas sim em termos da paridade de n .⁷ Na segunda linha da função `calc_termo` usamos o símbolo “\”, o *símbolo de continuação*, que indica ao Python que a linha que estamos a escrever continua na próxima linha. A utilização do símbolo de continuação não é obrigatória, servindo apenas para tornar o programa mais fácil de ler.

⁷A função `potencia` foi definida na Secção 3.4.2 e a função `factorial` na Secção 3.4.3.

```

def calc_termo(x, n):
    return sinal(n) * \
        potencia(x, 2 * n + 1) / factorial(2 * n + 1)

def sinal(n):
    if n % 2 == 0: # n é par
        return 1
    else:
        return -1

```

Com estas funções podemos gerar a interacção:

```

>>> sin(1.57)
0.9999996270418701

```

3.5 Estruturação de funções

Na Secção 3.1 estudámos o modo de definir funções em Python. Nesta secção, voltamos a considerar a definição de funções, apresentando o modo de criar uma estrutura nas funções e analisando as consequências da criação dessa estrutura.

Consideremos novamente a função `potencia` apresentada na Secção 3.4.2:

```

def potencia(x , n):
    res = 1
    while n != 0:
        res = res * x
        n = n - 1
    return res

```

É fácil constatar que esta função apenas produz resultados para valores do expoente que sejam inteiros positivos ou nulos. Se fornecermos à função um valor negativo para o expoente, o Python gera um ciclo infinito, pois a condição do ciclo `while`, `n != 0`, nunca é satisfeita. Isto corresponde a uma situação que queremos obviamente evitar⁸. Tendo em atenção que $x^{-n} = 1/x^n$, podemos

⁸A utilização da condição `n > 0` no ciclo `while` evitava o ciclo infinito, mas a função tinha sempre o valor 1 quando o expoente era negativo, o que ainda é pior, pois a função estaria a devolver um valor errado.

pensar na seguinte solução que lida tanto com o expoente positivo como com o expoente negativo:

```
def potencia(x, n) :
    res = 1
    if n >= 0:
        while n != 0:
            res = res * x
            n = n - 1
    else:
        while n != 0:
            res = res / x
            n = n + 1
    return res
```

Com esta nova função podemos gerar a interacção:

```
>>> potencia(3, 2)
9
>>> potencia(3, -2)
0.1111111111111111
>>> potencia(3, 0)
1
```

A desvantagem desta solução é a de exigir dois ciclos muito semelhantes, um que trata o caso do expoente positivo, multiplicando sucessivamente o resultado por x , e outro que trata o caso do expoente negativo, dividindo sucessivamente o resultado por x .

Podemos ainda pensar numa outra solução que recorre a uma função auxiliar (com o nome `potencia_aux`) para o cálculo do valor da potência, função essa que é sempre chamada com um expoente positivo:

```
def potencia(x, n):
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)
```

```
def potencia_aux(b, e):
    res = 1
    while e != 0:
        res = res * b
        e = e - 1
    return res
```

Com esta solução resolvemos o problema da existência dos dois ciclos semelhantes, mas potencialmente criámos um outro problema: nada impede que a função `potencia_aux` seja directamente utilizada para calcular a potência, eventualmente gerando um ciclo infinito se os seus argumentos não forem os correctos.

O Python apresenta uma alternativa para abordar problemas deste tipo, baseada no conceito de *estrutura de blocos*. A estrutura de blocos é muito importante em programação, existindo uma classe de linguagens de programação, chamadas *linguagens estruturadas em blocos*, que são baseadas na definição de blocos. Historicamente, a primeira linguagem desta classe foi o ALGOL, desenvolvida na década de 50, e muitas outras têm surgido, o Python é uma destas linguagens. A ideia subjacente à estrutura de blocos consiste em definir funções dentro das quais existem outras funções. Em Python, qualquer função pode ser considerada como um bloco, dentro da qual podem ser definidos outros blocos.

Nas linguagens estruturadas em blocos, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Esta regra permite a protecção efectiva da informação definida em cada bloco da utilização não autorizada por parte de outros blocos. Podemos pensar nos blocos como sendo egoístas, não permitindo o acesso à sua informação pelos blocos definidos fora deles.

A razão por que os blocos não podem usar a informação definida num bloco interior é que essa informação pode não estar pronta para ser utilizada até que algumas acções lhe sejam aplicadas ou pode exigir que certas condições sejam verificadas. No caso da função `potencia_aux`, estas condições correspondem ao facto de o expoente ter que ser positivo e a estrutura de blocos vai permitir especificar que a única função que pode usar a função `potencia_aux` é a função `potencia`, a qual garante que função `potencia_aux` só é usada com os argumentos correctos. O bloco onde essa informação é definida é o único que sabe

quais as condições de utilização dessa informação e, consequentemente, o único que pode garantir que essa sequência de acções é aplicada antes da informação ser usada.

Recorde-se da página 76 que uma função em Python foi definida do seguinte modo:

```
<definição de função> ::= def <nome> (<parâmetros formais>): CR
                           TAB <corpo> TAB-
<corpo> ::= <definição de função>* <instruções+>
```

Até agora, o corpo de todas as nossas funções tem sido apenas constituído por instruções e o componente opcional `<definição de função>` não tem existido. No entanto, são as definições de funções dentro do `<corpo>` que permitem a definição da estrutura de blocos num programa em Python: cada função definida no corpo de uma outra função corresponde a um bloco.

Recorrendo a esta definição, podemos escrever uma nova versão da função `potencia`:

```
def potencia(x , n):

    def potencia_aux(b, e):
        res = 1
        while e != 0:
            res = res * b
            e = e - 1
        return res

    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)
```

O corpo desta função contém a definição de uma função, `potencia_aux`, e uma instrução `if`. Com esta função, podemos gerar a seguinte interacção:

```
>>> potencia(3, 2)
```

```

def potencia(x, n):
    def potencia_aux(b, e):
        res = 1
        while e != 0:
            res = res * b
            e = e - 1
        return res
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)

```

Figura 3.5: Estrutura de blocos da função `potencia`.

```

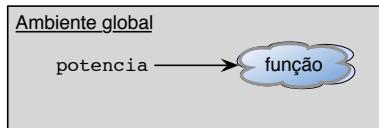
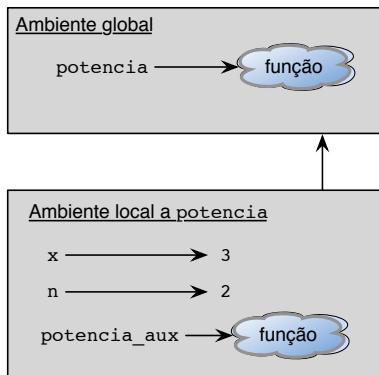
>>> potencia(3, -2)
0.1111111111111111
>>> potencia
<function potencia at 0x10f45d0>
>>> potencia_aux
NameError: name 'potencia_aux' is not defined

```

Esta interacção mostra que a função `potencia` é conhecida pelo Python e que a função `potencia_aux` não é conhecida pelo Python, mesmo depois de este a ter utilizado na execução de `potencia(3, 2)`!

Na Figura 3.5 apresentamos de um modo informal a estrutura de blocos que definimos para a nova versão da função `potencia`. Nesta figura, um bloco é indicado como um rectângulo, dentro do qual aparece a informação do bloco. Note-se que dentro da função `potencia` existe um bloco que corresponde à função `potencia_aux`. Nesta óptica, o comportamento que obtivemos na interacção anterior está de acordo com a regra associada à estrutura de blocos. De acordo com esta regra, toda a informação definida dentro de um bloco pertence a esse bloco, e só pode ser usada por esse bloco e pelos blocos definidos dentro dele. Isto faz com que a função `potencia_aux` só possa ser utilizada pela função `potencia`, o que efectivamente evita a possibilidade de utilizarmos directamente a função `potencia_aux`, como a interacção anterior o demonstra.

Para compreender o funcionamento da função `potencia`, iremos seguir o funcionamento do Python durante a interacção anterior. Ao definir a função `potencia` cria-se, no ambiente global, a associação entre o nome `potencia` e uma entidade computacional correspondente a uma função com certos parâmetros e um corpo (Figura 3.6).

Figura 3.6: Ambiente global com o nome `potencia`.Figura 3.7: Ambiente criado pela execução de `potencia`.

Até agora tudo corresponde ao que sabemos, o nome `potencia` está associado a uma função, cujo corpo é constituído por uma definição de uma função e por uma instrução `if`. Neste âmbito, o Python “sabe” que `potencia` é uma função e desconhece o nome `potencia_aux`.

Ao avaliar a expressão `potencia(3, 2)`, o Python associa os parâmetros formais da função `potencia` aos parâmetros concretos, cria um ambiente local e executa o corpo da função. Durante a execução do corpo da função, o Python encontra a definição da função `potencia_aux`, criando um nome correspondente no ambiente local como se mostra na Figura 3.7. A segunda instrução do corpo de `potencia` corresponde a uma instrução `if`, na qual ambas as alternativas levam à execução de `potencia_aux`, a qual existe no Ambiente local a `potencia`. A avaliação desta função leva à criação de um novo ambiente (indicado na Figura 3.8 com o nome Ambiente local a `potencia_aux`), no qual esta função é avaliada. Depois da execução de `potencia` terminar, os dois ambientes locais desaparecem, voltando-se à situação apresentada na Figura 3.6, e o Python deixa de conhecer o significado de `potencia_aux`, o que justifica a interacção apresentada.

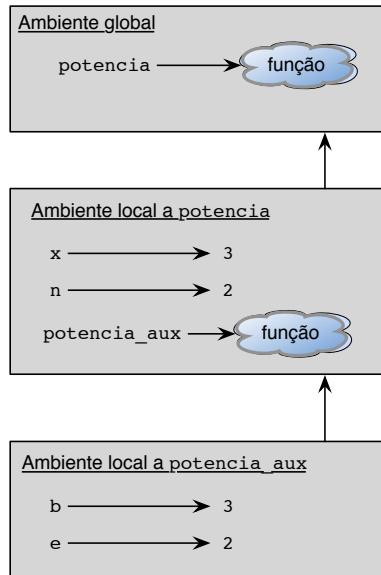


Figura 3.8: Ambientes durante a execução de `potencia`.

Uma questão pertinente a levantar neste momento é a de quando definir funções dentro de outras funções e quando definir funções cujos nomes são colocados no ambiente global. Para responder a esta pergunta devemos considerar que a actividade de programação corresponde à construção de componentes computacionais – por exemplo, funções – que podem ser utilizados como caixas pretas por outros componentes. Sob esta perspectiva, durante a actividade de programação a nossa linguagem de programação está a ser sucessivamente enriquecida no sentido em que vão surgindo operações mais potentes, que podem ser usadas como componentes para desenvolver operações adicionais. Ao escrever um programa devemos decidir se as funções que desenvolvemos devem ser ou não públicas. Por *função pública* entenda-se uma função que existe no ambiente global e, consequentemente, pode ser utilizada por qualquer outra função.

A decisão de definir funções no ambiente global vai prender-se com a utilidade da função e com as restrições impostas à sua utilização. No exemplo do cálculo da potência, é evidente que a operação `potencia` é suficientemente importante para ser disponibilizada publicamente; no entanto a operação auxiliar a que esta recorre, `potencia_aux`, é, como discutimos, privada da função `potencia`, existindo como consequência do algoritmo usado por `potencia`, e deve estar

escondida do exterior, evitando utilizações indevidas.

No caso do cálculo da raiz quadrada (apresentado na Secção 3.4.5) deve também ser claro que a função `calcula_raiz` não deve ser tornada pública, no sentido em que não queremos que o palpite inicial seja fornecido ao algoritmo; as funções `bom_palpite` e `novo_palpite` são claramente específicas ao modo de calcular a raiz quadrada, e portanto devem ser privadas. De acordo com esta discussão, a função `raiz` deverá ser definida do seguinte modo (na Figura 3.9 apresentamos a estrutura de blocos da função `raiz`):

```
def raiz(x):

    def calcula_raiz(x, palpito):

        def bom_palpite(x, palpito):
            return abs(x - palpito * palpito) < delta

        def novo_palpite(x, palpito):
            return (palpito + x / palpito) / 2

        while not bom_palpite(x, palpito):
            palpito = novo_palpite(x, palpito)
        return palpito

    delta = 0.001
    if x >= 0:
        return calcula_raiz (x, 1)
    else:
        raise ValueError ('raiz, argumento negativo')
```

Com a introdução da estrutura de blocos, podemos classificar os nomes utilizados por uma função em três categorias, nomes locais, livres e globais. Recorremos, da página 76, a definição do procedimento para calcular o quadrado de um número:

```
def quadrado(x):
    return x * x
```

```

def raiz(x):
    def calcula_raiz(x, palpito):
        def bom_palpite(x, palpito):
            return abs(x - palpito * palpito) < delta
        def novo_palpite(x, palpito):
            return (palpito + x / palpito) / 2
        while not bom_palpite(x, palpito):
            palpito = novo_palpite(x, palpito)
        return palpito
    delta = 0.001
    if x >= 0:
        return calcula_raiz(x, 1)
    else:
        raise ValueError ('raiz, argumento negativo')

```

Figura 3.9: Estrutura de blocos da função `raiz`.

Sabemos que o nome `x` será associado a um ambiente local durante a avaliação de uma expressão que utilize a função `quadrado`. Esta associação estará activa durante a avaliação do corpo da função. Numa função, qualquer parâmetro formal corresponde a um *nome local*, ou seja, apenas tem significado no âmbito do corpo da função.

Nada nos impede de escrever funções que utilizem nomes que não sejam locais, por exemplo, a função

```

def potencia_estranya(n):
    pot = 1
    while n != 0:
        pot = pot * x
        n = n - 1
    return pot

```

contém no seu corpo uma referência a um nome (`x`) que não é um nome local. Com esta função podemos gerar a seguinte interacção:

```

>>> potencia_estranya(3)
NameError: global name 'x' is not defined
>>> x = 5
>>> potencia_estranya(3)
125

```

Na interacção anterior, ao tentarmos avaliar a função `potencia_estranha`, o Python gerou um erro, pois não sabia qual o significado do nome `x`. A partir do momento em que dizemos que o nome `x` está associado ao valor 5, através da instrução `x = 5`, podemos executar, sem gerar um erro, a função `potencia_estranha`. No entanto esta tem uma particularidade: calcula sempre a potência para a base cujo valor está associado a `x` (no nosso caso, 5). Um nome, tal como `x` no nosso exemplo, que apareça no corpo de uma função e que não seja um nome local é chamado um *nome não local*.

Entre os nomes não locais, podemos ainda considerar duas categorias: os *nomes globais*, aqueles que pertencem ao ambiente global e que são conhecidos por todas as funções (como é o caso do nome `x` na interacção anterior), e os que não são globais, a que se dá o nome de *livres*. Consideremos a seguinte definição da função `potencia_estranha_2` que utiliza um nome livre:

```
def potencia_tambem_estranha(x, n):

    def potencia_estranha_2(n):
        pot = 1
        while n != 0:
            pot = pot * x
            n = n - 1
        return pot

    return potencia_estranha_2(n)
```

A função `potencia_estranha_2` utiliza o nome `x` que não é local, mas, neste caso, `x` também não é global, ele é um nome local (na realidade é um parâmetro formal) da função `potencia_tambem_estranha`.

Durante a execução da função `potencia_tambem_estranha`, o nome `x` está ligado a um objecto computacional, e função `potencia_estranha_2` vai utilizar essa ligação. Na Figura 3.10 apresentamos a estrutura de ambientes criados durante a avaliação de `potencia_tambem_estranha(4, 2)`. Note-se que no Ambiente local a `potencia_t_estranha_2`, a variável `x` não existe. Para calcular o seu valor, o Python vai explorar a sequência de ambientes obtidos seguindo as setas que ligam os ambientes: o Python começa por procurar o valor de `x` no Ambiente local a `potencia_estranha_2`, como este ambiente não contém o nome `x`, o Python

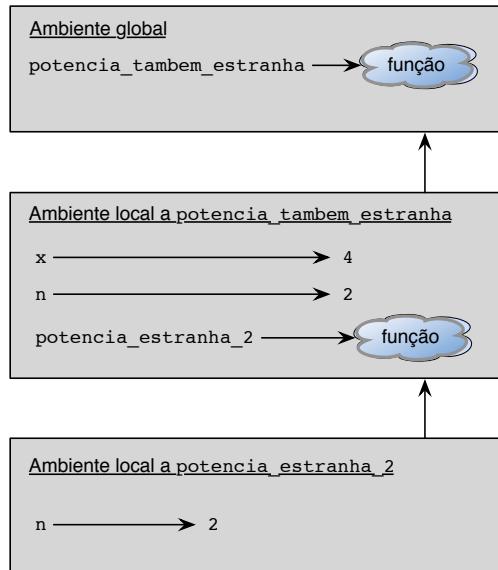


Figura 3.10: Ambientes criados pela avaliação de `potencia_tambem_estranya(4, 2)`.

procura o seu valor no ambiente **Ambiente local a `potencia_tambem_estranya`**, encontrando o valor 4. A utilização de nomes livres é útil quando se desenvolvem funções com muitos parâmetros e que utilizam a estrutura de blocos.

Em resumo, os nomes podem ser locais ou não locais, sendo estes últimos ainda divididos em livres e globais:

$$\text{nome} \left\{ \begin{array}{l} \text{local} \\ \text{não local} \end{array} \right. \left\{ \begin{array}{l} \text{global} \\ \text{livre} \end{array} \right.$$

Define-se *domínio* de um nome como a gama de instruções pelas quais o nome é conhecido, ou seja, o conjunto das instruções onde o nome pode ser utilizado. O domínio dos nomes locais é o corpo da função de que são parâmetros formais ou na qual são definidos; o domínio dos nomes globais é o conjunto de todas as instruções numa interacção em Python.

O tipo de domínio utilizado em Python é chamado *domínio estático*: o domínio de um nome é definido em termos da estrutura do programa (a hierarquia dos

seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

O Python permite a utilização de nomes não locais mas não permite a sua alteração. Isto significa que se numa função se executar uma instrução de atribuição a um nome que não é local, o Python cria uma variável local correspondente a esse nome e altera essa variável local, não alterando a variável não local. Este aspecto é ilustrado com a seguinte função:

```
def potencia_ainda_mais_estranha(x, n):

    def potencia_estranha_3(n):
        pot = 1
        x = 7
        while n > 0:
            pot = pot * x
            n = n - 1
        return pot

    print('x=', x)
    res = potencia_estranha_3(n)
    print('x=', x)
    return res
```

a qual permite gerar a seguinte interacção:

```
>>> potencia_ainda_mais_estranha(4, 2)
x= 4
x= 4
49
```

Com efeito, a instrução `x = 7` no corpo da função `potencia_estranha_3` cria o nome `x` como uma variável local à função `potencia_estranha_3`, sendo o seu valor utilizado no cálculo da potência. Fora desta função, a variável `x` mantém o seu valor.

De modo a permitir a partilha de variáveis não locais entre funções, existe em Python a instrução `global`, a qual tem a seguinte sintaxe em notação BNF⁹:

⁹Na realidade, esta instrução corresponde a uma *directive* para o interpretador do Python.

$\langle \text{instrução global} \rangle ::= \text{global } \langle \text{nomes} \rangle$

Quando a instrução `global` aparece no corpo de uma função, o Python considera que $\langle \text{nomes} \rangle$ correspondem a variáveis partilhadas e permite a sua alteração como variáveis não locais. A instrução `global` não pode referir parâmetros formais de funções. A utilização da instrução `global` é ilustrada na seguinte função:

```
def potencia_ainda_mais_estranha_2(n):

    def potencia_estranha_4(n):
        global x
        pot = 1
        x = 7
        while n > 0:
            pot = pot * x
            n = n - 1
        return pot

    print('x=', x)
    res = potencia_estranha_4(n)
    print('x=', x)
    return res
```

a qual permite gerar a interacção:

```
>>> x = 4
>>> potencia_ainda_mais_estranha_2(2)
x= 4
x= 7
49
```

Devido à restrição imposta pela instrução `global` de não poder alterar parâmetros formais, na função `potencia_ainda_mais_estranha_2` utilizamos `x` como uma variável global.

Ao terminar esta secção é importante dizer que a utilização exclusiva de nomes locais permite manter a independência entre funções, no sentido em que toda a

comunicação entre elas é limitada à associação dos parâmetros concretos com os parâmetros formais. Quando este tipo de comunicação é mantido, para utilizar uma função, apenas é preciso saber o que ela faz, e não como foi programada. Já sabemos que isto se chama abstracção procedural.

Embora a utilização de nomes locais seja vantajosa e deva ser utilizada sempre que possível, é por vezes conveniente permitir o acesso a nomes não locais. O facto de um nome ser não local significa que o objecto computacional associado a esse nome é compartilhado por vários blocos do programa, que o podem consultar e, eventualmente, modificar.

3.6 Módulos

Na Secção 3.4 desenvolvemos algumas funções matemáticas elementares, potência, factorial, máximo divisor comum, raiz quadrada e seno. É cada vez mais raro que ao escrever um programa se definam todas as funções que esse programa necessita, é muito mais comum, e produtivo, recorrer aos milhões de linhas de código que outros programadores escreveram e tornaram públicas.

Um *módulo* (também conhecido por *biblioteca*) é uma colecção de funções agrupadas num único ficheiro. As funções existentes no módulo estão relacionadas entre si. Por exemplo, muitas das funções matemáticas de uso comum, como a raiz quadrada e o seno, estão definidas no módulo `math`. Recorrendo à utilização de módulos, quando necessitamos de utilizar uma destas funções, em lugar de a escrevermos a partir do zero, utilizamos as suas definições que já foram programadas por outra pessoa.

Para utilizar um módulo, é necessário *importar* para o nosso programa as funções definidas nesse módulo. A importação de funções é realizada através da *instrução de importação*, a qual apresenta a seguinte sintaxe, utilizando a notação BNF:

```
<instrução de importação> ::= import <módulo> |
                           from <módulo> import <nomes a importar>
<módulo> ::= <nome>
<nomes a importar> ::= * |
                           <nomes>
```

$$\langle \text{nomes} \rangle ::= \langle \text{nome} \rangle | \\ \langle \text{nome} \rangle, \langle \text{nomes} \rangle$$

A instrução de importação apresenta duas formas distintas. A primeira destas, `import <módulo>`, indica ao Python para importar para o programa todas as funções existentes no módulo especificado. A partir do momento em que uma instrução de importação é executada, passam a existir no programa nomes correspondentes às funções existentes no módulo, como se de funções embutidas se tratasse.

As funções do módulo são referenciadas através de uma variação de nomes chamada *nome composto* (ver a Secção 2.3), a qual é definida sintaticamente através da seguinte expressão em notação BNF:

$$\langle \text{nome composto} \rangle ::= \langle \text{nome simples} \rangle . \langle \text{nome simples} \rangle$$

Neste caso, um `<nome composto>` corresponde à especificação do nome do módulo, seguido por um ponto, seguido pelo nome da função. Consideremos o módulo `math`, o qual contém, entre outras, as funções e os nomes apresentados na Tabela 3.5. Com este módulo, podemos originar a seguinte interacção:

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(4)
2.0
>>> math.sin(math.pi/2)
1.0
```

A segunda forma da instrução de importação, `from <módulo> import <nomes a importar>`, permite-nos indicar quais as funções ou nomes a importar do módulo, os `<nomes a importar>`. Após a instrução desta instrução, apenas as funções especificadas são importadas para o nosso programa. Para além disso, os nomes importados não têm que ser referenciados através da indicação de um nome composto, como o mostra a seguinte interacção:

```
>>> from math import pi, sin
>>> pi
3.141592653589793
```

Python	Matemática	Significado
<code>pi</code>	π	Uma aproximação de π
<code>e</code>	e	Uma aproximação de e
<code>sin(x)</code>	$\text{sen}(x)$	O seno de x (x em radianos)
<code>cos(x)</code>	$\text{cos}(x)$	O cosseno de x (x em radianos)
<code>tan(x)</code>	$\text{tg}(x)$	A tangente de x (x em radianos)
<code>log(x)</code>	$\ln(x)$	O logaritmo natural de x
<code>exp(x)</code>	e^x	A função inversa de \ln
<code>pow(x, y)</code>	x^y	O valor de x levantado a y
<code>sqrt(x)</code>	\sqrt{x}	A raiz quadrada de x
<code>ceil(x)</code>	$\lceil x \rceil$	O maior inteiro superior ou igual a x
<code>floor(x)</code>	$\lfloor x \rfloor$	O maior inteiro inferior ou igual a x

Tabela 3.5: Algumas funções disponíveis no módulo `math`.

```
>>> sqrt(4)
NameError: name 'sqrt' is not defined
>>> sin(pi/2)
1.0
```

Se na especificação de `<nomes a importar>` utilizarmos o símbolo `*`, então todos os nomes do módulo são importados para o nosso programa, como se ilustra na interacção:

```
>>> from math import *
>>> pi
3.141592653589793
>>> sqrt(4)
2.0
>>> sin(pi/2)
1.0
```

Aparentemente, a utilização de `from <módulo> import *` parece ser preferível a `import <módulo>`. No entanto, pode acontecer que dois módulos diferentes utilizem o mesmo nome para referirem funções diferentes. Suponhamos que o módulo `m1` define a função `f` e que o módulo `m2` também define a função `f`, mas com outro significado. A execução das instruções

```
from m1 import *
from m2 import *
```

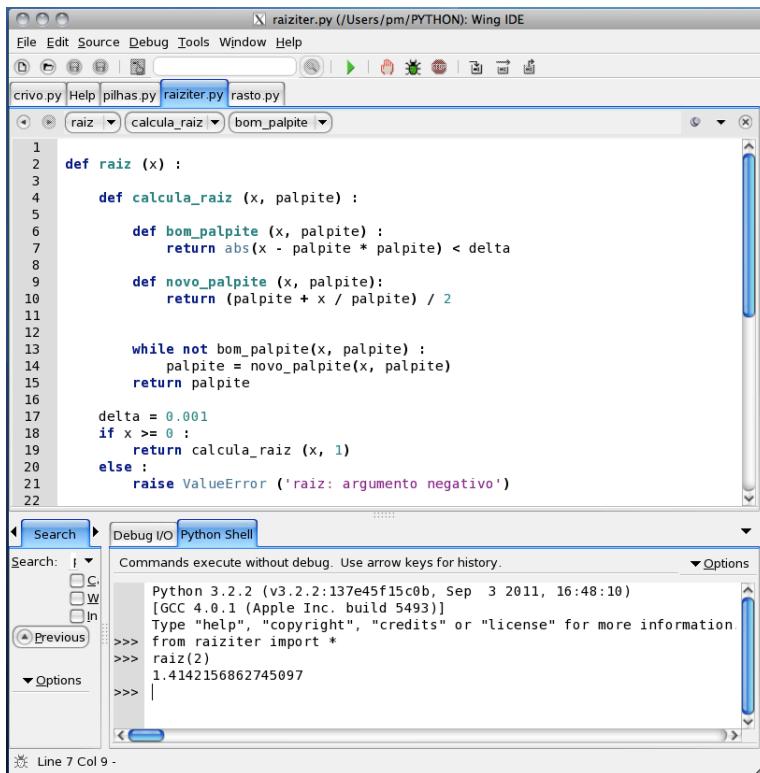


Figura 3.11: Utilização do módulo `raiziter` no Wing 101.

tem o efeito de “destruir” a definição da função `f` importada do módulo `m1`, substituindo-a pela definição da função `f` importada do módulo `m2` (pois esta importação é feita em segundo lugar). Por outro lado, a execução das instruções

```
import m1
import m2
```

permite a coexistência das duas funções `f`, sendo uma delas conhecida por `m1.f` e a outra por `m2.f`.

A lista de todos os módulos disponíveis em Python pode ser consultada em <http://docs.python.org/modindex>. Para a criação de novos módulos por parte do programador basta criar um ficheiro com extensão `.py`, e utilizar um dos comandos de importação de módulos descritos nesta secção, não indicando a extensão do ficheiro. Na Figura 3.11 apresentamos a utilização do módul

`raiziter`, o qual contém a definição da função raiz quadrada, tal como apresentada na Secção 3.4.5.

3.7 Notas finais

Apresentámos o modo de definir e utilizar novas funções em Python, e o conceito subjacente à sua utilização, a abstracção procedural, que consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam, ou seja, a separação do “*como*” de “*o que*”.

Discutimos a diferença entre a abordagem da matemática à definição de funções e a abordagem da informática ao mesmo assunto. Apresentámos exemplos de funções que calculam valores exactos e funções que calculam valores aproximados. Introduzimos o conceito de erro absoluto. O ramo da informática dedicado ao cálculo numérico é conhecido como *matemática numérica* ou *computação numérica*. Uma boa abordagem a este tema pode ser consultada em [Ascher e Greif, 2011].

Apresentámos o conceito de estrutura de blocos e a distinção entre nomes globais, livres e locais.

Finalmente introduzimos o conceito de módulo.

3.8 Exercícios

1. Escreva uma função com o nome `cinco` que tem o valor `True` se o seu argumento for `5` e `False` no caso contrário. Não pode utilizar uma instrução `if`.
2. Escreva uma função com o nome `bissesto` que determina se um ano é bissexto. Um ano é bissexto se for divisível por 4 e não for divisível por 100, a não ser que seja também divisível por 400. Por exemplo, 1984 é bissexto, 1100 não é, e 2000 é bissexto.
3. Um número *primo* é um número inteiro maior do que 1 que apenas é divisível por 1 e por si próprio. Por exemplo, 5 é primo porque apenas é divisível por si próprio e por um, ao passo que 6 não é primo pois é divisível por 1, 2, 3, e 6. Os números primos têm um papel muito importante tanto

em Matemática como em Informática. Um método simples, mas pouco eficiente, para determinar se um número, n , é primo consiste em testar se n é múltiplo de algum número entre 2 e \sqrt{n} . Usando este processo, escreva uma função em Python chamada `primo` que recebe um número inteiro e tem o valor `True` apenas se o seu argumento for primo.

4. Um número n é o n -ésimo *primo* se for primo e existirem $n - 1$ números primos menores que ele. Usando a função `primo` do exercício anterior, escreva uma função com o nome `n_esimo_primo` que recebe como argumento um número inteiro, `n`, e devolve o n -ésimo número primo.
5. Um número inteiro, n , diz-se *triangular* se existir um inteiro m tal que $n = 1 + 2 + \dots + (m - 1) + m$. Escreva uma função chamada `triangular` que recebe um número inteiro positivo `n`, e cujo valor é `True` apenas se o número for triangular. No caso de `n` ser 0 deverá devolver `False`. Por exemplo,

```
>>> triangular(6)
True
>>> triangular(8)
False
```

6. Escreva uma função em Python que calcula o valor aproximado da série para um determinado valor de x :

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

A sua função não pode utilizar as funções potência nem factorial.

Capítulo 4

Tuplos e ciclos contados

*“Then you keep moving round, I suppose?” said Alice.
“Exactly so,” said the Hatter: “as the things get used up.”
“But what happens when you come to the beginning again?” Alice ventured to ask.*

Lewis Carroll, *Alice’s Adventures in Wonderland*

Até agora, os elementos dos tipos de informação que considerámos correspondem a um único valor, um inteiro, um real ou um valor lógico. Este é o primeiro capítulo em que discutimos tipos estruturados de informação, ou seja, tipos de informação em que os seus elementos estão associados a um agregado de valores. Recorde-se que um tipo de informação corresponde a um conjunto de entidades, os elementos do tipo, e a um conjunto de operações aplicáveis a essas entidades. Os tipos de informação cujos elementos estão associados a um agregado de valores são chamados *tipos estruturados de informação, tipos de informação não elementares ou estruturas de informação*. Sempre que abordamos um tipo estruturado de informação, temos que considerar o modo como os valores estão agregados e as operações que podemos efectuar sobre os elementos do tipo.

4.1 Tuplos

Um *tuplo*, em Python designado por `tuple`, é uma sequência de elementos. Os tuplos correspondem à noção matemática de vector. Em matemática, para nos

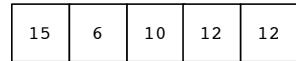


Figura 4.1: Representação gráfica de um tuplo.

referirmos aos elementos de um vector, utilizamos índices que caracterizam univocamente estes elementos. Por exemplo, se \vec{x} representa um vector com três elementos, estes são caracterizados, respectivamente, por x_1 , x_2 , e x_3 . Analogamente, em Python os elementos de um tuplo são referidos, indicando a posição que o elemento ocupa dentro do tuplo. Tal como em matemática, esta posição tem o nome de *índice*. Na Figura 4.1 apresentamos, de um modo esquemático, um tuplo com cinco elementos, 15, 6, 10, 12 e 12. O elemento que se encontra na primeira posição do tuplo é 15, o elemento na segunda posição é 6 e assim sucessivamente.

Em Python, a representação externa de um tuplo¹ é definida sintacticamente pelas seguintes expressões em notação BNF^{2, 3}:

```

⟨tuplo⟩ ::= () |
            ⟨elemento⟩, ⟨elementos⟩

⟨elementos⟩ ::= ⟨nada⟩ |
                ⟨elemento⟩ |
                ⟨elemento⟩, ⟨elementos⟩

⟨elemento⟩ ::= ⟨expressão⟩ |
                ⟨tuplo⟩ |
                ⟨lista⟩ |
                ⟨dicionário⟩

⟨nada⟩ ::=

```

O tuplo () não tem elementos e é chamado o *tuplo vazio*. As seguintes entidades representam tuplos em Python (1, 2, 3), (2, True), (1,). Note-se que o último tuplo apenas tem um elemento. A definição sintáctica de um tuplo exige que um tuplo com um elemento contenha esse elemento seguido de uma

¹ Recorde-se que a *representação externa* de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do como como esta é representada internamente no computador.

² É ainda possível representar tuplos sem escrever os parenteses, mas essa alternativa não é considerada neste livro.

³ As definições de ⟨lista⟩ e ⟨dicionário⟩ são apresentadas, respectivamente, nos capítulos 5 e 9.

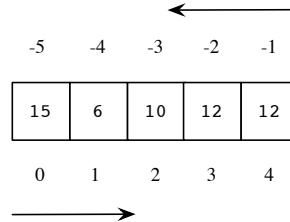


Figura 4.2: Valores dos índices de um tuplo.

vírgula, pelo que (1) *não* corresponde a um tuplo em Python. De acordo com as expressões anteriores em notação BNF, (1, 2,) e (1, 2, 3,) são tuplos, respectivamente com 2 e 3 elementos. O tuplo apresentado na Figura 4.1 corresponde a (15, 6, 10, 12, 12). A definição de um tuplo permite que os seus elementos sejam, por sua vez, outros tuplos. Por exemplo, ((1, 2, 3), 4, (5, 6)) é um tuplo com 3 elementos, sendo o primeiro e o último outros tuplos.

Depois da criação de um tuplo, podemos referir-nos a qualquer dos seus elementos especificando o nome do tuplo e a posição que o elemento desejado ocupa dentro deste. A referência a um elemento de um tuplo corresponde a um *nome indexado*, o qual é definido através da seguinte expressão em notação BNF:

$\langle \text{nome indexado} \rangle ::= \langle \text{nome} \rangle [\langle \text{expressão} \rangle]$

em que $\langle \text{nome} \rangle$ corresponde ao nome do tuplo e $\langle \text{expressão} \rangle$ (que é do tipo inteiro⁴) corresponde à especificação da posição do elemento dentro do tuplo. As entidades utilizadas para especificar a posição de um elemento de um tuplo são chamadas *índices*. Os índices começam no número zero (correspondente ao primeiro elemento do tuplo), aumentando linearmente até ao número de elementos do tuplo menos um; em alternativa, o índice -1 corresponde ao último elemento do tuplo, o índice -2 corresponde ao penúltimo elemento do tuplo e assim sucessivamente, como se mostra na Figura 4.2. Por exemplo, com base no tuplo apresentado na Figura 4.2, podemos gerar a seguinte interacção:

```
>>> notas = (15, 6, 10, 12, 12)
>>> notas
(15, 6, 10, 12, 12)
```

⁴Este aspecto não pode ser especificado utilizando a notação BNF.

```
>>> notas[0]
15
>>> notas[-2]
12
>>> i = 1
>>> notas[i+1]
10
>>> notas[i+10]
IndexError: tuple index out of range
```

Note-se que na última expressão da interacção anterior, tentamos utilizar o índice 11 ($= i + 10$), o que origina um erro, pois para o tuplo `notas` o maior valor do índice é 4.

Consideremos agora a seguinte interacção que utiliza um tuplo cujos elementos são outros tuplos:

```
>>> a = ((1, 2, 3), 4, (5, 6))
>>> a[0]
(1, 2, 3)
>>> a[0][1]
2
```

A identificação de um elemento de um tuplo (o nome do tuplo seguido do índice dentro de parêntesis rectos) é um nome indexado, pelo que poderemos ser tentados a utilizá-lo como uma variável e, consequentemente, sujeitá-lo a qualquer operação aplicável às variáveis do seu tipo. No entanto, os tuplos em Python são entidades *imutáveis*, significando que os elementos de um tuplo não podem ser alterados como o mostra a seguinte interacção:

```
>>> a = ((1, 2, 3), 4, (5, 6))
>>> a[1] = 10
TypeError: 'tuple' object does not support item assignment
```

Sobre os tuplos podemos utilizar as funções embutidas apresentadas na Tabela 4.1. Nesta tabela “Universal” significa qualquer tipo. Note-se que as operações `+` e `*` são sobrecarregadas, pois também são aplicáveis a inteiros e a reais.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$t_1 + t_2$	Tuplos	A concatenação dos tuplos t_1 e t_2 .
$t * i$	Tuplo e inteiro	A repetição i vezes do tuplo t .
$t[i_1:i_2]$	Tuplo e inteiros	O sub-tuplo de t entre os índices i_1 e $i_2 - 1$.
$e \text{ in } t$	Universal e tuplo	True se o elemento e pertence ao tuplo t ; False em caso contrário.
$e \text{ not in } t$	Universal e tuplo	A negação do resultado da operação $e \text{ in } t$.
<code>tuple(a)</code>	Lista ou dicionário ou cadeia de caracteres	Transforma o seu argumento num tuplo. Se não forem fornecidos argumentos, devolve o tuplo vazio.
<code>len(t)</code>	Tuplo	O número de elementos do tuplo t .

Tabela 4.1: Operações embutidas sobre tuplos.

A seguinte interacção mostra a utilização de algumas operações sobre tuplos:

```
>>> a = (1, 2, 3)
>>> b = (7, 8, 9)
>>> a + b
(1, 2, 3, 7, 8, 9)
>>> c = a + b
>>> c[2:4]
(3, 7)
>>> a * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> 3 in a
True
>>> 4 in a
False
>>> len(a)
3
>>> a[:2]
(1, 2)
>>> a[2:]
(3,)
>>> a[:]
(1, 2, 3)
```

As últimas linhas da interacção anterior mostram que se na operação $t[e_1 : e_2]$, um dos índices for omitido, então o Python assume o valor zero se o índice omitido for e_1 ou o maior índice do tuplo mais um se o índice omitido for e_2 .

Consideremos a seguinte interacção:

```
>>> a = (3, 4, 5, 6)
>>> b = (7, 8)
>>> a = a + b
>>> a
(3, 4, 5, 6, 7, 8)
```

Podemos ser levados a pensar que no penúltimo comando que fornecemos ao Python, `a = a + b`, alterámos o tuplo `a`, o que pode parecer uma violação ao facto de os tuplos serem entidades imutáveis. O que na realidade aconteceu, foi que modificámos o valor da variável `a`, a qual estava associada a um tuplo, sendo esta uma operação perfeitamente legítima. Quando afirmámos que os tuplos são imutáveis, queríamos dizer que não podemos alterar um valor de um elemento de um tuplo, podendo perfeitamente criar tuplos a partir de outros tuplos, como a interacção anterior o mostra.

Podemos escrever a seguinte função que recebe um tuplo (`t`), uma posição especificada por um índice positivo (`p`) e um valor qualquer (`v`) e que devolve um tuplo igual ao tuplo fornecido, excepto que o elemento que se encontra na posição `p` é substituído por `v`:

```
def substitui(t, p, v):
    if 0 <= p <= len(t)-1:
        return t[:p] + (v,) + t[p+1:]
    else:
        raise IndexError ('na função substitui')
```

Com esta função, podemos gerar a interacção:

```
>>> a = (3, 'a', True, 'b', 2, 0, False)
>>> substitui(a, 1, 'x')
(3, 'x', True, 'b', 2, 0, False)
>>> a
```

```
(3, 'a', True, 'b', 2, 0, False)
>>> substitui(a, 12, 'x')
IndexError: na função substitui
```

Uma das operações que é comum realizar sobre tipos estruturados que correspondem a sequências de elementos, de que os tuplos são um de muitos exemplos, consiste em processar, de um modo idêntico, todos os elementos da sequência. Como exemplo de uma operação deste tipo, suponhamos que desejávamos escrever uma função que recebe um tuplo e que calcula a soma dos seus elementos. Esta função deverá inicializar uma variável que conterá o valor da soma para o valor zero e, em seguida, deverá percorrer todos os elementos do tuplo, adicionando o valor de cada um deles à variável que corresponde à soma. Depois de percorridos todos os elementos do tuplo, a variável correspondente à soma irá conter a soma de todos os elementos. Podemos recorrer a um ciclo `while` para escrever a seguinte função:

```
def soma_elementos(t):
    soma = 0
    i = 0
    while i < len(t):
        soma = soma + t[i]
        i = i + 1
    return soma
```

Consideremos agora o problema, bastante mais complicado do que o anterior, de escrever uma função, chamada `alisa`, que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve um tuplo contendo todos os elementos correspondentes a tipos elementares de informação (inteiros, reais ou valores lógicos) do tuplo original. Por exemplo, com esta função, pretendemos obter a interacção:

```
>>> alisa((1, 2, ((3, ), ((4, ), ), 5), (6, ((7, ), ))))
(1, 2, 3, 4, 5, 6, 7)
>>> alisa((((((5, 6), ), ), ), ), )
(5, 6)
```

Para escrever a função `alisa`, iremos utilizar a função embutida `isinstance`, cuja sintaxe é definida através das seguintes expressões em notação BNF:

<code>t</code>	<code>i</code>	<code>t[:i]</code>	<code>t[i]</code>	<code>t[i+1:]</code>
<code>((1, 2), 3, (4, (5)))</code>	<code>0</code>	<code>()</code>	<code>(1, 2)</code>	<code>(3, (4, 5))</code>
<code>(1, 2, 3, (4, 5))</code>	<code>1</code>			
<code>(1, 2, 3, (4, 5))</code>	<code>2</code>			
<code>(1, 2, 3, (4, 5))</code>	<code>3</code>	<code>(1, 2, 3)</code>	<code>(4, 5)</code>	<code>()</code>
<code>(1, 2, 3, 4, 5)</code>	<code>4</code>			

Tabela 4.2: Funcionamento de `alisa(((1, 2), 3, (4, (5))))`.

```
isinstance(<expressão>, <designação de tipo>)

<designação de tipo> ::= <expressão> |
                           <tuplo>
```

A função de tipo lógico `isinstance`, tem o valor `True` apenas se o tipo da expressão que é o seu primeiro argumento corresponde ao seu segundo argumento ou se pertence ao tuplo que é seu segundo argumento. Por exemplo:

```
>>> isinstance(3, int)
True
>>> isinstance(False, (float, bool))
True
>>> isinstance((1, 2, 3), tuple)
True
>>> isinstance(3, float)
False
```

A função `alisa` recebe como argumento um tuplo, `t`, e percorre todos os elementos do tuplo `t`, utilizando um índice, `i`. Ao encontrar um elemento que é um tuplo, a função modifica o tuplo original, gerando um tuplo com todos os elementos antes do índice `i` (`t[:i]`), seguido dos elementos do tuplo correspondente a `t[i]`, seguido de todos os elementos depois do índice `i` (`t[i+1:]`). Se o elemento não for um tuplo, a função passa a considerar o elemento seguinte, incrementando o valor de `i`. Na Tabela 4.2 apresentamos o funcionamento desta função para a avaliação de `alisa(((1, 2), 3, (4, (5))))`. Como para certos valores de `i`, a expressão `isinstance(t[i], tuple)` tem o valor `False`, para esses valores não se mostram na Tabela 4.2 os valores de `t[:i]`, `t[i]` e `t[i+1:]`.

```
def alisa(t):
```

```

i = 0
while i < len(t):
    if isinstance(t[i], tuple):
        t = t[:i] + t[i] + t[i+1:]
    else:
        i = i + 1
return t

```

4.2 Ciclos contados

O ciclo que utilizámos na função `soma_elementos`, apresentada na página 121, obriga-nos a inicializar o índice para o valor zero (`i = 0`) e obriga-nos também a actualizar o valor do índice após termos somado o valor correspondente (`i = i + 1`). Uma alternativa para o ciclo `while` que utilizámos nessa função é a utilização de um ciclo contado.

Um *ciclo contado*⁵ é um ciclo cuja execução é controlada por uma variável, designada por *variável de controle*. Para a variável de controle é especificado o seu valor inicial, a forma de actualizar o valor da variável em cada passagem pelo ciclo e a condição de paragem do ciclo. Um ciclo contado executa repetidamente uma sequência de instruções, para uma sequência de valores da variável de controle.

Em Python, um ciclo contado é realizado através da utilização da instrução `for`, a qual permite especificar a execução repetitiva de uma instrução composta para uma sequência de valores de uma variável de controle. A sintaxe da instrução `for` é definida pela seguinte expressão em notação BNF⁶:

⟨instrução for⟩ ::= for ⟨nome simples⟩ in ⟨expressão⟩ : [CR]
 ⟨instrução composta⟩

Na definição sintáctica da instrução `for`, ⟨nome simples⟩ corresponde à variável de controle, ⟨expressão⟩ representa uma expressão cujo valor corresponde a uma sequência (novamente, este aspecto não pode ser especificado utilizando apenas a notação BNF) e ⟨instrução composta⟩ corresponde ao corpo do ciclo. Por agora, o único tipo de sequências que encontrámos foram os tuplos, embora

⁵Em inglês, “counted loop”.

⁶A palavra “for” traduz-se em português por “para”.

existam outros tipos de sequências, pelo que as sequências utilizadas nas nossas primeiras utilizações da instrução `for` apenas usam sequências correspondentes a tuplos.

A semântica da instrução `for` é a seguinte: ao encontrar a instrução `for <var> in <expressão>: [CR] <inst_comp>`, o Python executa as instruções correspondentes a `<inst_comp>` para os valores da variável `<var>` correspondentes aos elementos da sequência resultante da avaliação de `<expressão>`.

No corpo do ciclo de uma instrução `for` pode também ser utilizada a instrução `break` apresentada na página 66. Ao encontrar uma instrução `break`, o Python termina a execução do ciclo, independentemente do valor da variável que controla o ciclo.

Com a instrução `for` podemos gerar a seguinte interacção:

```
>>> for i in (1, 3, 5):
...     print(i)
...
1
3
5
```

Utilizando a instrução `for` podemos agora escrever a seguinte variação da função `soma_elementos`, apresentada na página 121, que recebe um tuplo e devolve a soma de todos os seus elementos:

```
def soma_elementos(t):
    soma = 0
    for e in t:
        soma = soma + e
    return soma
```

com a qual obtemos a interacção:

```
>>> soma_elementos((1, 2))
3
```

O Python fornece também a função embutida `range` que permite a geração

de sequências de elementos. A função `range` é definida através das seguintes expressões em notação BNF:

```
range(<argumentos>)

<argumentos> ::= <expressão> |
                  <expressão>, <expressão> |
                  <expressão>, <expressão>, <expressão>
```

Sendo e_1 , e_2 e e_3 expressões cujo valor é um inteiro, a função `range` origina uma progressão aritmética, definida do seguinte modo para cada possibilidade dos seus argumentos:

1. `range(e_1)` devolve a sequência contendo os inteiros entre 0 e $e_1 - 1$, ou seja devolve o tuplo $(0, 1, \dots, e_1 - 1)$. Se $e_1 \leq 0$, devolve o tuplo $()$. Por exemplo, o valor de `range(10)` corresponde ao tuplo $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$.
2. `range(e_1, e_2)` devolve a sequência contendo os inteiros entre e_1 e $e_2 - 1$, ou seja devolve o tuplo $(e_1, e_1 + 1, \dots, e_2 - 1)$. Se $e_2 \leq e_1$, devolve o tuplo $()$. Por exemplo, o valor de `range(5, 10)` corresponde ao tuplo $(5, 6, 7, 8, 9)$ e o valor de `range(-3, 3)` corresponde ao tuplo $(-3, -2, -1, 0, 1, 2)$.
3. `range(e_1, e_2, e_3)` devolve a sequência contendo os inteiros que começam em e_1 e nunca sendo superiores a $e_2 - 1$, em que cada elemento da sequência é obtido do anterior somando e_3 , ou seja corresponde ao tuplo $(e_1, e_1 + e_3, e_1 + 2 \cdot e_3, \dots)$. Novamente, se $e_2 \leq e_1$, devolve o tuplo $()$. Por exemplo, o valor de `range(2, 20, 3)` corresponde ao tuplo $(2, 5, 8, 11, 14, 17)$.

Recorrendo à função `range` podemos escrever a seguinte função alternativa para calcular a soma dos elementos de um tuplo:

```
def soma_elementos(t):
    soma = 0
    for i in range(len(t)):
        soma = soma + t[i]
    return soma
```

À primeira vista pode parecer que a utilização de `range` é inútil dado que podemos percorrer todos os elementos de um tuplo `t` usando a instrução `for` e `in t`. Contudo, esta instrução apenas nos permite percorrer os elementos do tuplo, fazendo operações com estes elementos. Suponhamos que desejávamos escrever uma função para determinar se os elementos de um tuplo aparecem ordenados, ou seja, se cada elemento é menor ou igual ao elemento seguinte. A instrução `for` e `in t` embora permita inspeccionar cada elemento do tuplo não nos permite relacioná-lo com o elemento seguinte. Recorrendo à função `range` podemos percorrer o tuplo usando índices, o que já nos permite a comparação desejada como o ilustra a seguinte função:

```
def tuplo_ordenado(t):
    for i in range(len(t)-1):
        if t[i] > t[i+1]:
            return False
    return True
```

Note-se que, em contraste com a função `soma_elementos`, a instrução `for` é executada para os valores de `i` em `range(len(t)-1)`, pois a função `tuplo_ordenado` compara cada elemento do tuplo com o seguinte. Se tivesse sido utilizado `range(len(t))`, quando `i` fosse igual a `len(t)-1` (o último valor de `i` neste ciclo), a expressão `t[i] > t[i+1]` dava origem a um erro pois `t[len(t)]` referencia um índice que não pertence ao tuplo.

Os ciclos `while` e `for`, têm características distintas. Assim, põe-se a questão de saber que tipo de ciclo escolher em cada situação.

Em primeiro lugar, notemos que o ciclo `while` permite fazer tudo o que o ciclo `for` permite fazer⁷. No entanto, a utilização do ciclo `for`, quando possível, é mais eficiente do que o ciclo `while` equivalente. Assim, a regra a seguir na escolha de um ciclo é simples: sempre que possível, utilizar um ciclo `for`; se tal não for possível, usar um ciclo `while`.

Convém também notar que existem certas situações em que processamos todos os elementos de um tuplo mas não podemos usar um ciclo `for`, como acontece com a função `alisa` apresentada na página 122. Na realidade, nesta função estamos a processar uma variável, cujo tuplo associado é alterado durante o

⁷Como exercício, deve exprimir o ciclo `for` em termos do ciclo `while`.

processamento (o número de elementos do tuplo pode aumentar durante o processamento) e consequentemente não podemos saber à partida quantos elementos vamos considerar.

4.3 Cadeias de caracteres revisitadas

No início do Capítulo 2 introduzimos as cadeias de caracteres como constantes. Dissemos que uma cadeia de caracteres é qualquer sequência de caracteres delimitada por plicas. Desde então, temos usado cadeias de caracteres nos nossos programas para produzir mensagens para os utilizadores.

Em Python, as cadeias de caracteres correspondem a um tipo estruturado de informação, designado por `str`⁸, o qual corresponde a uma sequência de caracteres individuais.

As cadeias de caracteres são definidas através das seguintes expressões em notação BNF:

```
<cadeia de caracteres> ::= '<carácter>*' , |  
                      "<carácter>*" |  
                      """<carácter>*"""
```

A definição anterior indica que uma cadeia de caracteres é uma sequência de zero ou mais caracteres delimitados por plicas, por aspas ou por três aspas, devendo os símbolos que delimitam a cadeia de caracteres ser iguais (por exemplo "`abc`" não é uma cadeia de caracteres). Como condição adicional, não apresentada na definição em notação BNF, os caracteres de uma cadeia de caracteres delimitadas por plicas não podem conter a plica e os caracteres de uma cadeia de caracteres delimitadas por aspas não podem conter aspas. As cadeias de caracteres ' ' e " " são chamadas *cadeias de caracteres vazias*. Ao longo do livro utilizamos as plicas para delimitar as cadeias de caracteres.

As cadeias de caracteres delimitadas por três aspas, chamadas *cadeias de caracteres de documentação*⁹, são usadas para documentar definições. Quando o Python encontra uma cadeia de caracteres de documentação, na linha imediatamente a seguir a uma linha que começa pela palavra `def` (a qual corresponde a uma definição), o Python associa o conteúdo dessa cadeia de caracteres à en-

⁸Da sua designação em inglês, “string”.

⁹Do inglês “docstring”.

tidade que está a ser definida. A ideia subjacente é permitir a consulta rápida de informação associada com a entidade definida, recorrendo à função `help`. A função `help(<nome>)` mostra no ecrã a definição associada a `<nome>`, bem como o conteúdo da cadeia de caracteres de documentação que lhe está associada.

Por exemplo, suponhamos que em relação à função `soma_elementos` apresentada na página 124, associávamos a seguinte cadeia de caracteres de documentação:

```
def soma_elementos(t):
    """
    Recebe um tuplo e devolve a soma dos seus elementos
    """
    soma = 0
    for e in t:
        soma = soma + e
    return soma
```

Com esta definição, podemos gerar a seguinte interacção:

```
>>> help(soma_elementos)
Help on function soma_elementos in module __main__:

soma_elementos(t)
    Recebe um tuplo e devolve a soma dos seus elementos
```

Deste modo, podemos rapidamente saber qual a forma de invocação de uma dada função e obter a informação do que a função faz. Neste momento, a utilidade da função `help` pode não ser evidente, mas quando trabalhamos com grande programas contendo centenas ou milhares de definições, esta torna-se bastante útil.

Tal como os tuplos, as cadeias de caracteres são tipos *imutáveis*, no sentido de que não podemos alterar os seus elementos.

Os elementos das cadeias de caracteres são referenciados utilizando um índice, de um modo semelhante ao que é feito em relação aos tuplos. Por exemplo, se `id_fp` for uma variável que corresponde à cadeia de caracteres 'FP 12' (Figura 4.3), então `id_fp[0]` e `id_fp[-1]` correspondem, respectivamente a 'F' e '2'. Cada um destes elementos é uma cadeia de caracteres com apenas um elemento. É

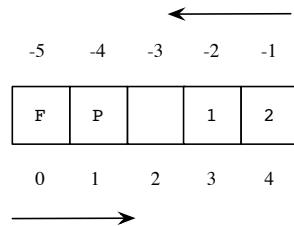


Figura 4.3: Valores dos índices de uma cadeia de caracteres.

importante notar que '2' não é o mesmo que o inteiro 2, é o carácter "2", como o mostra a seguinte interacção:

```
>>> id_fp = 'FP 12'
>>> id_fp
'FP 12'
>>> id_fp[0]
'F'
>>> id_fp[-1]
'2'
>>> id_fp[-1] == 2
False
```

Sobre as cadeias de caracteres podemos efectuar as operações indicadas na Tabela 4.3¹⁰ (note-se que estas operações são semelhantes às apresentadas na Tabela 4.1 e, consequentemente, todas estas operações são sobrecarregadas). A seguinte interacção mostra a utilização de algumas destas operações:

```
>>> cumprimento = 'bom dia!'
>>> cumprimento[0:3]
'bom'
>>> 'ola ' + cumprimento
'ola bom dia!'
>>> cumprimento * 3
'bom dia!bom dia!bom dia!'
>>> 'z' in cumprimento
False
```

¹⁰Nesta tabela, “Universal”, designa qualquer tipo.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$s_1 + s_2$	Cadeias de caracteres	A concatenação das cadeias de caracteres s_1 e s_2 .
$s * i$	Cadeia de caracteres e inteiro	A repetição i vezes da cadeia de caracteres s .
$s[i_1:i_2]$	Cadeia de caracteres e inteiros	A sub-cadeia de caracteres de s entre os índices i_1 e $i_2 - 1$.
$e \text{ in } s$	Cadeias de caracteres	True se o elemento e pertence à cadeia de caracteres s ; False em caso contrário.
$e \text{ not in } s$	Cadeias de caracteres	A negação do resultado da operação $e \text{ in } s$.
$\text{len}(s)$	Cadeia de caracteres	O número de elementos da cadeia de caracteres s .
$\text{str}(a)$	Universal	Transforma o seu argumento numa cadeia de caracteres.

Tabela 4.3: Algumas operações embutidas sobre cadeias de caracteres.

```
>>> 'ab' in 'abcd'
True
>>> len(cumprimento)
8
>>> str((1, 2))
'(1, 2)'
>>> str(2)
'2'
```

Como exemplo de utilização das operações existentes sobre cadeias de caracteres, a seguinte função recebe duas cadeias de caracteres e devolve os caracteres da primeira cadeia que também existem na segunda:

```
def simbolos_comum(s1, s2):
    # a variável s_comum contém os símbolos em comum de s1 e s2
    s_comum = ''      # s_comum é a cadeia de caracteres vazia
    for s in s1:
        if s in s2:
            s_comum = s_comum + s
    return s_comum
```

Com este programa, obtemos a interacção:

```
f1 = 'Fundamentos da programação'
f2 = 'Álgebra linear'
simbolos_comum(f1, f2)
'naen a rgraa'
```

Se a cadeia de caracteres `s1` contiver caracteres repetidos e estes caracteres existirem em `s2`, a função `simbolos_comum` apresenta um resultado com caracteres repetidos como o mostra a interacção anterior. Podemos modificar a nossa função de modo a que esta não apresente caracteres repetidos do seguinte modo:

```
def simbolos_comum_2(s1, s2):
    # a variável s_comum contém os símbolos em comum de s1 e s2
    s_comum = ''
    for s in s1:
        if s in s2 and not s in s_comum:
            s_comum = s_comum + s
    return s_comum
```

A qual permite originar a interacção:

```
>>> f1, f2 = 'Fundamentos de programação', 'Álgebra linear'
>>> simbolos_comum_2(f1, f2)
'nae rg'
```

Os caracteres são representados dentro do computador associados a um código numérico. Embora tenham sido concebidos vários códigos para a representação de caracteres, os computadores modernos são baseados no código ASCII (American Standard Code for Information Interchange)¹¹, o qual foi concebido para representar os caracteres da língua inglesa. Por exemplo, as letras maiúsculas são representadas em ASCII pelos inteiros entre 65 a 90. O código ASCII inclui a definição de 128 caracteres, 33 dos quais correspondem a caracteres de controle (muitos dos quais são actualmente obsoletos) e 95 caracteres visíveis, os

¹¹O código ASCII foi publicado pela primeira vez em 1963, tendo sofrido revisões ao longo dos anos, a última das quais em 1986.

quais são apresentados na Tabela 4.4. O problema principal associado ao código ASCII corresponde ao facto deste apenas abordar a representação dos caracteres existentes na língua inglesa. Para lidar com a representação de outros caracteres, por exemplo caracteres acentuados, foi desenvolvida uma representação, conhecida por *Unicode*,¹² a qual permite a representação dos caracteres de quase todas as linguagens escritas. Por exemplo, o carácter acentuado “á” corresponde ao código 227 e o símbolo do Euro ao código 8364. O código ASCII está contido no *Unicode*. O Python utiliza o *Unicode*.

Associado à representação de caracteres, o Python fornece duas funções embutidas, `ord` que recebe um carácter (sob a forma de uma cadeia de caracteres com apenas um elemento) e devolve o código decimal que o representa e a função `chr` que recebe um número inteiro positivo e devolve o carácter (sob a forma de uma cadeia de caracteres com apenas um elemento) representado por esse número. Por exemplo,

```
>>> ord('R')
82
>>> chr(125)
'}'
```

O código usado na representação de caracteres introduz uma ordem total entre os caracteres. O Python permite utilizar os operadores relacionais apresentados na Tabela 2.7 para comparar quer caracteres quer cadeias de caracteres. Com esta utilização, o operador “<” lê-se “aparece antes” e o operador “>” lê-se “aparece depois”. Assim, podemos gerar a seguinte interacção:

```
>>> 'a' < 'b'
True
>>> 'a' > 'A'
True
>>> '<' > '<'
False
>>> 'abc' > 'adg'
False
```

¹²As ideias iniciais para o desenvolvimento do *Unicode* foram lançadas em 1987 por Joe Becker da Xerox e por Lee Collins e Mark Davis da Apple, tendo sido publicados pela primeira vez em 1991.

Carácter	Representação decimal	Carácter	Representação decimal	Carácter	Representação decimal
!	32	©	64	'	96
"	33	À	65	à	97
#	34	À	66	à	98
\$	35	À	67	à	99
%	36	À	68	à	100
&	37	À	69	à	101
,	38	À	70	à	102
(39	À	71	à	103
)	40	À	72	à	104
*	41	À	73	à	105
+	42	À	74	à	106
,	43	À	75	à	107
-	44	À	76	à	108
.	45	À	77	à	109
/	46	À	78	à	110
0	47	À	79	à	111
1	48	À	80	à	112
2	49	À	81	à	113
3	50	À	82	à	114
4	51	À	83	à	115
5	52	À	84	à	116
6	53	À	85	à	117
7	54	À	86	à	118
8	55	À	87	à	119
9	56	À	88	à	120
:	57	À	89	à	121
;	58	À	90	à	122
<	59	À	91	à	123
=	60	À	92	à	124
>	61	À	93	à	125
	62	À	94	à	126

Tabela 4.4: Caracteres ASCII visíveis.

```
'abc' < 'abcd'
```

```
True
```

Utilizando a representação de caracteres, vamos escrever um programa que recebe uma mensagem (uma cadeia de caracteres) e codifica ou descodifica essa mensagem, utilizando uma cifra de substituição. Uma mensagem é codificada através de uma *cifra de substituição*, substituindo cada uma das suas letras por outra letra, de acordo com um certo padrão. A primeira utilização de uma cifra de substituição foi feita por Julio César (100–44 a.C.) durante as Guerras Gálicas. Uma variante da cifra de substituição foi também utilizada pela rainha Maria Stuart da Escócia (1542–1587) para conspirar contra a sua prima, a rainha Isabel I de Inglaterra (1533–1603), tendo a decifração deste código levado à sua execução¹³. Usando uma cifra de substituição (correspondente a uma cifra de substituição simples), cada letra da mensagem é substituída pela letra que está um certo número de posições, o *deslocamento*, à sua direita no alfabeto. Assim, se o deslocamento for 5, A será substituída por F, B por G e assim sucessivamente. Com esta cifra, as cinco letras do final do alfabeto serão substituídas pelas cinco letras no início do alfabeto, como se este correspondesse a um anel. Assim, V será substituída por A, W por B e assim sucessivamente. Com um deslocamento de 5, originamos a seguinte correspondência entre as letras do alfabeto:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
F G H I J K L M N O P Q R S T U V W X Y Z A B C D E

Na nossa cifra apenas consideramos letras maiúsculas, sendo qualquer símbolo que não corresponda a uma letra maiúscula substituído por ?. Os espaços entre as palavras mantêm-se como espaços.

A função de codificação recebe uma frase por codificar e o deslocamento (representado pela variável *n*). Para fazer a codificação de cada símbolo da nossa mensagem, por exemplo representado pela variável *c*, se esse símbolo corresponder a uma letra maiúscula, calculamos o seu código ASCII, dado por *ord(c)*, calculamos a distância a que essa letra se encontra em relação ao início do alfabeto, *ord(c) - ord('A')*, somamos *n* a essa distância e determinamos qual a distância a que a letra resultante está do início do alfabeto, (*ord(c) -*

¹³O livro [Sing, 1999] apresenta uma história interessante sobre os vários tipos de métodos para cifrar mensagens.

`ord('A') + n) % 26`, e somamos essa distância à posição da letra “A”, `ord('A') + (ord(c) - ord('A') + n) % 26`, calculando finalmente qual a letra que substitui a letra na mensagem original. Essa letra é adicionada ao final da mensagem codificada que estamos a construir.

A seguinte função recebe uma frase por codificar, `f`, o deslocamento a usar na cifra de substituição, `n`, e devolve a correspondente mensagem codificada. Para guardar a mensagem codificada, a função usa a variável `codif` que é inicializada com a cadeia de caracteres vazia.

```
def codifica(f, n):
    codif = ''
    for c in f:
        if c == ' ':
            codif = codif + c
        elif 'A' <= c <= 'Z':
            codif = codif + \
                chr(ord('A') + (ord(c) - ord('A') + n) % 26)
        else:
            codif = codif + '?'
    return codif
```

Usando um raciocínio semelhante, podemos escrever a seguinte função que recebe uma mensagem codificada e o deslocamento a usar na cifra de substituição e devolve a mensagem original. Um símbolo que não corresponda a uma letra maiúscula é substituído por um ponto.

```
def descodifica(f, n):
    desc = ''
    for c in f:
        if c == ' ':
            desc = desc + c
        elif 'A' <= c <= 'Z':
            desc = desc + \
                chr(ord('A') + (ord(c) - ord('A') - n) % 26)
        else:
            desc = desc + '.'
    return desc
```

Estamos agora em condições de escrever o nosso programa. Este programa utiliza as funções `codifica` e `descodifica` que acabámos de definir. O programa solicita ao utilizador que forneça uma mensagem e qual o tipo de operação a realizar (codificação ou descodificação), após o que efectua a operação solicitada e mostra o resultado obtido. O nosso programa usa um deslocamento de 5 na cifra de substituição.

```
def codificador():

    def codifica(f, n):
        codif = ''
        for c in f:
            if c == ' ':
                codif = codif + c
            elif 'A' <= c <= 'Z':
                codif = codif + \
                    chr(ord('A') + (ord(c) - ord('A') + n) % 26)
            else:
                codif = codif + '?'
        return codif

    def descodifica(f, n):
        desc = ''
        for c in f:
            if c == ' ':
                desc = desc + c
            elif 'A' <= c <= 'Z':
                desc = desc + \
                    chr(ord('A') + (ord(c) - ord('A') - n) % 26)
            else:
                desc = desc + '.'
        return desc

    original = input('Introduza uma mensagem\n-> ')
    tipo = input('C para codificar ou D para descodificar\n-> ')

    if tipo == 'C' :
```

```

print('A mensagem codificada é:\n',
      codifica(original, 5))
elif tipo == 'D' :
    print('A mensagem descodificada é:\n',
          descodifica(original, 5))
else :
    print('Oops .. não sei o que fazer')

```

A seguinte interacção mostra o funcionamento do nosso programa. Repare-se que o programa corresponde a uma função sem argumentos, chamada **codificador**, a qual não devolve um valor mas sim executa uma sequência de acções.

```

codificador()
Introduza uma mensagem
-> VAMOS CIFRAR ESTA MENSAGEM PARA QUE NINGUEM ENTENDA
C para codificar ou D para descodificar
-> C
A mensagem codificada é:
AFRTX HNKWFW JXYF RJSXFLJR UFWF VZJ SNSLZJR JSYJSIF
codificador()
Introduza uma mensagem
-> AFRTX HNKWFW JXYF RJSXFLJR UFWF VZJ SNSLZJR JSYJSIF
C para codificar ou D para descodificar
-> D
A mensagem descodificada é:
VAMOS CIFRAR ESTA MENSAGEM PARA QUE NINGUEM ENTENDA

```

4.4 Notas finais

Este foi o primeiro capítulo em que abordámos o estudo de tipos estruturados de informação. Considerámos dois tipos, os tuplos e as cadeias de caracteres, os quais partilham as propriedades de corresponderem a sequências de elementos e de serem tipos imutáveis. O acesso aos elementos destes tipos é feito recorrendo a um índice correspondendo a um valor inteiro. Introduzimos uma nova instrução de repetição, a instrução **for**, que corresponde a um ciclo contado.

4.5 Exercícios

1. Escreva em Python a função `duplica` que recebe um tuplo e tem como valor um tuplo idêntico ao original, mas em que cada elemento está repetido. Por exemplo,

```
>>> duplica((1, 2, 3))  
(1, 1, 2, 2, 3, 3)
```

2. Escreva uma função em Python com o nome `conta_menores` que recebe um tuplo contendo números inteiros e um número inteiro e que devolve o número de elementos do tuplo que são menores do que esse inteiro. Por exemplo,

```
>>> conta_menores((3, 4, 5, 6, 7), 5)  
2  
>>> conta_menores((3, 4, 5, 6, 7), 2)  
0
```

3. Escreva uma função em Python chamada `maior_elemento` que recebe um tuplo contendo números inteiros, e devolve o maior elemento do tuplo. Por exemplo,

```
>>> maior_elemento((2, 4, 23, 76, 3))  
76
```

4. Defina a função `juntos` que recebe um tuplo contendo inteiros e tem como valor o número de elementos iguais adjacentes. Por exemplo,

```
>>> juntos((1, 2, 2, 3, 4, 4))  
2  
>>> juntos((1, 2, 2, 3, 4))  
1
```

5. Defina uma função, `junta_ordenados`, que recebe dois tuplos contendo inteiros, ordenados por ordem crescente. e devolve um tuplo também ordenado com os elementos dos dois tuplos. Por exemplo,

```
>>> junta_ordenados((2, 34, 200, 210), (1, 23))  
(1, 2, 23, 34, 200, 210)
```

6. Escreva em Python uma função, chamada `soma_els_atomicos`, que recebe como argumento um tuplo, cujos elementos podem ser outros tuplos, e que devolve a soma dos elementos correspondentes a tipos elementares de informação que existem no tuplo original. Por exemplo,

```
>>> soma_els_atomicos((3, (((((6, (7, )), ), ), ), ), 2, 1))
19
>>> soma_els_atomicos(((((),),),))
0
```

7. A sequência de Racamán, tal como descrita em [Bellos, 2012],

0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, 22, 10, 23, 9, 24, ...

é uma sequência de números inteiros não negativos, definida do seguinte modo: (1) o primeiro termo da sequência é zero; (2) para calcular o n -ésimo termo, verifica-se se o termo anterior é maior do que n e se o resultado de subtrair n ao termo anterior ainda não apareceu na sequência, neste caso o n -ésimo termo é dado pela subtração entre o $(n - 1)$ -ésimo termo e n ; em caso contrário o n -ésimo termo é dado pela soma do $(n - 1)$ -ésimo termo com n . Ou seja,

$$r(n) = \begin{cases} 0 & \text{se } n = 0 \\ r(n - 1) - n & \text{se } r(n - 1) > n \wedge (r(n - 1) - n) \notin \{r(i) : 1 < i < n\} \\ r(n - 1) + n & \text{em caso contrário} \end{cases}$$

Escreva uma função em Python que recebe um inteiro positivo, n , e devolve um tuplo contendo os n primeiros elementos da sequência de Racamán. Por exemplo:

```
>>> seq_racaman(15)
(0, 1, 3, 6, 2, 7, 13, 20, 12, 21, 11, 22, 10, 23, 9)
```

8. Considere a gramática em notação BNF, apresentada no Exercício 4 do Capítulo 1:

```
<idt> ::= <letras> <numeros>
<letras> ::= <letra> |
             <letra> <letras>
```

```
⟨numeros⟩ ::= ⟨num⟩ |  
           ⟨num⟩ ⟨numeros⟩  
⟨letra⟩ ::= A | B | C | D  
⟨num⟩ ::= 1 | 2 | 3 | 4
```

Escreva uma função em Python, chamada `reconhece`, que recebe como argumento uma cadeia de caracteres e devolve *verdadeiro* se o seu argumento corresponde a uma frase da linguagem definida pela gramática e *falso* em caso contrário. Por exemplo,

```
>>> reconhece('A1')  
True  
>>> reconhece('ABBBBCDDDD23311')  
True  
>>> reconhece('ABC12C')  
False
```

Capítulo 5

Listas

*'Who are you?' said the Caterpillar.
This was not an encouraging opening for a conversation.
Alice replied, rather shyly, 'I – I hardly know, Sir, just at
present – at least I know who I was when I got up this
morning, but I think I must have been changed several
times since then.'*

Lewis Carroll, *Alice's Adventures in Wonderland*

Neste capítulo abordamos um novo tipo estruturado de informação, a lista. Uma *lista* é uma sequência de elementos de qualquer tipo. Esta definição é semelhante à de um tuplo. Contudo, em oposição aos tuplos, as listas são tipos *mutáveis*, no sentido de que podemos alterar destrutivamente os seus elementos. Na maioria das linguagens de programação existem tipos semelhantes às listas existentes em Python, sendo conhecidos por *vectores* ou por *tabelas*¹.

5.1 Listas em Python

Uma *lista*, em Python designada por `list`, é uma sequência de elementos. Em Python, a representação externa de uma lista é definida sintacticamente pelas seguintes expressões em notação BNF²:

¹Em inglês “array”.

²Devemos notar que esta definição não está completa e que a definição de `(dicionário)` é apresentada no Capítulo 9.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$l_1 + l_2$	Listas	A concatenação das listas l_1 e l_2 .
$l * i$	Lista e inteiro	A repetição i vezes da lista l .
$l[i_1:i_2]$	Lista e inteiros	A sub-lista de l entre os índices i_1 e $i_2 - 1$.
$\text{del}(els)$	Lista e inteiro(s)	Em que els pode ser da forma $l[i]$ ou $l[i_1:i_2]$. Remove os elemento(s) especificado(s) da lista l .
$e \text{ in } l$	Universal e lista	True se o elemento e pertence à lista l ; False em caso contrário.
$e \text{ not in } l$	Universal e lista	A negação do resultado da operação $e \text{ in } l$.
$\text{list}(a)$	Tuplo ou dicionário ou cadeia de caracteres	Transforma o seu argumento numa lista. Se não forem fornecidos argumentos, o seu valor é a lista vazia.
$\text{len}(l)$	Lista	O número de elementos da lista l .

Tabela 5.1: Operações embutidas sobre listas.

```

⟨lista⟩ ::= [] |
           [⟨elementos⟩]

⟨elementos⟩ ::= ⟨elemento⟩ |
                  ⟨elemento⟩, ⟨elementos⟩

⟨elemento⟩ ::= ⟨expressão⟩ |
                  ⟨tuplo⟩ |
                  ⟨lista⟩ |
                  ⟨dicionário⟩
  
```

A lista [] tem o nome de *lista vazia*. As seguintes entidades representam listas em Python [1, 2, 3], [2, (1, 2)], ['a']. Em oposição aos tuplos, uma lista de um elemento não contém a vírgula. Tal como no caso dos tuplos, os elementos de uma lista podem ser, por sua vez, outras listas, pelo que a seguinte entidade é uma lista em Python [1, [2], [[3]]].

Sobre as listas podemos efectuar as operações apresentadas na Tabela 5.1. Note-se a semelhança entre estas operações e as operações sobre tuplos e as operações sobre cadeias de caracteres, apresentadas, respectivamente, nas tabelas 4.1 e 4.3, sendo a excepção a operação `del` que existe para listas e que não existe nem para tuplos nem para cadeias de caracteres.

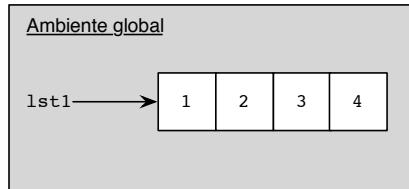
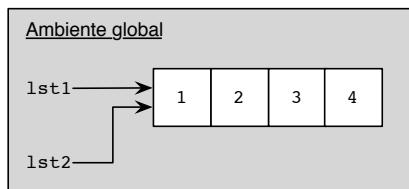
A seguinte interacção mostra a utilização de algumas operações sobre listas:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [[4, 5]]
>>> lst = lst1 + lst2
>>> lst
[1, 2, 3, [4, 5]]
>>> len(lst)
4
>>> lst[3]
[4, 5]
>>> lst[3][0]
4
>>> lst[2] = 'a'
>>> lst
[1, 2, 'a', [4, 5]]
>>> del(lst[1])
>>> lst
[1, 'a', [4, 5]]
>>> del(lst[1:])
>>> lst
[1]
```

Sendo as listas entidades mutáveis, podemos alterar qualquer dos seus elementos. Na interacção anterior, atribuímos a cadeia de caracteres 'a' ao elemento da lista `lst` com índice 2 (`lst[2] = 'a'`), removemos da lista `lst` o elemento com índice 1 (`del(lst[1])`), após o que removemos da lista `lst` todos os elementos com índice igual ou superior a 1 (`del(lst[1:])`).

Consideremos agora a seguinte interacção:

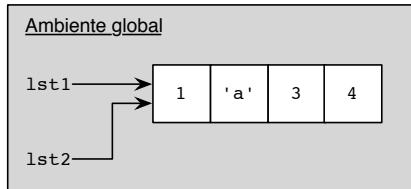
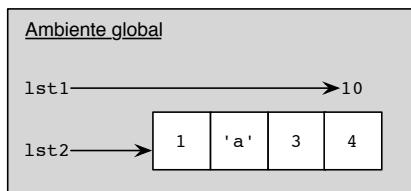
```
>>> lst1 = [1, 2, 3, 4]
>>> lst2 = lst1
>>> lst1
[1, 2, 3, 4]
>>> lst2
[1, 2, 3, 4]
>>> lst2[1] = 'a'
>>> lst2
[1, 'a', 3, 4]
```

Figura 5.1: Ambiente após a definição da lista `lst1`.Figura 5.2: Ambiente após a atribuição `lst2 = lst1`.

```
>>> lst1
[1, 'a', 3, 4]
```

Nesta interacção, começamos por definir a lista `lst1`, após o que definimos a lista `lst2` como sendo igual a `lst1`. Ao alterarmos a lista `lst2` estamos indirec-tamente a alterar a lista `lst1`. Este comportamento, aparentemente estranho, é explicado quando consideramos o ambiente criado por esta interacção. Ao criar a lista `lst1`, o Python dá origem ao ambiente apresentado na Figura 5.1 (usamos uma representação esquemática para listas que é semelhante à usada para os tuplos). A instrução `lst2 = lst1`, define um novo nome, `lst2`, como sendo o valor de `lst1`. Na realidade, a semântica da instrução de atribuição es-pefica que ao executar a instrução `lst2 = lst1`, o Python começa por avaliar a expressão `lst1` (um nome), sendo o seu valor a entidade associada ao nome (a lista `[1, 2, 3, 4]`), após o que cria ao nome `lst2`, associando-o ao valor desta expressão. Esta instrução origina o ambiente apresentado na Figura 5.2. As lis-tas `lst1` e `lst2` correspondem a *pseudónimos*³ para a mesma entidade. Assim, ao alterarmos uma delas estamos implicitamente a alterar a outra (Figura 5.3). Esta situação não se verifica com tuplos nem com cadeias de caracteres pois estes são estruturas imutáveis.

³Do inglês “alias”.

Figura 5.3: Ambiente após a alteração de `lst1`.Figura 5.4: Ambiente após nova alteração de `lst1`.

Suponhamos agora que continuávamos a interacção anterior do seguinte modo:

```
>>> lst1 = 10
>>> lst1
10
>>> lst2
[1, 'a', 3, 4]
```

Esta nova interacção dá origem ao ambiente apresentado na Figura 5.4. A variável `lst1` passa a estar associada ao inteiro 10 e o valor da variável `lst2` mantém-se inalterado. A segunda parte deste exemplo mostra a diferença entre alterar um elemento de uma lista que é partilhada por várias variáveis e alterar uma dessas variáveis.

5.2 Métodos de passagem de parâmetros

Com base no exemplo anterior estamos agora em condições de analisar de um modo mais detalhado o processo de comunicação com funções. Como sabemos, quando uma função é avaliada (ou chamada) estabelece-se uma correspondência entre os parâmetros concretos e os parâmetros formais, associação essa que é

feita com base na posição que os parâmetros ocupam na lista de parâmetros. O processo de ligação entre os parâmetros concretos e os parâmetros formais é denominado *método de passagem de parâmetros*. Existem vários métodos de passagem de parâmetros. Cada linguagem de programação utiliza um, ou vários, destes métodos para a comunicação com funções. O Python utiliza dois métodos de passagem de parâmetros, a passagem por valor e a passagem por referência.

5.2.1 Passagem por valor

O método de passagem de parâmetros em Python, quando lida com tipos elementares de informação, é a passagem por valor. Quando um parâmetro é passado por *valor*, o valor do parâmetro concreto é calculado (independentemente de ser uma constante, uma variável ou uma expressão mais complicada), e esse valor é associado com o parâmetro formal correspondente. Ou seja, utilizando passagem por valor, a função recebe o valor de cada um dos parâmetros e nenhuma informação adicional.

Um parâmetro formal em que seja utilizada a passagem por valor comporta-se, dentro do bloco associado à sua função, como um nome local que é inicializado com o início da avaliação da função.

Para exemplificar, consideremos a função `troca` definida do seguinte modo:

```
def troca(x, y):
    x, y = y, x # os valores são trocados
```

e a seguinte interacção que utiliza a função `troca`:

```
>>> x = 3
>>> y= 10
>>> troca(x, y)
>>> x
3
>>> y
10
```

As duas primeiras linhas desta interacção têm como efeito a criação dos nomes `x` e `y` no ambiente global como se mostra na Figura 5.5. Quando o Python

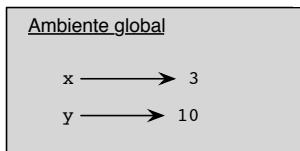


Figura 5.5: Ambiente global antes da invocação de `troca`.

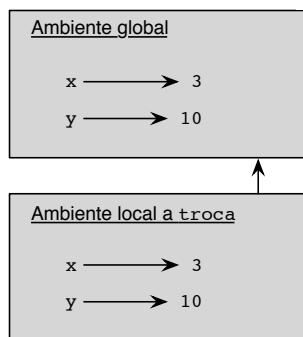


Figura 5.6: Ambiente local criado com a invocação de `troca`.

invoca a função `troca`, avalia os parâmetros concretos e associa os parâmetros concretos aos parâmetros formais da função `troca`, criando o ambiente local que se mostra na Figura 5.6.

A instrução de atribuição `x, y = y, x` executada pela função `troca`, altera o ambiente local como se mostra na Figura 5.7. Com efeito, recorde-se da Secção 2.3 que ao encontrar esta instrução, o Python avalia as expressões à direita do símbolo “=”, as quais têm os valores 10 e 3, respectivamente, após o que atribui estes valores às variáveis `x` e `y`, respectivamente. Esta instrução tem pois o efeito de trocar os valores das variáveis `x` e `y`. Os valores dos nomes locais `x` e `y` são alterados, mas isso não vai afectar os nomes `x` e `y` que existiam antes da avaliação da função `troca`.

Quando a função `troca` termina a sua execução o ambiente que lhe está associado desaparece, voltando-se à situação apresentada na Figura 5.5. Ou seja, quando se utiliza a passagem por valor, a única ligação entre os parâmetros

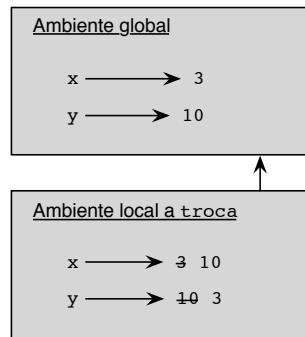


Figura 5.7: Ambientes após a execução da instrução de atribuição.

concretos e os parâmetros formais é uma associação unidireccional de valores. É *unidireccional* porque é feita do ponto de chamada para a função.

5.2.2 Passagem por referência

Quando um parâmetro é passado por *referência*, o que é associado ao parâmetro formal correspondente não é o valor do parâmetro concreto, mas sim a localização na memória do computador que contém o seu valor. Utilizando passagem por referência, os parâmetros formais e os parâmetros concretos vão *partilhar* o mesmo local (dentro da memória do computador) e, consequentemente, qualquer modificação feita aos parâmetros formais reflecte-se nos parâmetros concretos. Um parâmetro formal em que seja utilizada a passagem por referência corresponde à *mesma variável* que o parâmetro concreto correspondente, apenas, eventualmente, com outro nome.

O Python utiliza a passagem por referência sempre que um parâmetro concreto corresponde a uma estrutura de informação. Consideremos agora a função `troca_2` que recebe como argumentos uma lista e dois inteiros e que troca os elementos da lista cujos índices correspondem a esses inteiros.

```
def troca_2(lst, i1, i2):
    lst[i1], lst[i2] = lst[i2], lst[i1] # os valores são trocados
```

Com esta função podemos originar a seguinte interacção:

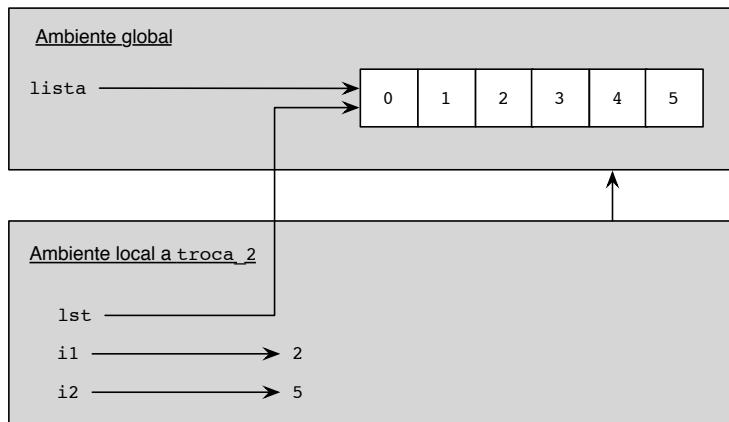


Figura 5.8: Ambientes após o início da execução de `troca_2`.

```
>>> lista = [0, 1, 2, 3, 4, 5]
>>> troca_2(lista, 2, 5)
>>> lista
[0, 1, 5, 3, 4, 2]
```

Neste caso, o parâmetro concreto `lst` partilha o mesmo espaço de memória que a variável global `lista`, pelo que qualquer alteração a `lst` reflecte-se na variável `lista`. Na Figura 5.8 mostramos os ambientes criados no início da execução da função `troca_2`.

5.3 O Crivo de Eratóstenes

O *Crivo de Eratóstenes* é um algoritmo para calcular números primos que, segundo a tradição, foi criado pelo matemático grego Eratóstenes (c. 285–194 a.C.), o terceiro bibliotecário chefe da Biblioteca de Alexandria. Para um dado inteiro positivo n , o algoritmo calcula todos os números primos inferiores a n . Para isso, começa por criar uma lista com todos os inteiros positivos de 2 a n e selecciona o primeiro elemento da lista, o número 2. Enquanto o número seleccionado não for maior que \sqrt{n} executam-se as seguintes acções: (a) removem-se da lista todos os múltiplos do número seleccionado; (b) passa-se ao número seguinte na lista. No final do algoritmo, quando o número seleccionado for superior a \sqrt{n} , a lista apenas contém números primos.

Vamos agora escrever uma função, `crivo` que recebe um inteiro positivo n e calcula a lista de números primos inferiores a n de acordo com o Crivo de Eratóstenes.

A nossa função deve começar por criar a lista com os inteiros entre 2 e n , após o que percorre a lista para os elementos inferiores ou iguais a \sqrt{n} , removendo da lista todos os múltiplos desse elemento. Partindo do princípio da existência de uma função chamada `remove_multiplos` que recebe uma lista e um número e modifica essa lista, removendo todos os múltiplos desse número, a função `crivo` será:

```
def crivo(n):

    from math import sqrt

    # Criação da lista com inteiros de 2 a n
    lista = []
    for i in range(2, n + 1):
        lista = lista + [i]

    # Remoção de elementos seleccionados da lista
    i = 0
    while lista[i] <= sqrt(n):
        remove_multiplos(lista, lista[i])
        i = i + 1
    return lista
```

É importante perceber a razão de não termos recorrido a um ciclo `for` na remoção dos elementos da lista. Um ciclo `for` calcula de antemão o número de vezes que o ciclo será executado, fazendo depois esse número de passagens pelo ciclo com o valor da variável de controlo seguindo uma progressão aritmética. Este é o ciclo ideal para fazer o processamento de tuplos e de cadeias de caracteres e, tipicamente, é também o ciclo ideal para fazer o processamento de listas. No entanto, no nosso exemplo, o número de elementos da lista vai diminuindo à medida que o processamento decorre, e daí a necessidade de fazer o controlo da execução do ciclo de um outro modo. Sabemos que temos que processar os elementos da lista não superiores a \sqrt{n} e é este o mecanismo de controle que usámos no ciclo `while`.

Vamos agora concentrar-nos no desenvolvimento da função `remove_multiplos`. Apesar da discussão que acabámos de apresentar em relação à utilização de uma instrução `while`, nesta função vamos utilizar um ciclo `for` que percorre os elementos da lista, do final da lista para o início, deste modo podemos percorrer todos os elementos relevantes da lista, sem termos que nos preocupar com eventuais alterações dos índices, originadas pela remoção de elementos da lista. Para exemplificar o nosso raciocínio, suponhamos que estamos a percorrer a lista [2, 3, 4, 5, 6, 7, 8, 9, 10], começando no seu último elemento, cujo índice é 8, filtrando os múltiplos de 2. O primeiro elemento considerado é `lst[8]`, cujo valor é 10, pelo que este elemento é removido da lista, o próximo elemento a considerar é `lst[7]`, cujo valor é 9, pelo que, não sendo um múltiplo de 2, este mantém-se na lista, segue-se o elemento `lst[6]`, que é removido da lista e assim sucessivamente. Resta-nos decidir em que elemento da lista devemos parar o nosso processamento. No nosso exemplo, não devemos processar o elemento `lst[0]`, pois sendo 2, será erradamente retirado da lista. Para isso, usamos uma função, `pos`, que recebe a lista e o elemento a ser processado e devolve a posição deste elemento na lista original. É esta posição que corresponderá ao fim do ciclo.

```
def remove_multiplos(lst, n):
    for i in range(len(lst)-1, pos(lst, n)+1, -1):
        if lst[i] % n == 0:
            del(lst[i])
```

A função `pos` é trivialmente definida do seguinte modo:

```
def pos(lst, n):
    for i in range(n):
        if lst[i] == n:
            return i
```

Podemos agora apresentar a função `crivo` de um modo completo, usando a estrutura de blocos:

```
def crivo(n):
    def remove_multiplos(lst, n):
```

```

def pos(lst, n):
    for i in range(n):
        if lst[i] == n:
            return i

    for i in range(len(lst)-1, pos(lst, n)+1, -1):
        if lst[i] % n == 0:
            del(lst[i])

from math import sqrt

lista = []
for i in range(2, n + 1):
    lista = lista + [i]

i = 0
while lista[i] <= sqrt(n):
    remove_multiplos(lista, lista[i])
    i = i + 1
return lista

```

Com esta função geramos a interacção:

```

>>> crivo(60)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59]

```

5.4 Algoritmos de procura

'Just look along the road, and tell me if you can see either of them.'
'I see nobody on the road,' said Alice.
'I only wish I had such eyes,' the King remarked in a fretful tone.
'To be able to see Nobody! And at that distance, too! Why, it's as much as I can do to see real people, by this light!'

Lewis Carroll, *Through the Looking Glass*

A procura de informação é uma das nossas actividades quotidianas. Procuramos palavras no Google, palavras em dicionários, livros em bibliotecas, produtos em

supermercados, etc. Na função `pos` (apresentada na página 151) procurámos a posição de um número numa lista. Com o aumento constante da quantidade de informação armazenada por computadores, é natural que a procura de informação se tenha tornado uma das actividades preponderantes num sistema computacional. Embora a localização de um dado elemento entre um conjunto de elementos pareça ser muito simples, o desenvolvimento de programas eficientes de procura levanta muitos problemas. Como exemplo, suponhamos que procurávamos a palavra “Python” no Google, a nossa procura devolve cerca de 198 milhões de resultados, tendo a procura demorado 0.26 segundos. Imagine-se quanto demoraria esta procura se o Google não utilizasse métodos de procura altamente sofisticados.

A procura de informação é frequentemente executada recorrendo a listas. Nesta secção, apresentamos dois métodos para procurar um elemento numa lista, a procura sequencial e a procura binária. Para estes dois tipos de procura, iremos desenvolver uma função com o nome `procura` que recebe como argumentos uma lista (`lst`) e um elemento (a que chamamos `chave`). Esta função devolve um inteiro positivo correspondente ao índice do elemento da lista cujo valor é igual a `chave` (estamos a partir do pressuposto que a lista `lst` não tem elementos repetidos). Vamos também convencionar que se a lista não contiver nenhum elemento igual a `chave`, a função devolve `-1`.

5.4.1 Procura sequencial

Uma das maneiras mais simples de procurar um dado elemento numa lista consiste em começar no primeiro elemento da lista e comparar sucessivamente o elemento procurado com o elemento na lista. Este processo é repetido até que o elemento seja encontrado ou o fim da lista seja atingido. Este método de procura é chamado *procura sequencial*, ou *procura linear*. Apresentamos a função `procura` utilizando a procura sequencial:

```
def procura(lst, chave):
    for i in range(len(lst)):
        if lst[i] == chave:
            return i
    return -1
```

Com a qual podemos obter a interacção:

```
>>> lst = [2, 3, 1, -4, 12, 9]
>>> procura(lst, 1)
2
>>> procura(lst, 5)
-1
```

5.4.2 Procura binária

A procura sequencial é muito simples, mas pode exigir a inspecção de todos os elementos da lista, o que acontece sempre que o elemento que estamos a procurar não se encontra na lista.

A *procura binária* é uma alternativa, mais eficiente, à procura sequencial, exigindo, contudo, que os elementos sobre os quais a procura está a ser efectuada se encontrem ordenados. Utilizando a procura binária, consideramos em primeiro lugar o elemento que se encontra no meio da lista:

1. Se este elemento é maior do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na segunda metade da lista. Repetimos então o processo da procura binária para a primeira metade da lista.
2. Se o elemento no meio da lista é menor do que o elemento que estamos a procurar então podemos garantir que o elemento que estamos a procurar não se encontra na primeira metade da lista. Repetimos então o processo da procura binária para a segunda metade da lista.
3. Se o elemento no meio da lista for igual ao elemento que estamos a procurar, então a procura termina.

Note-se que, em cada passo, a procura binária reduz o número de elementos a considerar para metade (e daí o seu nome).

A seguinte função utiliza a procura binária. Esta função pressupõe que a lista `lst` está ordenada. As variáveis `linf` e `lsup` representam, respectivamente, o menor e o maior índice da gama dos elementos que estamos a considerar.

```

def procura(lst, chave):
    linf = 0
    lsup = len(lst) - 1

    while linf <= lsup:
        meio = (linf + lsup) // 2
        if chave == lst[meio]:
            return meio
        elif chave > lst[meio]:
            linf = meio + 1
        else:
            lsup = meio - 1
    return -1

```

A procura binária exige que mantenhamos uma lista ordenada contendo os elementos em que queremos efectuar a procura, consequentemente, o custo computacional da inserção de um novo elemento na lista é maior se usarmos este método do que se usarmos a procura sequencial. Contudo, como as procura são normalmente mais frequentes do que as inserções, é preferível utilizar uma lista ordenada.

5.5 Algoritmos de ordenação

Um conjunto de elementos é normalmente ordenado para facilitar a procura. Na nossa vida quotidiana, ordenamos as coisas para tornar a procura mais fácil, e não porque somos arrumados. Em programação, a utilização de listas ordenadas permite a escrita de algoritmos de procura mais eficientes, por exemplo, a procura binária em lugar da procura sequencial. Nesta secção, abordamos o estudo de alguns algoritmos para ordenar os valores contidos numa lista.

Os algoritmos de ordenação podem ser divididos em dois grandes grupos, os algoritmos de ordenação interna e os algoritmos de ordenação externa. Os algoritmos de *ordenação interna* ordenam um conjunto de elementos que estão simultaneamente armazenados em memória, por exemplo, numa lista. Os algoritmos de *ordenação externa* ordenam elementos que, devido à sua quantidade, não podem estar simultaneamente em memória, estando parte deles armazena-

dos algures no computador (num ficheiro, para ser mais preciso). Neste segundo caso, em cada momento apenas estão em memória parte dos elementos a ordenar. Neste livro apenas consideramos algoritmos de ordenação interna.

Na nossa apresentação dos algoritmos de ordenação, vamos supor que desejamos ordenar uma lista de números inteiros. O nosso programa de ordenação, `prog_ordena`, começa por ler os números a ordenar, `le_elementos`, números esses que são guardados numa lista. Em seguida, ordena os elementos, recorrendo à função `ordena` (a qual ainda não foi desenvolvida) e escreve-os por ordem crescente. A ordenação é efectuada pela função `ordena`, que recebe uma lista e retorna, por referência, a lista ordenada.

```
def prog_ordena():

    def le_elementos():
        print('Introduza os elementos a ordenar')
        el = input('separados por espaços\n')

        elementos = []
        i = 0

        # Ignora espaços em branco iniciais
        while i < len(el) and el[i] == ' ':
            i = i + 1

        while i < len(el): # Processa o resto da linha
            # Transforma um elemento lido num inteiro
            num = 0
            while i < len(el) and el[i] != ' ':
                num = num * 10 + eval(el[i])
                i = i + 1
            elementos = elementos + [num]

        # Ignora espaços em branco depois do número
        while i < len(el) and el[i] == ' ':
            i = i + 1
```

```

    return elementos

elementos = le_elementos()
print('Elementos fornecidos:', elementos)
ordena(elementos)
print('Elementos ordenados:', elementos)

```

Antes de abordar o desenvolvimento da função `ordena`, vamos considerar a função `le_elementos`. Esta função interage com o utilizador, solicitando a introdução dos elementos a ordenar, separados por espaços em branco, e devolve a lista contendo os inteiros fornecidos pelo utilizador. A função contempla a possibilidade do utilizador escrever qualquer número de espaços em branco antes ou depois de cada um dos elementos a ordenar. Após a leitura da cadeia de caracteres (`el`) contendo a sequência de elementos fornecidos pelo utilizador, a função deve processar essa cadeia de caracteres de modo apropriado. Numa primeira fase, a função ignora todos os espaços em branco iniciais existentes em `el`. Ao encontrar um símbolo que não corresponda ao espaço em branco, esta função tem que criar um número inteiro a partir de caracteres, utilizando o seguinte algoritmo: inicialmente o número a construir, `num`, toma o valor zero, enquanto a cadeia de caracteres contiver caracteres por tratar ($i < \text{len}(el)$) e o carácter lido não for o espaço em branco (`el[i] != ' '`), o número correspondente a esse carácter será adicionado como elemento das unidades ao número já formado (`num = num * 10 + eval(el[i])`) e o índice dos caracteres lidos é actualizado em uma unidade. Após a criação de um número, este é adicionado à lista dos elementos (`elementos = elementos + [num]`). Por exemplo, suponhamos que a cadeia de caracteres fornecida foi `'45 9854 23 654 8 1'`. Ao formar o número 45, o programa segue os seguintes passos: `num = 0`, `num = 0 * 10 + eval('4')` (obtendo 4), `num = 4 * 10 + eval('5')` (obtendo 45). Depois de obtido um número, os espaços em branco que o seguem são ignorados. É importante dizer que esta função não está preparada para lidar com dados fornecidos pelo utilizador que não sigam o formato indicado.

Com uma função de ordenação apropriada, o programa `prog_ordena` permite gerar a interacção:

```

>>> prog_ordena()
Introduza os inteiros a ordenar
separados por espaços

```

```
?      45 88775      665443 34 122 1      23
Elementos fornecidos: [45, 88775, 665443, 34, 122, 1, 23]
Elementos ordenados: [1, 23, 34, 45, 122, 88775, 665443]
```

Para apresentar os algoritmos de ordenação, vamos desenvolver várias versões de uma função chamada `ordena`. Esta função recebe como parâmetro uma lista contendo os elementos a ordenar (correspondendo ao parâmetro formal `lst`), a qual, após execução da função, contém os elementos ordenados. Cada versão que apresentamos desta função corresponde a um algoritmo de ordenação.

5.5.1 Ordenação por borbulhamento

O primeiro algoritmo de ordenação que vamos considerar é conhecido por ordenação *por borbulhamento*⁴. A ideia básica subjacente a este algoritmo consiste em percorrer os elementos a ordenar, comparando elementos adjacentes, trocando os pares de elementos que se encontram fora de ordem, ou seja, que não estão ordenados. De um modo geral, uma única passagem pela sequência de elementos não ordena a lista, pelo que é necessário efectuar várias passagens. A lista encontra-se ordenada quando se efectua uma passagem completa em que não é necessário trocar a ordem de nenhum elemento. A razão por que este método é conhecido por “ordenação por borbulhamento” provém do facto de os menores elementos da lista se movimentarem no sentido do início da lista, como bolhas que se libertam dentro de um recipiente com um líquido.

A seguinte função utiliza a ordenação por borbulhamento. Esta função utiliza a variável de tipo lógico, `nenhuma_troca`, cujo valor é `False` se, durante uma passagem pela lista, se efectua alguma troca de elementos, e tem o valor `True` em caso contrário. Esta variável é inicializada para `False` na segunda linha da função de modo a que o ciclo `while` seja executado.

Dado que em cada passagem pelo ciclo colocamos o maior elemento da lista na sua posição correcta⁵, cada passagem subsequente pelos elementos da lista considera um valor a menos e daí a razão da variável `maior_indice`, que contém o maior índice até ao qual a ordenação se processa.

```
def ordena(lst):
```

⁴Em inglês, “bubble sort”.

⁵Como exercício, o leitor deve convencer-se deste facto.

```

maior_indice = len(lst) - 1
nenhuma_troca = False

while not nenhuma_troca:
    nenhuma_troca = True
    for i in range(maior_indice):
        if lst[i] > lst[i+1]:
            lst[i], lst[i+1] = lst[i+1], lst[i]
            nenhuma_troca = False
    maior_indice = maior_indice - 1

```

5.5.2 Ordenação Shell

Uma variante da ordenação por borbulhamento, a ordenação *Shell*⁶ consiste em comparar e trocar, não os elementos adjacentes, mas sim os elementos separados por um certo intervalo. Após uma ordenação completa, do tipo borbulhamento, com um certo intervalo, esse intervalo é dividido ao meio e o processo repete-se com o novo intervalo. Este processo é repetido até que o intervalo seja 1 (correspondendo a uma ordenação por borbulhamento). Como intervalo inicial, considera-se metade do número de elementos a ordenar. A ordenação *Shell* é mais eficiente do que a ordenação por borbulhamento, porque as primeiras passagens que consideram apenas um subconjunto dos elementos a ordenar permitem uma arrumação grosseira dos elementos da lista e as últimas passagens, que consideram todos os elementos, já os encontram parcialmente ordenados. A seguinte função corresponde à ordenação *Shell*:

```

def ordena(lst):

    intervalo = len(lst) // 2

    while not intervalo == 0:
        nenhuma_troca = False

```

⁶Em inglês, “Shell sort”, em honra ao seu criador Donald Shell [Shell, 1959].

```

while not nenhuma_troca:
    nenhuma_troca = True

    for i in range(len(lst)-intervalo):
        if lst[i] > lst[i+intervalo]:
            lst[i], lst[i+intervalo] = \
                lst[i+intervalo], lst[i]
            nenhuma_troca = False

    intervalo = intervalo // 2

```

5.5.3 Ordenação por selecção

Uma terceira alternativa de ordenação que iremos considerar, a ordenação *por selecção*⁷, consiste em percorrer os elementos a ordenar e, em cada passagem, colocar um elemento na sua posição correcta. Na primeira passagem, coloca-se o menor elemento na sua posição correcta, na segunda passagem, o segundo menor, e assim sucessivamente. A seguinte função efectua a ordenação por selecção:

```

def ordena(lst):
    for i in range(len(lst)):
        pos_menor = i
        for j in range(i + 1, len(lst)):
            if lst[j] < lst[pos_menor]:
                pos_menor = j
        lst[i], lst[pos_menor] = lst[pos_menor], lst[i]

```

5.6 Exemplo

Para ilustrar a utilização de algoritmos de procura e de ordenação, vamos desenvolver um programa que utiliza duas listas, `nomes` e `telefones`, contendo,

⁷Em inglês, “selection sort”.

respectivamente nomes de pessoas e números de telefones. Assumimos que o número de telefone de uma pessoa está armazenado na lista `telefones` na mesma posição que o nome dessa pessoa está armazenado na lista `nomes`. Por exemplo, o número de telefone da pessoa cujo nome é `nomes[3]`, está armazenado em `telefones[3]`. Listas com esta propriedade são chamadas *listas paralelas*. Assumimos também que cada pessoa tem, no máximo, um número de telefone.

O programa interacciona com um utilizador, ao qual fornece o número de telefone da pessoa cujo nome é fornecido ao programa. Esta interacção é repetida até que o utilizador forneça, como nome, a palavra `fim`.

O nosso programa utiliza uma abordagem muito simplista, na qual as listas são definidas dentro do próprio programa. Uma abordagem mais realista poderia ser obtida através da leitura desta informação do exterior, o que requer a utilização de ficheiros, os quais são apresentados no Capítulo 8.

O programa utiliza a função `procura` correspondente à procura binária, a qual foi apresentada na secção 5.4.2, e que não é repetida no nosso programa. O algoritmo para ordenar as listas de nomes e de telefones corresponde à ordenação por selecção apresentada na secção 5.5.3. Contudo, como estamos a lidar com listas paralelas, o algoritmo que usamos corresponde a uma variante da ordenação por selecção. Na realidade, a função `ordena_nomes` utiliza a ordenação por selecção para ordenar os nomes das pessoas, a lista que é usada para procurar os nomes, mas sempre que dois nomes são trocados, a mesma acção é aplicada aos respectivos números de telefone.

```
def lista_tel():

    def ordena_nomes(pessoas, telefs):
        for i in range(len(pessoas)):
            pos_menor = i
            for j in range(i + 1, len(pessoas)):
                if pessoas[j] < pessoas[pos_menor]:
                    pos_menor = j
            pessoas[i], pessoas[pos_menor] = \
                pessoas[pos_menor], pessoas[i]
            telefs[i], telefs[pos_menor] = \
                telefs[pos_menor], telefs[i]
```

```

pessoas = ['Ricardo Saldanha', 'Francisco Nobre', \
           'Leonor Martins', 'Hugo Dias', 'Luiz Leite', \
           'Ana Pacheco', 'Fausto Almeida']

telefones = [211234567, 919876543, 937659862, 964876347, \
             218769800, 914365986, 229866450]

ordena_nomes(pessoas, telefones)

quem = input('Qual o nome?\n(fim para terminar)\n? ')
while quem != 'fim':
    pos_tel = procura(pessoas, quem)
    if pos_tel == -1:
        print('Telefone desconhecido')
    else:
        print('O telefone é:', telefones[pos_tel])
    quem = input('Qual o nome?\n(fim para terminar)\n? ')

```

5.7 Considerações sobre eficiência

Dissemos que a procura binária é mais eficiente do que a procura sequencial, e que a ordenação Shell é mais eficiente do que a ordenação por borbulhamento. Nesta secção, discutimos métodos para comparar a eficiência de algoritmos. O estudo da eficiência de um algoritmo é um aspecto importante em informática, porque fornece uma medida grosseira do trabalho envolvido na execução de um algoritmo.

Um dos processos para determinar o trabalho envolvido na execução de um algoritmo consiste em considerar um conjunto de dados e seguir a execução do algoritmo para esses dados. Este aspecto está ilustrado nas figuras 5.9 a 5.11, em que mostramos a evolução da posição relativa dos elementos da lista [4, 8, 17, 3, 11, 2], utilizando os três métodos de ordenação que apresentámos. Nestas figuras, cada linha corresponde a uma passagem pela lista, um arco ligando duas posições da lista significa que os elementos da lista nessas posições foram comparados e uma seta por baixo de duas posições da lista significa que

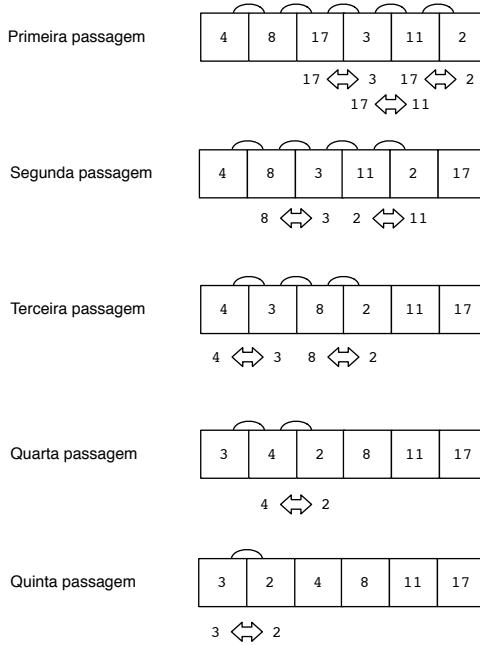


Figura 5.9: Padrão de ordenação utilizando a ordenação por borbulhamento.

os elementos nessas posições (os quais estão indicados nas extremidades da seta) foram trocados durante uma passagem.

Esta abordagem, contudo, tem a desvantagem do trabalho envolvido na execução de um algoritmo poder variar drasticamente com os dados que lhe são fornecidos (por exemplo, se os dados estiverem ordenados, a ordenação por borbulhamento só necessita de uma passagem). Por esta razão, é importante encontrar uma medida do trabalho envolvido na execução de um algoritmo que seja independente dos valores particulares dos dados que são utilizados na sua execução.

Este problema é resolvido com outro modo de descrever o trabalho envolvido na execução de um algoritmo, que recorre à noção de *ordem de crescimento*⁸ para obter uma avaliação grosseira dos recursos exigidos pelo algoritmo à medida que a quantidade de dados manipulados aumenta. A ideia subjacente a esta abordagem é isolar uma operação que seja fundamental para o algoritmo, e contar o número de vezes que esta operação é executada. Para o caso dos

⁸Do inglês, “order of growth”.

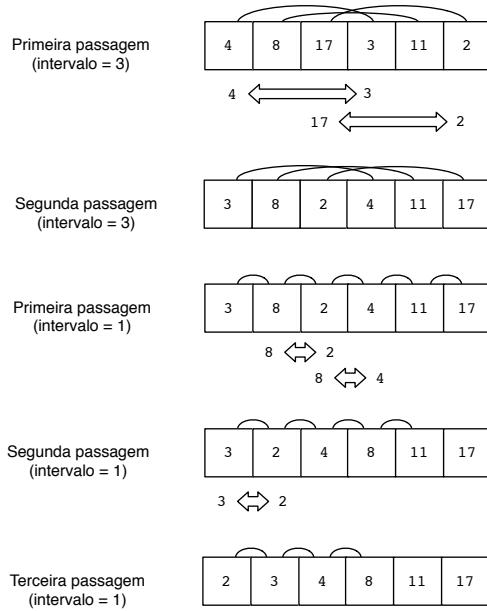
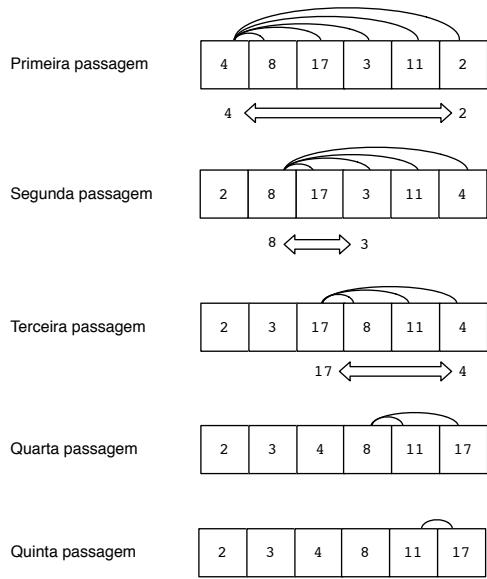
Figura 5.10: Padrão de ordenação utilizando a ordenação *Shell*.

Figura 5.11: Padrão de ordenação utilizando a ordenação por selecção.

algoritmos de procura e de ordenação, esta operação é a comparação (dados dois elementos, qual deles é o maior?). Segundo este processo, a medida de eficiência de um algoritmo de procura ou de ordenação é o número de comparações que estes efectuam. Dados dois algoritmos, diremos que o algoritmo que efectuar o menor número de comparações é o mais eficiente.

Na procura sequencial, percorremos a sequência de elementos até encontrar o elemento desejado. Se o elemento desejado se encontrar na primeira posição da lista, necessitamos apenas de uma comparação mas poderemos ter de percorrer toda a lista, se o elemento procurado não se encontrar na lista. O número médio de comparações cai entre estes dois extremos, pelo que o número médio de comparações na procura sequencial é:

$$\frac{n}{2}$$

em que n é o número de elementos na lista.

Na procura binária somos capazes de reduzir para metade o número de elementos a considerar sempre que efectuamos uma comparação (ver a função apresentada na página 154). Assim, se começarmos com n elementos, o número de elementos depois de uma passagem é $n/2$; o número de elementos depois de duas passagens é $n/4$ e assim sucessivamente. No caso geral, o número de elementos depois de i passagens é $n/2^i$. O algoritmo termina quando o número de elementos é menor do que 1, ou seja, terminamos depois de i passagens quando

$$\frac{n}{2^i} < 1$$

ou

$$n < 2^i$$

ou

$$\log_2(n) < i.$$

Deste modo, para uma lista com n elementos, a procura binária não exige mais do que $\log_2(n)$ passagens. Em cada passagem efectuamos três comparações (uma comparação na instrução `if` e duas comparações para calcular a expressão que controla o ciclo `while`). Consequentemente, o número de comparações exigida pela procura binária é inferior ou igual a $3 \cdot \log_2(n)$.

No algoritmo de ordenação por borbulhamento, trocamos pares adjacentes de

valores. O processo de ordenação consiste num certo número de passagens por todos os elementos da lista. O algoritmo da página 158 utiliza duas estruturas de repetição encadeadas. A estrutura exterior é realizada através de uma instrução `while`, e a interior com uma instrução `for`. Sendo n o número de elementos da lista, cada vez que o ciclo interior é executado, efectuam-se $n - 1$ comparações. O ciclo exterior é executado até que todos os elementos estejam ordenados. Na situação mais favorável, é executado apenas uma vez; na situação mais desfavorável é executado n vezes, neste caso, o número de comparações será $n \cdot (n - 1)$. O caso médio cai entre estes dois extremos, pelo que o número médio de comparações necessárias para a ordenação por borbulhamento será

$$\frac{n \cdot (n - 1)}{2} = \frac{1}{2} \cdot (n^2 - n).$$

No algoritmo de ordenação por selecção (apresentado na página 160), a operação básica é a selecção do menor elemento de uma lista. O algoritmo de ordenação por selecção é realizado através da utilização de dois ciclos `for` encadeados. Para uma lista com n elementos, o ciclo exterior é executado $n - 1$ vezes; o número de vezes que o ciclo interior é executado depende do valor da variável `i` do ciclo exterior. A primeira vez será executado $n - 1$ vezes, a segunda vez $n - 2$ vezes e a última vez será executado apenas uma vez. Assim, o número de comparações exigidas pela ordenação por selecção é

$$(n - 1) + (n - 2) + \dots + 2 + 1$$

Utilizando a fórmula para calcular a soma de uma progressão aritmética, obtemos o seguinte resultado para o número de comparações na ordenação por selecção:

$$(n - 1) \cdot \frac{(n - 1) + 1}{2} = \frac{1}{2} \cdot (n^2 - n).$$

5.7.1 A notação do Omaiúsculo

A nossa preocupação com a eficiência está relacionada com problemas que envolvem um número muito grande de elementos. Se estivermos a procurar numa lista com cinco elementos, mesmo o algoritmo menos eficiente nos resolve rapidamente o problema. Contudo, à medida que o número de elementos considerado cresce, o esforço necessário começa a diferir consideravelmente de algoritmo

para algoritmo. Por exemplo, se a lista em que efectuamos a comparação tiver 100 000 000 de elementos, a procura sequencial requer uma média de 50 000 000 de comparações, ao passo que a procura binária requer, no máximo, 61 comparações!

Podemos exprimir uma aproximação da relação entre a quantidade de trabalho necessário e o número de elementos considerados, utilizando uma notação matemática conhecida por ordem de magnitude ou *notação do Omaiúsculo* (lido, ó maiúsculo⁹).

A notação do Omaiúsculo é usada em matemática para descrever o comportamento de uma função, normalmente em termos de outra função mais simples, quando o seu argumento tende para o infinito. A notação do Omaiúsculo pertence a uma classe de notações conhecidas por notações de Bachmann-Landau¹⁰ ou notações assintóticas. A notação do Omaiúsculo caracteriza funções de acordo com a sua taxa de crescimento, funções diferentes com a mesma taxa de crescimento podem ser representadas pela mesma notação do Omaiúsculo. Existem outras notações associadas à notação do Omaiúsculo, usando os símbolos o , Ω , ω e Θ para descrever outros tipos de taxas de crescimento.

Sejam $f(x)$ e $g(x)$ duas funções com o mesmo domínio definido sobre o conjunto dos números reais. Escrevemos $f(x) = O(g(x))$ se e só se existir uma constante positiva k , tal que para valores suficientemente grandes de x , $|f(x)| \leq k \cdot |g(x)|$, ou seja, se e só se existe um número real positivo k e um real x_0 tal que $|f(x)| \leq k \cdot |g(x)|$ para todo o $x > x_0$.

A ordem de magnitude de uma função é igual à ordem do seu termo que cresce mais rapidamente em relação ao argumento da função. Por exemplo, a ordem de magnitude de $f(n) = n + n^2$ é n^2 uma vez que, para grandes valores de n , o valor de n^2 domina o valor de n (é tão importante face ao valor de n , que no cálculo do valor da função podemos praticamente desprezar este termo). Utilizando a notação do Omaiúsculo, a ordem de magnitude de $n^2 + n$ é $O(n^2)$.

Tendo em atenção esta discussão, podemos dizer que a procura binária é de ordem $O(\log_2(n))$, a procura sequencial é de ordem $O(n)$, e tanto a ordenação por borbulhamento como a ordenação por selecção são de ordem $O(n^2)$.

Os recursos consumidos por uma função não dependem apenas do algoritmo

⁹Do inglês, “Big-O”.

¹⁰Em honra aos matemáticos Edmund Landau (1877–1938) e Paul Bachmann (1837–1920).

utilizado mas também do grau de dificuldade ou dimensão do problema a ser resolvido. Por exemplo, vimos que a procura binária exige menos recursos do que a procura sequencial. No entanto, estes recursos dependem da dimensão do problema em causa (do número de elementos da lista): certamente que a procura de um elemento numa lista de 5 elementos recorrendo à procura sequencial consome menos recursos do que a procura numa lista de 100 000 000 elementos utilizando a procura binária. O *grau de dificuldade* de um problema é tipicamente dado por um número que pode estar relacionado com o valor de um dos argumentos do problema (nos casos da procura e da ordenação, o número de elementos da lista) ou o grau de precisão exigido (como no caso do erro admitido na função para o cálculo da raiz quadrada apresentada na Secção 3.4.5), etc.

De modo a obter uma noção das variações profundas relacionadas com as ordens de algumas funções, suponhamos que dispomos de um computador capaz de realizar 3 mil milhões de operações por segundo (um computador com um processador de 3GHz). Na Tabela 5.2 apresentamos a duração aproximada de alguns algoritmos com ordens de crescimento diferentes, para alguns valores do grau de dificuldade do problema. Notemos que para um problema com grau de dificuldade 19, se o seu crescimento for de ordem $O(6^n)$, para que a função termine nos nossos dias, esta teria que ter sido iniciada no período em que surgiu o *Homo sapiens* e para uma função com crescimento de ordem $O(n!)$ esta teria que ter sido iniciada no tempo dos dinossauros para estar concluída na actualidade. Por outro lado, ainda com base na Tabela 5.2, para um problema com grau de dificuldade 21, a resolução de um problema com ordem $O(\log_2(n))$ demoraria na ordem das 4.6 centésimas de segundo ao passo que a resolução de um problema com ordem $O(n!)$ requeria mais tempo do que a idade do universo.

5.8 Notas finais

Neste capítulo apresentámos o tipo lista, o qual é muito comum em programação, sendo tipicamente conhecido por tipo vector ou tipo tabela. A lista é um tipo mutável, o que significa que podemos alterar destrutivamente os seus elementos. Apresentámos alguns algoritmos de procura e de ordenação aplicados a listas. Outros métodos de procura podem ser consultados em [Knuth, 1973b] e [Cormen et al., 2009].

As listas existentes em Python diferem dos tipos vector e tabela existentes

n	$\log_2(n)$	n^3	6^n	$n!$
19	0.044 segundos	72.101 segundos	2.031×10^5 anos (Homo sapiens)	4.055×10^7 anos (tempo dos dinossauros)
20	0.045 segundos	84.096 segundos	1.219×10^6 anos	8.110×10^8 anos (início da vida na terra)
21	0.046 segundos	97.351 segundos	7.312×10^6 anos	1.703×10^{10} anos (superior à idade do universo)
22	0.047 segundos	1.866 minutos	4.387×10^7 anos (tempo dos dinossauros)	
23	0.048 segundos	2.132 minutos	2.632×10^8 anos	
24	0.048 segundos	2.422 minutos	1.579×10^9 anos	
25	0.049 segundos	2.738 minutos	9.477×10^9 anos (superior à idade da terra)	
26	0.049 segundos	3.079 minutos	5.686×10^{10} anos (superior à idade do universo)	
100	0.069 segundos	2.920 horas		
500	0.094 segundos	15.208 dias		
1000	0.105 segundos	121.667 dias		

Tabela 5.2: Duração comparativa de algoritmos.

em outras linguagens de programação, normalmente conhecidos pela palavra inglesa “array”, pelo facto de permitirem remover elementos existentes e aumentar o número de elementos existentes. De facto, em outras linguagens de programação, o tipo “array” apresenta uma característica estática: uma vez criado um elemento do tipo “array”, o número dos seus elementos é fixo. Os elementos podem ser mudados, mas não podem ser removidos e novos elementos não podem ser alterados. O tipo lista em Python mistura as características do tipo “array” de outras linguagens de programação com as características de outro tipo de informação correspondente às *listas ligadas*¹¹.

¹¹Do inglês “linked list”.

Apresentámos uma primeira abordagem à análise da eficiência de um algoritmo através do estudo da ordem de crescimento e introdução da notação do Omaiúsculo. Para um estudo mais aprofundado da análise da eficiência de algoritmos, recomendamos a leitura de [Cormen et al., 2009], [Edmonds, 2008] ou [McConnell, 2008].

5.9 Exercícios

1. Suponha que a operação `in` não existia em Python. Escreva uma função em Python, com o nome `pertence`, que recebe como argumentos uma lista e um inteiro e devolve `True`, se o inteiro está armazenado na lista, e `False`, em caso contrário. Não pode usar um ciclo `for` pois este recorre à operação `in`. Por exemplo,

```
>>> pertence(3, [2, 3, 4])
True
>>> pertence(1, [2, 3, 4])
False
```

2. Escreva uma função chamada `substitui` que recebe uma lista, `lst`, dois valores, `velho` e `novo`, e devolve a lista que resulta de substituir em `lst` todas as ocorrências de `velho` por `novo`. Por exemplo,

```
>>> substitui([1, 2, 3, 2, 4], 2, 'a')
[1, 'a', 3, 'a', 4]
```

3. Escreva uma função chamada `posicoes_lista` que recebe uma lista e um elemento, e devolve uma lista contendo todas as posições em que o elemento ocorre na lista. Por exemplo,

```
>>> posicoes_lista(['a', 2, 'b', 'a'], 'a')
[0, 3]
```

4. Escreva uma função chamada `parte` que recebe como argumentos uma lista, `lst`, e um elemento, `e`, e que devolve uma lista de dois elementos, contendo na primeira posição a lista com os elementos de `lst` menores que `e`, e na segunda posição a lista com os elementos de `lst` maiores ou iguais a `e`. Por exemplo,

```
>>> parte([2, 0, 12, 19, 5], 6)
[[2, 0, 5], [12, 19]]
>>> parte([7, 3, 4, 12], 3)
[[], [7, 3, 4, 12]]
```

5. Escreva um programa em Python que leia uma lista contendo inteiros e mude a ordem dos seus elementos de modo que estes apareçam por ordem inversa.
6. Uma *matriz* é uma tabela bidimensional em que os seus elementos são referenciados pela linha e pela coluna em que se encontram. Uma matriz pode ser representada como uma lista cujos elementos são listas. Com base nesta representação, escreva uma função, chamada `elemento_matriz` que recebe como argumentos uma matriz, uma linha e uma coluna e que devolve o elemento da matriz que se encontra na linha e coluna indicadas. A sua função deve permitir a seguinte interacção:

```
>>> m = [[1, 2, 3], [4, 5, 6]]
>>> elemento_matriz(m, 0, 0)
1
>>> elemento_matriz(m, 0, 3)
Índice inválido: coluna 3
>>> elemento_matriz(m, 4, 1)
Índice inválido: linha 4
```

7. Considere uma matriz como definida no exercício anterior. Escreva uma função em Python que recebe uma matriz e que a escreve sob a forma

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{matrix}$$

8. Considere, de novo, o conceito de matriz. Escreva uma função em Python que recebe como argumentos duas matrizes e devolve uma matriz correspondente ao produto das matrizes que são seus argumentos. Os elementos

da matriz produto são dados por

$$p_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

9. As funções de ordenação apresentadas neste capítulo correspondem a uma ordenação destrutiva, pois a lista original é destruída e substituída pela lista ordenada. Um processo alternativo consiste em criar uma lista com índices, que representam as posições ordenadas dos elementos da lista. Escreva uma função em Python para efectuar a ordenação por selecção, criando uma lista com índices. A sua função deve permitir a interacção:

```
>>> original = [5, 2, 9, 1, 4]
>>> ord = ordena(original)
>>> ord
[3, 1, 4, 0, 2]
>>> original
[5, 2, 9, 1, 4]
>>> for i in range(len(original)):
...     print(original[ord[i]])
...
1
2
4
5
9
```

Capítulo 6

Funções revisitadas

'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean neither more nor less.'

'The question is,' said Alice, 'whether you can make words mean so many different things.'

'The question is,' said Humpty Dumpty, 'which is to be the master — that's all.'

Lewis Carroll, *Through the Looking Glass*

6.1 Funções recursivas

Nesta secção abordamos um processo de definição de funções, denominado definição *recursiva* ou *por recorrência*. Diz-se que uma dada entidade é *recursiva* se ela for definida em termos de si própria. As definições recursivas são frequentemente utilizadas em matemática. Ao longo deste livro temos utilizado definições recursivas na apresentação das expressões em notação BNF. Por exemplo, a definição de $\langle \text{nomes} \rangle$ apresentada na página 76:

$$\begin{aligned}\langle \text{nomes} \rangle ::= & \langle \text{nome} \rangle | \\ & \langle \text{nome} \rangle , \langle \text{nomes} \rangle\end{aligned}$$

é recursiva pois $\langle \text{nomes} \rangle$ é definido em termos de si próprio. Esta definição afirma que $\langle \text{nomes} \rangle$ é ou um $\langle \text{nome} \rangle$ ou um $\langle \text{nome} \rangle$, seguido de uma vírgula, seguida de $\langle \text{nomes} \rangle$. A utilização de recursão é também frequente em imagens,

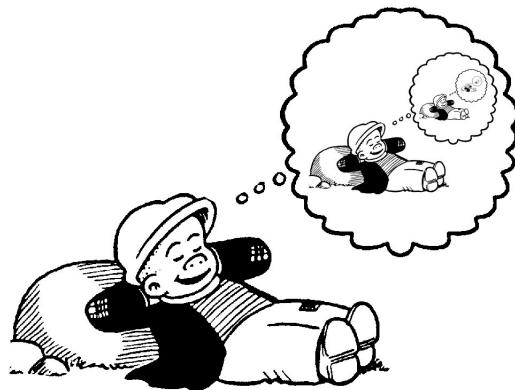


Figura 6.1: Exemplo de uma imagem recursiva.

um exemplo das quais apresentamos na Figura 6.1¹.

Como um exemplo utilizado em matemática de uma definição recursiva, consideremos a seguinte definição do conjunto dos números naturais: (1) 1 é um número natural; (2) o sucessor de um número natural é um número natural. A segunda parte desta definição é recursiva, porque utiliza o conceito de número natural para definir número natural: 2 é um número natural porque é o sucessor do número natural 1 (sabemos que 1 é um número natural pela primeira parte desta definição).

Outro exemplo típico de uma definição recursiva utilizada em matemática é a seguinte definição da função factorial:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n - 1)! & \text{se } n > 0 \end{cases}$$

Esta definição deve ser lida do seguinte modo: o factorial de 0 ($n = 0$) é 1; se n é maior do que 0, então o factorial de n é dado pelo produto entre n e o factorial de $n - 1$. A definição recursiva da função factorial é mais sugestiva, e rigorosa, do que a seguinte definição que frequentemente é apresentada:

$$n! = n \cdot (n - 1) \cdot (n - 2) \dots 2 \cdot 1.$$

¹Reproduzido com autorização de United Feature Syndicate. © United Feature Syndicate, Inc. Nancy and Sluggo are a registered trademark of United Feature Syndicate, Inc.

Note-se que na definição não recursiva de factorial, “...” significa que existe um padrão que se repete, padrão esse que deve ser descoberto por quem irá calcular o valor da função, ao passo que na definição recursiva, o processo de cálculo dos valores da função está completamente especificado.

O poder das definições recursivas baseia-se na possibilidade de definir um conjunto infinito utilizando uma frase finita. Do mesmo modo, um número arbitrariamente grande de cálculos pode ser especificado através de uma função recursiva, mesmo que esta função não contenha, explicitamente, estruturas de repetição.

Suponhamos que desejávamos definir a função factorial em Python. Podemos usar a definição não recursiva, descodificando o padrão correspondente às reticências, dando origem à seguinte função (a qual corresponde a uma variante, utilizando um ciclo `for`, da função apresentada na Secção 3.4.3):

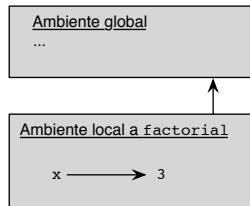
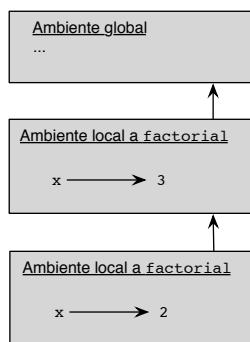
```
def factorial(n):
    fact = 1
    for i in range(n, 0, -1):
        fact = fact * i
    return fact
```

ou podemos utilizar directamente a definição recursiva, dando origem à seguinte função:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Esta segunda versão de `factorial` não contém explicitamente nenhum ciclo.

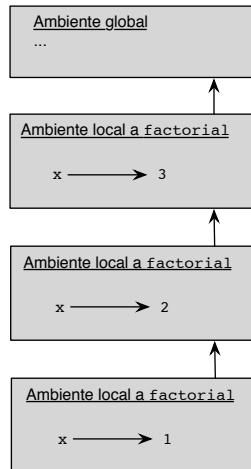
Como exercício, iremos seguir o funcionamento da versão recursiva da função `factorial` para calcular o factorial de 3. Ao avaliar `factorial(3)`, o valor 3 é associado ao parâmetro formal `n` da função `factorial`, criando-se o ambiente local apresentado na Figura 6.2 (nesta figura e nas seguintes, estamos propostadamente a ignorar as outras ligações que possam existir no ambiente global), no qual o corpo da função é executado. Como a expressão `3 == 0` tem o valor

Figura 6.2: Primeira chamada a `factorial`.Figura 6.3: Segunda chamada a `factorial`.

`False`, o valor devolvido será `3 * factorial(2)`.

Encontramos agora uma outra expressão que utiliza a função `factorial` e esta expressão inicia o cálculo de `factorial(2)`. É importante recordar que neste momento existe um cálculo suspenso, o cálculo de `factorial(3)`. Para calcular `factorial(2)`, o valor 2 é associado ao parâmetro formal `n` da função `factorial`, criando-se o ambiente local apresentado na Figura 6.3. Como a expressão `2 == 0` tem o valor `False`, o valor devolvido por `factorial(2)` é `2 * factorial(1)` e o valor de `factorial(1)` terá que ser calculado. Dois cálculos da função `factorial` estão agora suspensos: o cálculo de `factorial(3)` e o cálculo de `factorial(2)`.

Encontramos novamente outra expressão que utiliza a função `factorial` e esta expressão inicia o cálculo de `factorial(1)`. Para calcular `factorial(1)`, o valor 1 é associado ao parâmetro formal `n` da função `factorial`, criando-se o ambiente local apresentado na Figura 6.4. Como a expressão `1 == 0` tem o valor `False`, o valor devolvido por `factorial(1)` é `1 * factorial(0)` e o valor de

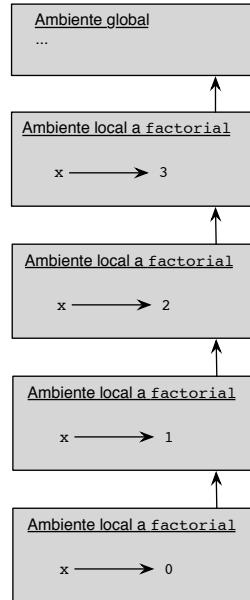
Figura 6.4: Terceira chamada a `factorial`.

`factorial(0)` terá que ser calculado. Três cálculos da função factorial estão agora suspensos: o cálculo de `factorial(3)`, `factorial(2)` e `factorial(1)`.

Para calcular `factorial(0)` associa-se o valor 0 ao parâmetro formal `n` da função `factorial`, criando-se o ambiente local apresentado na Figura 6.5 e executa-se o corpo da função. Neste caso a expressão `0 == 0` tem valor `True`, pelo que o valor de `factorial(0)` é 1.

Pode-se agora continuar o cálculo de `factorial(1)`, que é `1 * factorial(0)` = `1 * 1 = 1`, o que, por sua vez, permite calcular `factorial(2)`, cujo valor é `2 * factorial(1) = 2 * 1 = 2`, o que permite calcular `factorial(3)`, cujo valor é `3 * factorial(2) = 3 * 2 = 6`.

Em resumo, uma função diz-se *recursiva* se for definida em termos de si própria. A ideia fundamental numa função recursiva consiste em definir um problema em termos de uma versão semelhante, embora mais simples, de si próprio. Com efeito, quando definirmos $n!$ como $n \cdot (n - 1)!$ estamos a definir factorial em termos de uma versão mais simples de si próprio, pois o número de que queremos calcular o factorial é mais pequeno. Esta definição é repetida sucessivamente até se atingir uma versão do problema para a qual a solução seja conhecida. Utiliza-se então essa solução para sucessivamente calcular a solução de cada um dos subproblemas gerados e produzir a resposta desejada.

Figura 6.5: Quarta chamada a `factorial`.

Como vimos, durante a avaliação de uma expressão cujo operador é `factorial`, gera-se um encadeamento de operações suspensas à espera do valor de outras operações cujo operador é o próprio `factorial`. Na Figura 6.6 mostramos o encadeamento das operações geradas durante o cálculo de `factorial(3)`. Este encadeamento de operações pode ser representado do seguinte modo:

```

factorial(3)
3 * factorial(2)
3 * (2 * factorial(1))
3 * (2 * (1 * factorial(0)))
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
  
```

Podemos constatar que, usando a versão recursiva da função `factorial`, durante o processo para o cálculo do factorial de um número existem duas fases distintas: (1) numa primeira fase, a execução da operação de multiplicação vai sendo

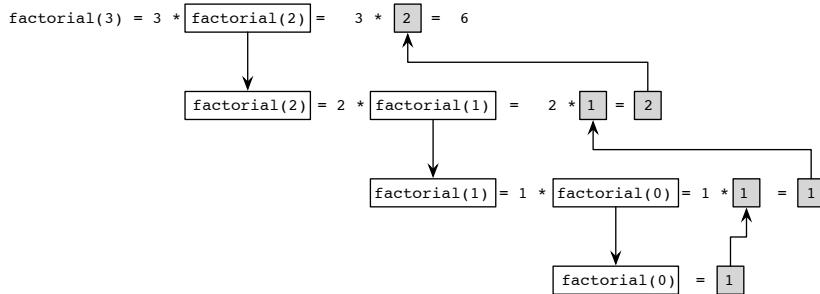


Figura 6.6: Encadeamento das operações no cálculo de `factorial(3)`.

sucessivamente adiada até se atingir o valor que corresponde à parte básica da função `factorial` (no nosso exemplo, `factorial(0)`); (2) numa segunda fase, as operações que foram adiadas são executadas.

Tanto as definições recursivas como as funções recursivas são constituídas por duas partes distintas:

1. Uma *parte básica*, também chamada *caso terminal*, a qual constitui a versão mais simples do problema para o qual a solução é conhecida. No caso da função factorial esta parte corresponde ao caso em que $n = 0$.
2. Uma *parte recursiva*, também chamada *caso geral*, na qual o problema é definido em termos de uma versão mais simples de si próprio. No caso da função factorial, isto corresponde ao caso em que $n > 0$.

Como segundo exemplo, consideremos novamente o algoritmo de Euclides (apresentado na Secção 3.4.4) para o cálculo do máximo divisor comum entre dois números. O algoritmo apresentado para o cálculo do máximo divisor comum afirma que:

1. O máximo divisor comum entre um número e zero é o próprio número.
2. Quando dividimos um número por um menor, o máximo divisor comum entre o resto da divisão e o divisor é o mesmo que o máximo divisor comum entre o dividendo e o divisor.

Este algoritmo corresponde claramente a uma definição recursiva. No entanto, na Secção 3.4.4 analisámos o seu comportamento e traduzimo-lo numa função

não recursiva. Podemos agora aplicar directamente o algoritmo, dado origem à seguinte função que calcula o máximo divisor comum:

```
def mdc(m, n):
    if n == 0:
        return m
    else:
        return mdc(n, m % n)
```

Esta função origina o seguinte processo de cálculo, por exemplo, para calcular `mdc(24, 16)`:

```
mdc(24, 16)
mdc(16, 8)
mdc(8, 0)
8
```

6.2 Funções de ordem superior

Grande parte das linguagens de programação apresentam regras rígidas quanto ao modo como objectos computacionais podem ser manipulados, quanto aos objectos computacionais que podem ser utilizados e quanto ao local onde estes podem ser utilizados. Em Python existe o mínimo possível destas regras. Em Python podemos utilizar um objecto computacional, sempre que a sua utilização fizer sentido. Não faz sentido somar dois valores lógicos ou calcular a conjunção de dois números, mas tanto valores lógicos como números podem ser nomeados, podem ser utilizados como argumentos de uma função ou podem ser o resultado da aplicação de uma função.

De acordo com o cientista inglês Christopher Stratchey (1916–1975), os objectos computacionais que podem ser nomeados, que podem ser utilizados como argumentos de funções, que podem ser devolvidos por funções e que podem ser utilizados como componentes de estruturas de informação, são designados por cidadãos de primeira classe.

A ideia subjacente à definição de um objecto computacional como um cidadão de primeira classe é a de que todos estes objectos computacionais têm os mesmos

direitos e responsabilidades. De um modo geral as linguagens de programação não são democráticas, dando os direitos de cidadão de primeira classe apenas a alguns dos seus objectos computacionais. O Python, e outras linguagens recentes, é mais democrático, dando este direito a todos, ou quase todos, os objectos computacionais.

Um dos objectos computacionais que tradicionalmente não é tratado como cidadão de primeira classe é a função. Em Python as funções são cidadãs de primeira classe e, consequentemente, podem ser utilizadas como argumentos de outras funções e podem corresponder ao valor devolvido por funções. Nesta secção vamos discutir como esta regalia das funções em Python permite construir abstracções mais poderosas do que as que temos utilizado até aqui.

6.2.1 Funções como parâmetros

Nos capítulos anteriores argumentámos que as funções correspondem a abstracções que definem operações compostas, independentemente dos valores por estas utilizados. Na realidade, ao definirmos a função `quadrado` como

```
def quadrado(x):
    return x * x
```

não estamos a falar do quadrado de um número em particular, mas sim de um padrão de cálculo que permite calcular o quadrado de qualquer número. A abstracção procedural permite introduzir o *conceito de quadrado*, evitando que tenhamos sempre de exprimi-lo em termos das operações elementares da linguagem. Sem a introdução do conceito de quadrado, poderíamos calcular o quadrado de qualquer número, mas faltava-nos a abstracção através da qual podemos referir-nos à operação capturada pelo conceito.

Nesta secção vamos apresentar conceitos mais abstractos do que os que temos utilizado até aqui com a noção de função. Recordemos mais uma vez que uma função captura um padrão de cálculo. Consideremos agora as seguintes operações que, embora distintas, englobam um padrão de cálculo comum:

1. Soma dos números naturais, inferiores ou iguais a 40:

$$1 + 2 + \dots + 40$$

2. Soma dos quadrados dos números naturais, inferiores ou iguais a 40:

$$1^2 + 2^2 + \cdots + 40^2$$

3. Soma dos inversos dos quadrados dos números naturais ímpares, inferiores ou iguais a 40:

$$\frac{1}{1^2} + \frac{1}{3^2} + \cdots + \frac{1}{39^2}$$

Estas operações podem ser realizadas, respectivamente, através das seguintes funções:

```
def soma_inteiros(linf, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + linf
        linf = linf + 1
    return soma

def soma_quadrados(linf, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + quadrado(linf)
        linf = linf + 1
    return soma

def soma_inv_quadrados_impares(linf, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + 1 / quadrado(linf)
        linf = linf + 2
    return soma
```

As quais permitem gerar a interacção:

```
>>> soma_inteiros(1, 40)
```

820

```
>>> soma_quadrados(1, 40)
22140
>>> soma_inv_quadrados_impares(1, 40)
1.2212031520286797
```

Estas três funções, embora semelhantes, apresentam as seguintes diferenças: (1) o seu nome; (2) o processo de cálculo do termo a ser adicionado; e (3) o processo do cálculo do próximo termo a adicionar.

Independentemente destas diferenças, as três funções anteriores partilham o seguinte padrão de cálculo:

```
def <nome> (linf, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + <calc_termo>(linf)
        linf = <prox>(linf)
    return soma
```

em que `<nome>`, `<calc_termo>` e `<prox>` são símbolos não terminais que assumem valores particulares para cada uma das funções. A existência deste padrão mostra que existe uma abstracção escondida subjacente a este cálculo. Os matemáticos identificaram esta abstracção através do conceito de somatório:

$$\sum_{n=linf}^{l_{sup}} f(n) = f(linf) + f(linf + 1) + \dots + f(lsup)$$

O poder da abstracção correspondente ao somatório permite lidar com o *conceito de soma* em vez de tratar apenas com somas particulares. A existência da abstracção correspondente ao somatório leva-nos a pensar em definir uma função correspondente a esta abstracção em vez de apenas utilizar funções que calculam somas particulares.

Um dos processos para tornar as funções mais gerais corresponde a utilizar parâmetros adicionais que indicam quais as operações a efectuar sobre os objectos computacionais manipulados pelas funções, e assim idealizar uma função correspondente a um somatório que, para além de receber a indicação sobre os elementos extremos a somar (`linf` e `lsup`), recebe também como parâmetros as funções para calcular um termo do somatório (`calc_termo`) e para calcular

o próximo termo a adicionar (`prox`):

```
def somatorio(calc_termo, linf, prox, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + calc_termo(linf)
        linf = prox(linf)
    return soma
```

Definindo agora funções para adicionar uma unidade ao seu argumento (`inc1`), para devolver o seu próprio argumento (`identidade`), para adicionar duas unidades ao seu argumento (`inc2`), para calcular o quadrado do seu argumento (`quadrado`) e para calcular o inverso do quadrado do seu argumento (`inv-quadrado`):

```
def inc1(x):
    return x + 1

def identidade(x):
    return x

def inc2(x):
    return x + 2

def quadrado (x):
    return x * x

def inv_quadrado(x):
    return 1 / quadrado(x)
```

Obtemos a seguinte interacção:

```
>>> somatorio(identidade, 1, inc1, 40)
820
```

```
>>> somatorio(quadrado, 1, inc1, 40)
22140
>>> somatorio(inv_quadrado, 1, inc2, 40)
1.2212031520286797
```

Ou seja, utilizando funções como parâmetros, somos capazes de definir a função `somatorio` que captura o conceito de somatório utilizado em Matemática. Com esta função, cada vez que precisamos de utilizar um somatório, em lugar de escrever um programa, utilizamos um programa existente.

Existem dois aspectos que é importante observar em relação à função anterior. Em primeiro lugar, a função `somatorio` não foi exactamente utilizada com funções como parâmetros, mas sim com parâmetros que correspondem a nomes de funções. Em segundo lugar, esta filosofia levou-nos a criar funções que podem ter pouco interesse, fora do âmbito das somas que estamos interessados em calcular, por exemplo, a função `identidade`.

Recorde-se a discussão sobre funções apresentada nas páginas 74–75. Podemos reparar que o nome da função é, de certo modo, supérfluo, pois o que na realidade nos interessa é saber *como* calcular o valor da função para um dado argumento – a função é o conjunto dos pares ordenados. A verdadeira importância do nome da função é a de fornecer um modo de podermos *falar sobre* ou *designar* a função. Em 1941, o matemático Alonzo Church inventou uma notação para modelar funções a que se dá o nome de *cálculo lambda*². No cálculo lambda, a função que soma 3 ao seu argumento é representada por $\lambda(x)(x + 3)$. Nesta notação, imediatamente a seguir ao símbolo λ aparece a lista dos argumentos da função, a qual é seguida pela expressão designatória que permite calcular o valor da função. Uma das vantagens do cálculo lambda é permitir a utilização de funções sem ter que lhes dar um nome. Para representar a aplicação de uma função a um elemento do seu domínio, escreve-se a função seguida do elemento para o qual se deseja calcular o valor. Assim, $(\lambda(x)(x + 3))(3)$ tem o valor 6; da mesma forma, $(\lambda(x, y)(x \cdot y))(5, 6)$ tem o valor 30.

Em Python existe a possibilidade de definir funções sem nome, as *funções anónimas*, recorrendo à notação lambda. Uma função anónima é definida através da seguinte expressão em notação BNF:

$\langle \text{função anónima} \rangle ::= \text{lambda } \langle \text{parâmetros formais} \rangle : \langle \text{expressão} \rangle$

²Ver [Church, 1941].

Nesta definição, *⟨expressão⟩* corresponde a uma expressão em Python, a qual não pode conter ciclos. Este último aspecto não é capturado na nossa definição de *⟨função anónima⟩*.

Ao encontrar uma *⟨função anónima⟩*, o Python cria uma função cujos parâmetros formais correspondem a *⟨parâmetros formais⟩* e cujo corpo corresponde a *⟨expressão⟩*. Quando esta função anónima é executada, os parâmetros concretos são associados aos parâmetros formais, e o valor da função é o valor da *⟨expressão⟩*. Note-se que numa função anónima não existe uma instrução `return` (estando esta implicitamente associada a *⟨expressão⟩*).

Por exemplo, `lambda x : x + 1` é uma função anónima que devolve o valor do seu argumento mais um. Com esta função anónima podemos gerar a interacção:

```
>>> (lambda x : x + 1)(3)
4
```

O que nos interessa fornecer à função `somatorio` não são os nomes das funções que calculam um termo do somatório e que calculam o próximo termo a adicionar, mas sim as próprias funções. Tendo em atenção, por exemplo, que a função que adiciona 1 ao seu argumento é dado por `lambda x : x + 1`, independentemente do nome com que é baptizada, podemos utilizar a função `somatorio`, recorrendo a funções anónimas como mostra a seguinte interacção (a qual pressupõe a definição da função `quadrado`):

```
>>> somatorio(lambda x : x, 1, lambda x : x + 1, 40)
820
>>> somatorio(quadrado, 1, lambda x : x + 1, 40)
22140
>>> somatorio(lambda x : 1/quadrado(x), 1, lambda x : x + 2, 40)
1.2212031520286797
```

Note-se que, a partir da definição da função `somatorio`, estamos em condições de poder calcular qualquer somatório. Por exemplo, usando a definição da função `factorial` apresentada na página 175:

```
def factorial(n):
    if n == 0:
```

```

        return 1
else:
    return n * factorial(n-1)

```

Podemos calcular a soma dos factoriais dos 20 primeiros números naturais através de:

```
>>> somatorio(factorial, 1, lambda x : x + 1, 20)
2561327494111820313
```

Funcionais sobre listas

Com a utilização de listas é vulgar recorrer a um certo número de funções de ordem superior (ou funcionais). Nesta secção apresentamos algumas das funções de ordem superior aplicáveis a listas.

- Um *transformador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e devolve uma lista em que cada elemento resulta da aplicação da operação ao elemento correspondente da lista original. Podemos realizar um transformador através da seguinte função:

```

def transforma(tr, lista):
    res = list()
    for e in lista:
        res = res + [tr(e)]
    return res

```

A seguinte interacção corresponde a uma utilização de um transformador, utilizando a função `quadrado` da página 76:

```
>>> transforma(quadrado, [1, 2, 3, 4, 5, 6])
[1, 4, 9, 16, 25, 36]
```

- Um *filtro* é um funcional que recebe como argumentos uma lista e um predicado aplicável aos elementos da lista, e devolve a lista constituída apenas pelos elementos da lista original que satisfazem o predicado. Podemos realizar um filtro através da seguinte função:

```
def filtra(teste, lista):
    res = list()
    for e in lista:
        if teste(e):
            res = res + [e]
    return res
```

A seguinte interacção corresponde a uma utilização de um filtro que testa se um número é par:

```
>>> filtra(lambda x : x % 2 == 0, [1, 2, 3, 4, 5, 6])
[2, 4, 6]
```

- Um *acumulador* é um funcional que recebe como argumentos uma lista e uma operação aplicável aos elementos da lista, e aplica sucessivamente essa operação aos elementos da lista original, devolvendo o resultado da aplicação da operação a todos os elementos da lista. Podemos realizar um acumulador através da seguinte função³:

```
def acumula(fn, lst):
    res = lst[0]
    for i in range(1, len(lst)):
        res = fn(res, lst[i])
    return res
```

A seguinte interacção corresponde a uma utilização de um acumulador, calculando o produto de todos os elementos da lista:

```
>>> acumula(lambda x, y : x * y, [1, 2, 3, 4, 5])
120
```

Funções como métodos gerais

Nesta secção apresentamos um exemplo que mostra como a utilização de funções como argumentos de funções pode originar métodos gerais de computação, independentemente das funções envolvidas.

³Como exercício, o leitor deverá explicar a razão da variável `res` ser inicializada para `lst[0]` e não para 0.

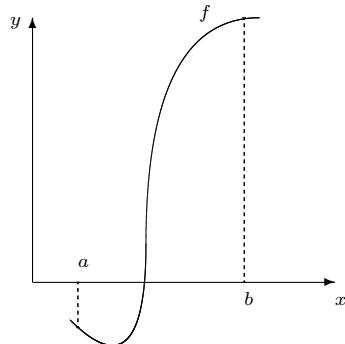


Figura 6.7: Situação à qual o método do intervalo é aplicável.

Para isso, apresentamos um processo de cálculo de raízes de equações pelo *método do intervalo*. Este método é aplicável ao cálculo de raízes de funções contínuas entre dois pontos e baseia-se no corolário do teorema de Bolzano, o qual afirma que, se $f(x)$ é uma função contínua entre os pontos a e b , tais que $f(a) < 0 < f(b)$, então podemos garantir que $f(x)$ tem um zero entre os pontos a e b (Figura 6.7).

Nesta situação, para calcular a raiz da função f no intervalo $[a, b]$, calcula-se o valor de $f(x)$ no ponto médio do intervalo. Se este valor for positivo, podemos garantir que $f(x)$ tem um zero entre a e o valor médio, $(a+b)/2$; se este valor for negativo, podemos garantir que $f(x)$ tem um zero entre o valor médio, $(a+b)/2$, e b . Podemos então repetir o processo com o novo intervalo. Este processo será repetido até que o intervalo em consideração seja suficientemente pequeno.

Consideremos a seguinte função em Python que recebe como argumentos uma função contínua (f) e os extremos de um intervalo no qual a função assume um valor negativo e um valor positivo, respectivamente, p_{neg} e p_{pos} . Esta função calcula o zero da função no intervalo especificado, utilizando o método do intervalo.

```
def raiz(f, pneg, ppos):
    while not suf_perto(pneg, ppos):
        pmedio = (pneg + ppos) / 2
        if f(pmedio) > 0:
            ppos = pmedio
        else:
```

```

    elif f(pmedio) < 0:
        pneg = pmedio
    else:
        return pmedio
    return pmedio

```

Esta função parte da existência de uma função para decidir se dois valores estão suficientemente próximos (`suf_perto`). Note-se ainda que o `else` da instrução de selecção serve para lidar com os casos em que `f(pmedio)` não é nem positivo, nem negativo, ou seja, `f(pmedio)` corresponde a um valor nulo e consequentemente é um zero da função.

Para definir quando dois valores estão suficientemente próximos utilizamos a mesma abordagem que usámos no cálculo da raiz quadrada apresentado na Secção 3.4.5: quando a sua diferença em valor absoluto for menor do que um certo limiar (suponhamos que este limiar é 0.001). Podemos assim escrever a função `suf_perto`:

```

def suf_perto(a, b):
    return abs(a - b) < 0.001

```

Notemos agora que a função `raiz` parte do pressuposto que os valores extremos do intervalo correspondem a valores da função, `f`, com sinais opostos e que `f(pneg) < 0 < f(ppos)`. Se algum destes pressupostos não se verificar, a função não calcula correctamente a raiz. Para evitar este problema potencial, definimos a seguinte função:

```

def met_intervalo(f, a, b):
    fa = f(a)
    fb = f(b)
    if fa < 0 < fb:
        return calcula_raiz(f, a, b)
    elif fb < 0 < fa:
        return calcula_raiz(f, b, a)
    else:
        print('Metodo do intervalo: valores não opostos')

```

Com base nesta função, podemos obter a seguinte interacção:

```
>>> met_intervalo(lambda x : x * x * x - 2 * x - 3, 1, 2)
1.8935546875
>>> met_intervalo(lambda x : x * x * x - 2 * x - 3, 1, 1.2)
Metodo do intervalo: valores não opostos
```

Sabemos também que devemos garantir que a função `raiz` apenas é utilizada pela função `met_intervalo`, o que nos leva a definir a seguinte estrutura de blocos:

```
def met_intervalo(f, a, b):

    def calcula_raiz(f, pneg, ppos):
        while not suf_perto(pneg, ppos):
            pmedio = (pneg + ppos) / 2
            if f(pmedio) > 0:
                ppos = pmedio
            elif f(pmedio) < 0:
                pneg = pmedio
            else:
                return pmedio
        return pmedio

    def suf_perto(a, b):
        return abs(a - b) < 0.001

    fa = f(a)
    fb = f(b)
    if fa < 0 < fb:
        return calcula_raiz(f, a, b)
    elif fb < 0 < fa:
        return calcula_raiz(f, b, a)
    else:
        print('Metodo do intervalo: valores não opostos')
```

6.2.2 Funções como valor de funções

Na secção anterior vimos que em Python as funções podem ser utilizadas como argumentos de outras funções. Esta utilização permite a criação de abstracções mais gerais do que as obtidas até agora. Nesta secção, vamos apresentar outro aspecto que torna as funções cidadãos de primeira classe, discutindo funções como valores produzidos por funções.

Cálculo de derivadas

Suponhamos que queremos desenvolver uma função que calcula o valor da derivada, num dado ponto, de uma função real de variável real, f . Esta função real de variável real pode ser, por exemplo, o quadrado de um número, e deverá ser um dos argumentos da função.

Por definição, sendo a um ponto do domínio da função f , a derivada de f no ponto a , representada por $f'(a)$, é dada por:

$$f'(a) = \lim_{x \rightarrow a} \frac{f(x) - f(a)}{x - a}$$

ou, fazendo $h = x - a$,

$$f'(a) = \lim_{h \rightarrow 0} \frac{f(a + h) - f(a)}{h}$$

Sendo dx um número suficientemente pequeno, podemos considerar que a seguinte fórmula fornece uma boa aproximação para a derivada da função f no ponto a :

$$f'(a) \cong \frac{f(a + dx) - f(a)}{dx}$$

Podemos agora definir a função `derivada_ponto`, a qual recebe como argumento uma função correspondente a `f`, e um ponto do seu domínio, `a`, e produz o valor da derivada nesse ponto:

```
def derivada_ponto(f, a):
    return (f(a + dx) - f(a)) / dx
```

Esta função, juntamente com a definição do que se entende por algo suficientemente pequeno, `dx` (por exemplo, 0.00001),

```
dx = 0.00001
```

permite calcular a derivada de funções arbitrárias em pontos particulares do seu domínio. A seguinte interacção calcula a derivada da função `quadrado` (apresentada na página 76) para os pontos 3 e 10^4 :

```
>>> derivada_ponto(quadrado, 3)
6.000009999951316
>>> derivada_ponto(quadrado, 10)
20.00000999942131
```

Em Matemática, após a definição de derivada num ponto, é habitual definir a *função derivada*, a qual a cada ponto do domínio da função original associa o valor da derivada nesse ponto. O conceito de derivada de uma função é suficientemente importante para ser capturado como uma abstracção. Repare-se que a derivada num ponto arbitrário x é calculada substituindo x por a na função que apresentámos. O que nos falta na função anterior é capturar o “conceito de função”. Assim, podemos definir a seguinte função:

```
def derivada(f):
    def fn_derivada(x):
        return (f(x + dx) - f(x)) / dx
    return fn_derivada
```

A função anterior recebe como parâmetro uma função, `f`, e devolve como valor a função (ou seja, `derivada(f)` é uma função) que, quando aplicada a um valor, `a`, produz a derivada da função `f` para o ponto `a`. Ou seja, `derivada(f)(a)` corresponde à derivada de `f` no ponto `a`.

Podemos agora gerar a interacção:

```
>>> derivada(quadrado)
<function fn_derivada at 0x10f45d0>
```

⁴Note-se que a definição da função `quadrado` não é necessária, pois podemos usar `lambda x : x * x`. No entanto, a definição da função `quadrado` torna as nossas expressões mais legíveis.

```
>>> derivada(quadrado)(3)
6.00000999951316
>>> derivada(quadrado)(10)
20.0000099942131
```

Nada nos impede de dar um nome a uma função derivada, obtendo a interacção:

```
>>> der_quadrado = derivada(quadrado)
>>> der_quadrado(3)
6.00000999951316
```

Raízes pelo método de Newton

Na Secção 6.2.1 apresentámos uma solução para o cálculo de raízes de equações pelo método do intervalo. Outro dos métodos muito utilizados para determinar raízes de equações é o método de Newton, o qual é aplicável a funções diferenciáveis, e consiste em partir de uma aproximação, x_n , para a raiz de uma função diferenciável, f , e calcular, como nova aproximação, o ponto onde a tangente ao gráfico da função no ponto $(x_n, f(x_n))$ intersecta o eixo dos xx (Figura 6.8). É fácil de concluir que a nova aproximação será dada por:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

A função matemática que para um dado valor de uma aproximação calcula uma nova aproximação é chamada *transformada de Newton*. A transformada de Newton é definida por:

$$t_N(x) = x - \frac{f(x)}{f'(x)}$$

A determinação da raiz de uma função f , utilizando o método de Newton, corresponde a começar com uma primeira aproximação para a raiz (um palpite), x_0 , e gerar os valores:

$$\begin{aligned} &x_0 \\ &x_1 = t_N(x_0) \\ &x_2 = t_N(x_1) \end{aligned}$$

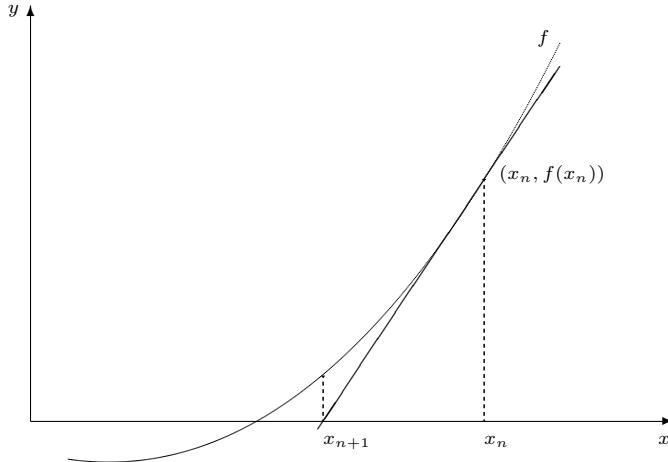


Figura 6.8: Representação gráfica subjacente ao método de Newton.

$$\begin{gathered} \vdots \\ x_n = t_N(x_{n-1}) \end{gathered}$$

Podemos agora definir as seguintes funções para calcular a raiz de uma função (representada por f), com base num palpite inicial (**palpite**):

```
def met_newton(f, palpite):
    while not boa_aprox(f(palpito)):
        palpito = transf_newton(f)(palpito)
    return palpito

def transf_newton(f):
    def t_n(x):
        return x - f(x) / derivada(f)(x)
    return t_n

def boa_aprox(x):
    return abs(x) < dx
```

Com base no método de Newton, e definindo dx como 0.0001, podemos agora calcular os zeros das seguintes funções:

```
>>> met_newton(lambda x : x * x * x - 2 * x - 3, 1.0)
1.8932892212475259
>>> from math import *
>>> met_newton(sin, 2.0)
3.1415926536589787
```

Suponhamos agora que queríamos definir a função arco de tangente (*arctg*) a partir da função tangente (*tg*). Sabemos que $\text{arctg}(x)$ é o número y tal que $x = \text{tg}(y)$, ou seja, para um dado x , o valor de $\text{arctg}(x)$ corresponde ao zero da equação $\text{tg}(y) - x = 0$. Recorrendo ao método de Newton, podemos então definir:

```
def arctg(x):
    # tg no módulo math tem o nome tan
    return met_newton(lambda y : tan(y) - x, 1.0)
```

obtendo a interacção:

```
>>> from math import *
>>> arctg(0.5)
0.4636478065118169
```

6.3 Programação funcional

Existe um paradigma de programação, chamado *programação funcional* que é baseado exclusivamente na utilização de funções. Um *paradigma de programação* é um modelo para abordar o modo de raciocinar durante a fase de programação e, consequentemente, o modo como os programas são escritos.

O tipo de programação que temos utilizado até agora tem o nome de *programação imperativa*. Em programação imperativa, um programa é considerado como um conjunto de ordens dadas ao computador, e daí a designação de “imperativa”, por exemplo, actualiza o valor desta variável, chama esta função, repete a execução destas instruções até que certa condição se verifique.

Uma das operações centrais em programação imperativa corresponde à instrução de atribuição através da qual é criada uma associação entre um nome (considerado como uma variável) e um valor ou é alterado o valor que está associado

com um nome (o que corresponde à alteração do valor da variável). A instrução de atribuição leva a considerar variáveis como referências para o local onde o seu valor é guardado e o valor guardado nesse local pode variar. Um outro aspecto essencial em programação imperativa é o conceito de ciclo, uma sequência de instruções associada a uma estrutura que controla o número de vezes que essas instruções são executadas.

Em programação funcional, por outro lado, um programa é considerado como uma função matemática que cumpre os seus objectivos através do cálculo dos valores (ou da avaliação) de outras funções. Em programação funcional não existe o conceito de instrução de atribuição e podem nem existir ciclos. O conceito de repetição é realizado exclusivamente através da recursão.

Para ilustrar a utilização da programação funcional, consideremos a função potência apresentada na Secção 3.4.2. Uma alternativa para definir x^n é através da seguinte análise de casos:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x.(x^{n-1}) & \text{se } n > 1 \end{cases}$$

ou seja, x^0 é 1, e para $n > 0$, x^n é o produto de x pela potência de x com expoente imediatamente inferior $x.(x^{n-1})$.

Podemos traduzir directamente esta definição para a seguinte função em Python⁵:

```
def potencia(x, n):
    if n == 0:
        return 1
    else:
        return x * potencia(x, n-1)
```

Repare-se que estamos perante uma função que corresponde a uma definição recursiva. A função `potencia` é definida através de uma parte básica, para a qual a solução é conhecida (se $n = 0$, o valor da potência é 1), e de uma parte recursiva, para qual a potência é definida através de uma versão mais simples de si própria (o expoente é menor).

Nesta função não existe uma definição explícita de repetição, sendo essa re-

⁵Propositalmente, não estamos a fazer nenhuma verificação em relação aos possíveis valores fornecidos a esta função.

petição tratada implicitamente pela parte recursiva da função. Note-se que também não existe nenhuma instrução de atribuição, os parâmetros formais recebem os seus valores quando a função é chamada (em programação funcional, diz-se que a *função é avaliada*), sendo os valores devolvidos pela função utilizados nos cálculos subsequentes.

Como segundo exemplo, consideremos a função `raiz` apresentada na Secção 3.4.5. A seguinte função corresponde a uma versão funcional de `raiz`:

```
def raiz(x):

    def calcula_raiz(x, palpito):

        def bom_palpite(x, palpito):
            return abs(x - palpito * palpito) < 0.001

        def novo_palpite(x, palpito):
            return (palpito + x / palpito) / 2

        if bom_palpite(x, palpito):
            return palpito
        else:
            return calcula_raiz(x, novo_palpite(x, palpito))

    if x >= 0:
        return calcula_raiz (x, 1)
    else:
        raise ValueError ('raiz: argumento negativo')
```

A função `calcula_raiz` apresentada na página 89 foi substituída por uma versão recursiva que evita a utilização do ciclo `while`.

Como exemplo final, o seguinte programa devolve a soma dos dígitos do número fornecido pelo utilizador, recorrendo à função `soma_digitos`. Repare-se na não existência de instruções de atribuição, sendo o valor fornecido pelo utilizador o parâmetro concreto da função `soma_digitos`.

```
def prog_soma():
```

```
return soma_digitos(eval(input('Escreva um inteiro\n? ')))  
  
def soma_digitos(n):  
    if n == 0:  
        return n  
    else:  
        return n % 10 + soma_digitos(n // 10)
```

6.4 Notas finais

Neste capítulo, introduzimos o conceito de função recursiva, uma função que se utiliza a si própria. Generalizámos o conceito de função através da introdução de dois aspectos, a utilização de funções como argumentos para funções e a utilização de funções que produzem objectos computacionais que correspondem a funções. Esta generalização permite-nos definir abstracções de ordem superior através da construção de funções que correspondem a métodos gerais de cálculo.

Introduzimos também um paradigma de programação conhecido por programação funcional.

O livro [Hofstader, 1979] (também disponível em português [Hofstader, 2011]), galardoado com o Prémio Pulitzer em 1980, ilustra o tema da recursão (auto-referência) discutindo como a utilização de regras formais permite que sistemas adquiram significado apesar de serem constituídos por símbolos sem significado.

6.5 Exercícios

1. Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve a soma dos seus dígitos pares. Por exemplo,

```
>>> soma_digitos_pares(234567)  
12
```

2. Escreva uma função recursiva em Python que recebe um número inteiro positivo e devolve o inteiro correspondente a inverter a ordem dos seus dígitos. Por exemplo,

```
>>> inverte_digitos(7633256)
6523367
```

3. Utilizando os funcionais sobre listas escreva uma função que recebe uma lista de inteiros e que devolve a soma dos quadrados dos elementos da lista.
4. Defina uma função de ordem superior que recebe funções para calcular as funções reais de variável real f e g e que se comporta como a seguinte função matemática:

$$h(x) = f(x)^2 + 4g(x)^3$$

5. A função `piatorio` devolve o produto dos valores de uma função, `fn`, para pontos do seu domínio no intervalo $[a, b]$ espaçados pela função `prox`:

```
def piatorio(fn, a, prox, b):
    res = 1
    valor = a
    while valor <= b:
        res = res * fn(valor)
        valor = prox(valor)
    return res
```

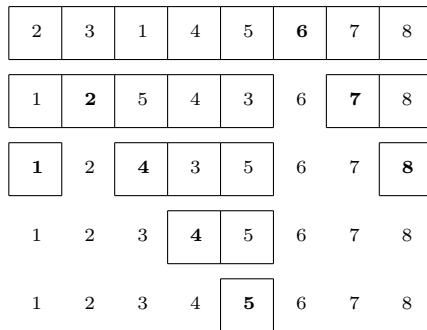
Use a função `piatorio` para definir a função `sin` que calcula o seno de um número, usando a seguinte aproximação:

$$\sin(x) = x \left(1 - \left(\frac{x}{\pi}\right)^2\right) \left(1 - \left(\frac{x}{2\pi}\right)^2\right) \left(1 - \left(\frac{x}{3\pi}\right)^2\right) \dots$$

A sua função deverá receber o número para o qual se pretende calcular o seno e o número de factores a considerar.

6. Escreva a função `faz_potencia` que recebe, como argumento, um inteiro `n` não negativo e devolve uma função que calcula a potência `n` de um número. Por exemplo, `faz_potencia(3)` devolve a função que calcula o cubo do seu argumento. Esta função é ilustrada na seguinte interacção:

```
>>> faz_potencia(3)
<function f_p at 0x10f4540>
>>> faz_potencia(3)(2)
8
```

Figura 6.9: Passos seguidos no *quick sort*.

Este exercício é um exemplo da definição de uma função de dois argumentos como uma função de um argumento, cujo valor é uma função de um argumento. Esta técnica, que é útil quando desejamos manter um dos argumentos fixo, enquanto o outro pode variar, é conhecida como método de *Curry*, e foi concebida por Moses Schönfinkel [Schönfinkel, 1977] e baptizada em honra do matemático americano Haskell B. Curry (1900–1982).

7. Um método de ordenação muito eficiente, chamado *quick sort*, consiste em considerar um dos elementos a ordenar (em geral, o primeiro elemento da lista), e dividir os restantes em dois grupos, um deles com os elementos menores e o outro com os elementos maiores que o elemento considerado. Este é colocado entre os dois grupos, que são por sua vez ordenados utilizando *quick sort*. Por exemplo, os passos apresentados na Figura 6.9 correspondem à ordenação da lista

6	2	3	1	8	4	7	5
---	---	---	---	---	---	---	---

utilizando *quick sort* (o elemento escolhido em cada lista representa-se a carregado).

Escreva uma função em Python para efectuar a ordenação de uma lista utilizando *quick sort*.

8. Considere a função `derivada` apresentada na página 193. Com base na sua definição escreva uma função que recebe uma função correspondente

a uma função e um inteiro n ($n \geq 1$) e devolve a derivada de ordem n da função. A derivada de ordem n de uma função é a derivada da derivada de ordem $n - 1$.

9. Escreva uma função chamada `rasto` que recebe como argumentos uma cadeia de caracteres correspondendo ao nome de uma função, e uma função de um argumento.

A função `rasto` devolve uma função de um argumento que escreve no ecrã a indicação de que a função foi avaliada e o valor do seu argumento, escreve também o resultado da função e tem como valor o resultado da função. Por exemplo, partindo do princípio que a função `quadrado` foi definida, podemos gerar a seguinte interacção:

```
>>> rasto_quadrado = rasto('quadrado', quadrado)
>>> rasto_quadrado(3)
Avaliação de quadrado com argumento 3
Resultado 9
9
```

10. Tendo em atenção que \sqrt{x} é o número y tal que $y^2 = x$, ou seja, para um dado x , o valor de \sqrt{x} corresponde ao zero da equação $y^2 - x = 0$, utilize o método de Newton para escrever uma função que calcula a raiz quadrada de um número.
11. Dada uma função f e dois pontos a e b do seu domínio, um dos métodos para calcular a área entre o gráfico da função e o eixo dos xx no intervalo $[a, b]$ consiste em dividir o intervalo $[a, b]$ em n intervalos de igual tamanho, $[x_0, x_1], [x_1, x_2], \dots, [x_{n-2}, x_{n-1}], [x_{n-1}, x_n]$, com $x_0 = a$, $x_n = b$, e $\forall i, j \quad x_i - x_{i-1} = x_j - x_{j-1}$. A área sob o gráfico da função será dada por (Figura 6.10):

$$\sum_{i=1}^n f\left(\frac{x_i + x_{i-1}}{2}\right) (x_i - x_{i-1})$$

Escreva em Python uma função de ordem superior que recebe como argumentos uma função (correspondente à função f) e os valores de a e b e que calcula o valor da área entre o gráfico da função e o eixo dos xx no intervalo $[a, b]$. A sua função poderá começar a calcular a área para

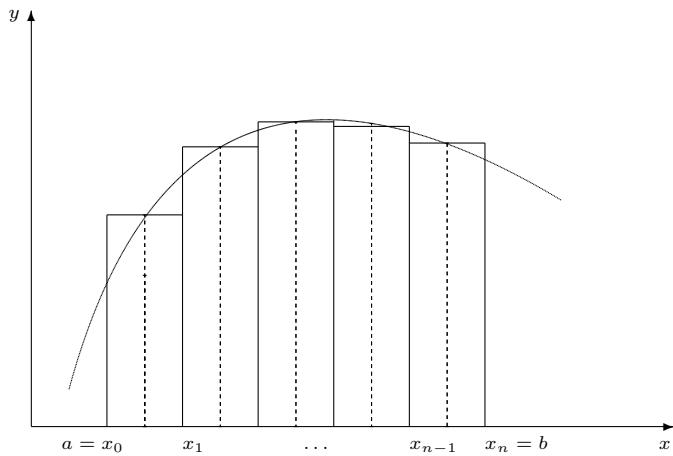


Figura 6.10: Área aproximada sob a curva.

o intervalo $[x_0, x_1] = [a, b]$ e ir dividindo sucessivamente os intervalos $[x_{i-1}, x_i]$ ao meio, até que o valor da área seja suficientemente bom.

Capítulo 7

Recursão e iteração

*'Well, I'll eat it,' said Alice, 'and if it makes me grow larger, I can reach the key; and if it makes me grow smaller, I can creep under the door: so either way I'll get into the garden, and I don't care which happens!'
She ate a little bit, and said anxiously to herself, 'Which way? Which way?'*

Lewis Carroll, *Alice's Adventures in Wonderland*

Nos capítulos anteriores considerámos alguns aspectos da programação, estudámos o modo de criar novas funções e utilizámos tipos estruturados de informação existentes em Python. Embora sejamos já capazes de escrever programas, o conhecimento que adquirimos ainda não é suficiente para podermos programar de um modo eficiente. Falta-nos saber que funções vale a pena definir e quais as consequências da execução de uma função.

Recordemos que a entidade básica subjacente à computação é o processo computacional. Na actividade de programação planeamos a sequência de acções a serem executadas por um programa. Para podermos desenvolver programas adequados a um dado fim é essencial que tenhamos uma compreensão clara dos processos computacionais gerados pelos diferentes tipos de funções. Este aspecto é abordado neste capítulo.

Uma função pode ser considerada como a especificação da evolução local de um processo computacional. Por *evolução local* entenda-se que a função define, em cada instante, o comportamento do processo computacional, ou seja, especifica

como construir cada estágio do processo a partir do estágio anterior. Ao abordar processos computacionais, queremos estudar a evolução global do processo cuja evolução local é definida por uma função¹. Neste capítulo, apresentamos alguns padrões típicos da evolução de processos computacionais, estudando a ordem de grandeza do número de operações associadas e o “espaço” exigido pela evolução global do processo.

7.1 Recursão linear

Começamos por considerar funções que geram processos que apresentam um padrão de evolução a que se chama recursão linear.

Consideremos a função `factorial` apresentada na Secção 6.1:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Vimos que durante a avaliação de uma expressão cujo operador é `factorial`, existem duas fases distintas: numa primeira fase, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica e numa segunda fase, as operações que foram adiadas são executadas.

Consideremos a definição da função `potencia` apresentada na Secção 6.3:

```
def potencia(x, n):
    if n == 0:
        return 1
    else:
        return x * potencia(x, n - 1)
```

Durante a avaliação de uma expressão cujo operador é `potencia`, gera-se também um encadeamento de operações suspensas à espera do valor de outras expressões

¹Note-se que, na realidade, a função também define a evolução global do processo.

cujo operador é a própria **potencia**. Este encadeamento de operações pode ser representado do seguinte modo:

```
potencia(2, 8)
2 * potencia(2, 7)
2 * (2 * potencia(2, 6))
2 * (2 * (2 * potencia(2,5)))
2 * (2 * (2 * (2 * potencia(2,4))))
2 * (2 * (2 * (2 * (2 * potencia(2, 3))))))
2 * (2 * (2 * (2 * (2 * (2 * potencia(2,2))))))
2 * (2 * (2 * (2 * (2 * (2 * (2 * potencia(2,1)))))))
2 * (2 * (2 * (2 * (2 * (2 * (2 * potencia(2, 0)))))))
2 * (2 * (2 * (2 * (2 * (2 * (2 * (2 * 1)))))))
2 * (2 * (2 * (2 * (2 * (2 * (2 * 2))))))
2 * (2 * (2 * (2 * (2 * (2 * 4))))))
2 * (2 * (2 * (2 * (2 * 8)))))
2 * (2 * (2 * (2 * 16)))))
2 * (2 * (2 * 32)))
2 * (2 * 64)
2 * 128
256
```

No processo para o cálculo de uma potência existem também duas fases distintas: na primeira, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função **potencia**, na segunda, as operações que foram adiadas são executadas.

Como terceiro exemplo, consideremos uma função, **soma_elementos**, para calcular a soma dos elementos de uma lista². Podemos escrever a seguinte função em Python para calcular a soma dos elementos de uma lista:

```
def soma_elementos(lst):
    if lst == []:
        return 0
    else:
        return lst[0] + soma_elementos(lst[1:])
```

²Consideramos que a lista contém números, pelo que não faremos qualquer teste ao seu conteúdo.

Ou seja, se a lista não tem elementos, a soma dos seus elementos é zero, em caso contrário, a soma dos seus elementos corresponde ao resultado de somar o primeiro elemento da lista ao resultado de calcular a soma dos restantes elementos da lista.

Podemos agora considerar a forma do processo gerado pela função `soma_elementos`:

```
soma_elementos([1, 2, 3, 4])
1 + soma_elementos([2, 3, 4])
1 + (2 + soma_elementos([3, 4]))
1 + (2 + (3 + soma_elementos([4])))
1 + (2 + (3 + (4 + soma_elementos([]))))
1 + (2 + (3 + (4 + 0)))
1 + (2 + (3 + 4))
1 + (2 + 7)
1 + 9
10
```

Voltamos a deparar-nos com um processo que apresenta duas fases distintas; numa primeira fase verifica-se uma expansão, devido à existência de operações cuja aplicação vai sendo sucessivamente adiada, seguida por uma fase de contracção na qual as operações que foram adiadas são executadas.

Embora as três funções que apresentámos tenham finalidades distintas, todas elas geram processos computacionais que têm um comportamento semelhante: são caracterizados por uma fase de expansão, devido à existência de operações adiadas, seguida por uma fase de contracção em que essas operações são executadas. Este padrão de evolução de um processo é muito comum em programação e tem o nome de processo recursivo.

Num *processo recursivo* existe uma fase de *expansão* correspondente à construção de uma cadeia de operações adiadas, seguida por uma fase de *contracção* correspondente à execução dessas operações. A informação acerca destas operações adiadas é mantida internamente pelo Python.

Nos três casos que apresentámos, o número de operações adiadas cresce linearmente com um determinado valor. Este valor pode corresponder a um dos

parâmetros da função (como é o caso do inteiro para o qual se está a calcular o factorial, do expoente relativamente à potência e do número de elementos da lista), mas pode também corresponder a outras coisas.

A um processo recursivo que cresce linearmente com um valor dá-se o nome de processo *recursivo linear*.

7.2 Iteração linear

Nesta secção consideramos de novo os problemas discutidos na secção anterior e apresentamos a sua solução através de funções que geram processos que apresentam um padrão de evolução a que se chama iteração linear. Após a análise dos processos gerados por estas funções, caracterizamos a iteração linear.

Suponhamos que somos postos perante o problema de calcular manualmente o factorial de um número, por exemplo 8!. O nosso processo de cálculo seria de novo bem diferente do processo recursivo linear apresentado na Secção 7.1. Começaríamos a efectuar multiplicações, por exemplo da esquerda para a direita, em lugar de adiar a sua realização: o nosso processo seguiria o raciocínio: oito vezes sete 56, vezes seis 336, vezes cinco 1680, vezes quatro 6720, vezes três 20160, vezes dois 40320. Durante o nosso processo temos de nos lembrar do valor do produto acumulado e, adicionalmente, de qual o próximo número a utilizar na multiplicação.

O nosso processo de cálculo pode ser traduzido pela seguinte função em Python, na qual `prod_ac` representa o produto acumulado dos números que já multiplicámos e `prox` representa o próximo número a ser multiplicado:

```
def factorial_aux(prod_ac, prox):
    if prox == 0:
        return prod_ac
    else:
        return factorial_aux (prod_ac * prox, prox - 1)
```

Com base nesta função, podemos agora escrever a seguinte função para o cálculo de factorial, a qual esconde do exterior a utilização das variáveis `prod_ac` e `prox`:

```
\index{factorial@\tt factorial}
```

```

def factorial(n):

    def factorial_aux(prod_ac, prox):
        if prox == 0:
            return prod_ac
        else:
            return factorial_aux (prod_ac * prox, prox - 1)

    return factorial_aux (1, n)

```

Durante a avaliação de uma expressão cujo operador é `factorial`, não se gera um encadeamento de operações suspensas e, consequentemente, o processo não se expande nem se contrai:

```

factorial(8)
factorial_aux(1, 8)
factorial_aux(8, 7)
factorial_aux(56, 6)
factorial_aux(336, 5)
factorial_aux(1680, 4)
factorial_aux(6720, 3)
factorial_aux(20160, 2)
factorial_aux(40320, 1)
factorial_aux(40320, 0)
40320

```

Adicionalmente, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado, `prod_ac`) e o que ainda nos falta fazer (o número de multiplicações a realizar, `prox`), informação essa que não existe explicitamente no processo recursivo linear. Isto significa que podemos interromper o processo computacional em qualquer instante e recomeçá-lo mais tarde, utilizando a informação disponível no instante em que este foi interrompido.

O raciocínio que utilizámos nesta segunda versão da função `factorial` é diretamente aplicado no desenvolvimento de uma versão desta função recorrendo à programação imperativa:

```
def factorial(n):
    prod_ac = 1
    for prox in range(1, n + 1):
        prod_ac = prod_ac * prox
    return prod_ac
```

Suponhamos agora que somos postos perante o problema de calcular manualmente uma potência de um número, por exemplo 2^8 . O nosso processo de cálculo seria bem diferente do processo recursivo linear. Começaríamos a efectuar multiplicações, em lugar de adiar a sua realização: o nosso processo seguiria o seguinte raciocínio: dois vezes dois 4, vezes dois 8, vezes dois 16, e assim sucessivamente. Durante a evolução do nosso processo, temos de nos lembrar do valor parcial que já calculámos para a potência e, adicionalmente, de quantas vezes já efectuámos a multiplicação.

Em resumo, este processo de cálculo corresponde a manter um valor para o produto acumulado num dado instante e, simultaneamente, saber quantas vezes ainda nos falta multiplicar. Cada vez que efectuamos uma multiplicação, o valor do produto acumulado altera-se (pois ele é multiplicado pela base) e o número de vezes que ainda temos de multiplicar diminui em uma unidade. O processo termina quando não falta multiplicar nenhuma vez.

A seguinte função simula o método de cálculo que acabámos de descrever. Na função `potencia_aux`, o nome `n_mult` corresponde ao número de multiplicações que temos de executar, e o nome `prod_ac` corresponde ao produto acumulado. Em cada passo, actualizamos simultaneamente os valores do número de multiplicações a efectuar e do produto acumulado. Ao iniciar o cálculo, estabelecemos que o produto acumulado é a base (`x`), e o número de multiplicações que nos falta efectuar corresponde ao expoente (`n`) menos um (embora não tenhamos feito qualquer operação, já contámos com a base no produto acumulado)³:

```
def potencia(x, n):

    def potencia_aux(x, n_mult, prod_ac):
        if n_mult == 0:
            return prod_ac
```

³Em alternativa, podíamos estabelecer como valores iniciais do produto acumulado e do número de multiplicações a efectuar, 1 e o expoente, respectivamente.

```

else:
    return potencia_aux (x, n_mult - 1, x * prod_ac)

return potencia_aux (x, n - 1, x)

```

Podemos agora representar a sequência de expressões que são avaliadas pelo Python, envolvendo estas duas funções, na sequência da avaliação de `potencia(2, 8)`:

```

potencia(2, 8)
potencia_aux(2, 7, 2)
potencia_aux(2, 6, 4)
potencia_aux(2, 5, 8)
potencia_aux(2, 4, 16)
potencia_aux(2, 3, 32)
potencia_aux(2, 2, 64)
potencia_aux(2, 1, 128)
potencia_aux(2, 0, 256)
256

```

Este processo apresenta um padrão muito diferente do encontrado no processo recursivo linear. Aqui não existem operações suspensas, e o processo não se expande nem se contrai.

Adicionalmente, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado) e o que ainda nos falta fazer (o número de multiplicações a realizar), informação essa que não existe explicitamente no processo recursivo linear. Isto significa que podemos interromper o processo computacional em qualquer instante e recomeçá-lo mais tarde, utilizando a informação disponível no instante em que este foi interrompido.

Novamente, podemos aplicar directamente o nosso raciocínio ao desenvolvimento de uma versão da função `potencia` recorrendo à programação imperativa:

```

def potencia(x, n):
    prod_ac = 1
    for n_mult in range(n, 0, -1):
        prod_ac = prod_ac * x
    return prod_ac

```

Finalmente, para calcular a soma dos elementos de uma lista, podemos utilizar um raciocínio semelhante aos anteriores. Teremos que manter um registo das somas dos elementos que já considerámos, o qual é representado pela variável `soma`, e quando a lista for vazia, o valor de `soma` corresponde à soma dos elementos da lista.

Este processo é traduzido pela seguinte função:

```
def soma_elementos(lst):

    def soma_elementos_aux(lst, soma):
        if lst == []:
            return soma
        else:
            return soma_elementos_aux(lst[1:], lst[0] + soma)

    return soma_elementos_aux(lst, 0)
```

Tal como nos dois casos anteriores, a avaliação da função `soma_elementos` origina um processo que não se expande nem se contrai. Em cada instante, toda a informação que é manipulada pelo processo está explicitamente disponível: a lista que está a ser considerada nesse instante e a soma dos elementos já encontrados:

```
soma_elementos([1, 2, 3, 4])
soma_elementos_aux([1, 2, 3, 4], 0)
soma_elementos_aux([2, 3, 4], 1)
soma_elementos_aux([3, 4], 3)
soma_elementos_aux([4], 6)
soma_elementos_aux([], 10)
10
```

Usando a programação imperativa, no nosso raciocínio origina a seguinte versão da função `soma_elementos`:

```
def soma_elementos(lst):
    soma = 0
    for e in lst:
```

```
soma = soma + e
return soma
```

Novamente, embora as três funções que apresentámos tenham finalidades distintas, todas elas geram processos computacionais que têm um comportamento semelhante: os processos não se expandem nem se contraem. Eles são caracterizados por um certo número de variáveis que fornecem uma descrição completa do estado da computação em cada instante. Esta situação não acontece num processo recursivo no qual o número de operações suspensas é “escondido” pelo Python. O que fizemos nas três funções anteriores foi tornar explícita a informação que estava escondida no processo recursivo correspondente. O padrão de evolução de um processo que descrevemos nesta secção é também muito comum em programação, e tem o nome de *processo iterativo*. Recorrendo ao Wikcionário⁴, a palavra “iterativo” tem o seguinte significado “Diz-se do processo que se repete diversas vezes para se chegar a um resultado e a cada vez gera um resultado parcial que será usado na vez seguinte”.

Um *processo iterativo* é caracterizado por um certo número de variáveis, chamadas *variáveis de estado*, juntamente com uma regra que especifica como actualizá-las. Estas variáveis fornecem uma descrição completa do estado da computação em cada momento. Um processo iterativo não se expande nem se contrai.

Nos nossos exemplos, o número de operações efectuadas sobre as variáveis de estado cresce linearmente com uma grandeza associada à função (o inteiro para o qual se está a calcular o factorial, o expoente no caso da potência e o o número de elementos da lista).

A um processo iterativo cujo número de operações cresce linearmente com um valor dá-se o nome de processo *iterativo linear*.

7.3 Recursão em processos e em funções

Da discussão apresentada no Capítulo 6 e na Secção 7.1 podemos concluir que a palavra “recursão” tem dois significados distintos, conforme se refere à recursão em funções ou à recursão em processos. A *recursão em funções* refere-se à de-

⁴<http://pt.wiktionary.org/wiki/>.

finição da função em termos de si própria, ao passo que a *recursão em processos* refere-se ao padrão de evolução do processo.

Podemos também concluir que a evolução de processos pode ser classificada como uma evolução recursiva ou como uma evolução iterativa:

1. Um *processo recursivo* é caracterizado por uma fase de expansão (correspondente à construção de uma cadeia de operações adiadas) seguida de uma fase de contracção (correspondente à execução dessas operações). O computador mantém informação “escondida” que regista o ponto onde está o processo na cadeia de operações adiadas.
2. Um *processo iterativo* não cresce nem se contrai. Este é caracterizado por um conjunto de variáveis de estado e um conjunto de regras que definem como estas variáveis evoluem. As variáveis de estado fornecem uma descrição completa do estado do processo em cada ponto.

Repare-se que uma função recursiva tanto pode gerar um processo recursivo como um processo iterativo. Por exemplo, tanto a função `potencia` da Secção 7.1 como a função `potencia_aux` da Secção 7.2 são funções recursivas (são definidas em termos de si próprias), no entanto a primeira gera um processo recursivo, e a segunda gera um processo iterativo.

7.4 Recursão em árvore

Nesta secção vamos considerar um outro padrão da evolução de processos que também é muito comum em programação, a recursão em árvore.

7.4.1 Os números de Fibonacci

Para ilustrar a recursão em árvore vamos considerar uma sequência de números descoberta no século XIII pelo matemático italiano Leonardo Fibonacci (c. 1170–c. 1250), também conhecido por Leonardo de Pisa, ao tentar resolver o seguinte problema:

“Quantos casais de coelhos podem ser produzidos a partir de um único casal durante um ano se cada casal originar um novo casal em cada mês, o qual se torna fértil a partir do segundo mês; e não ocorrerem mortes.”

Fibonacci chegou à conclusão que a evolução do número de casais de coelhos era ditada pela seguinte sequência: 0, 1, 1, 2, 3, 5, 8, 13, 21, ... Nesta sequência, conhecida por *sequência de Fibonacci*, cada termo, excepto os dois primeiros, é a soma dos dois anteriores. Os dois primeiros termos são respectivamente 0 e 1. Os números da sequência de Fibonacci são conhecidos por *números de Fibonacci*, e podem ser descritos através da seguinte definição:

$$fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fib(n - 1) + fib(n - 2) & \text{se } n > 1 \end{cases}$$

Suponhamos que desejávamos escrever uma função em Python para calcular os números de Fibonacci. Com base na definição anterior podemos produzir a seguinte função:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 2) + fib(n - 1)
```

Vejamos agora qual a forma do processo que é gerado para o cálculo de um número de Fibonacci, por exemplo, `fib(5)`. Assumindo que as sub-expressões numa expressão composta são avaliadas da esquerda para a direita⁵, obtemos a seguinte evolução:

```
fib(5)
fib(4) + fib(3)
(fib(3) + fib(2)) + fib(3)
((fib(2) + fib(1)) + fib(2)) + fib(3)
(((fib(1) + fib(0)) + fib(1)) + fib(2)) + fib(3)
(((1 + 0) + fib(1)) + fib(2)) + fib(3)
((1 + 1) + fib(2)) + fib(3)
(2 + (fib(1) + fib(0))) + fib(3)
(2 + (1 + 0)) + fib(3)
```

⁵No caso de uma ordem diferente de avaliação, a “forma” do processo é semelhante.

```

3 + fib(3)
3 + (fib(2) + fib(1))
3 + ((fib(1)+ fib(0)) + fib(1))
3 + ((1 + 0) + fib(1))
3 + (1 + fib(1))
3 + (1 + 1)
3 + 2
5

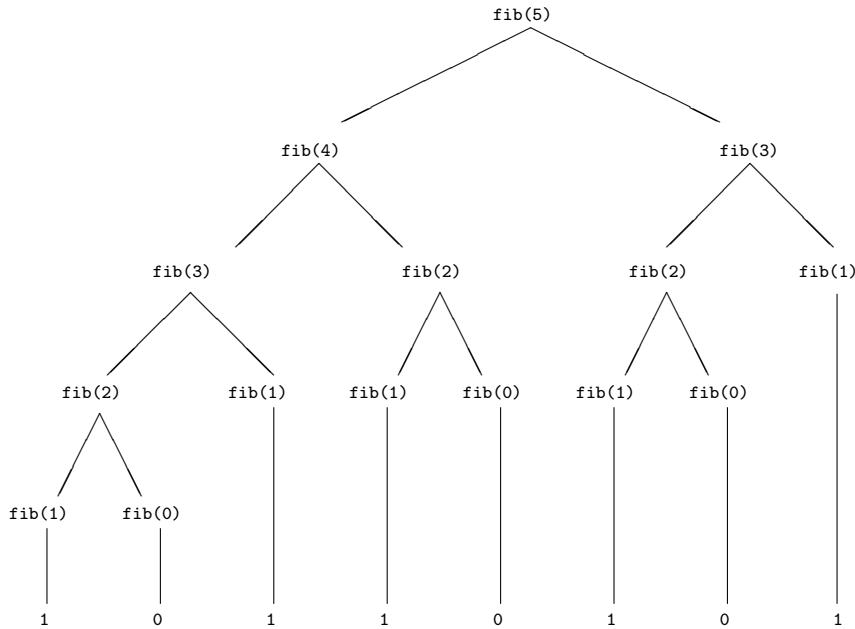
```

Ao analisarmos a forma do processo anterior, verificamos que esta não corresponde a nenhum dos padrões já estudados. No entanto, este apresenta um comportamento que se assemelha ao processo recursivo. Tem fases de crescimento, originadas por operações adiadas, seguidas por fases de contracção em que algumas das operações adiadas são executadas.

Ao contrário do que acontece com o processo recursivo linear, estamos perante a existência de múltiplas fases de crescimento e de contracção que são originadas pela dupla recursão que existe na função `fib` (esta refere-se duas vezes a si própria). A este tipo de evolução de um processo dá-se o nome de *recursão em árvore*. Esta designação deve-se ao facto de a evolução do processo ter a forma de uma árvore. Cada avaliação da expressão composta cujo operador é a função `fib` dá origem a duas avaliações (dois ramos de uma árvore), excepto para os dois últimos valores. Na Figura 7.1 apresentamos o resultado da árvore gerada durante o cálculo de `fib(5)`.

Estudando a evolução do processo anterior, verificamos facilmente que este é muito ineficiente, pois existem muitos cálculos que são repetidos múltiplas vezes. Para além disso, o processo utiliza um número de passos que não cresce linearmente com o valor de n . Demonstra-se que este número cresce exponencialmente com o valor de n .

Para evitar os cálculos repetidos, vamos agora calcular os números de Fibonacci através de um processo iterativo. Sabemos que cada termo da sequência de Fibonacci, excepto os dois primeiros, é a soma dos dois anteriores. Para calcular um dos termos da sequência teremos, pois, de saber os dois termos anteriores. Suponhamos que, no cálculo de um termo genérico f_n , os dois termos anteriores são designados por f_{n-2} e f_{n-1} . Neste caso, o próximo número de Fibonacci será dado por $f_n = f_{n-2} + f_{n-1}$ e a partir de agora os dois últimos termos são f_{n-1} e f_n . Podemos agora, usando o mesmo raciocínio, calcular o próximo termo,

Figura 7.1: Árvore gerada durante o cálculo de $\text{fib}(5)$.

f_{n+1} . Para completarmos o nosso processo teremos de explicitar o número de vezes que temos de efectuar estas operações.

Assim, podemos escrever a seguinte função que traduz o nosso processo de raciocínio. Esta função recebe os dois últimos termos da sequência (`f_n_2` e `f_n_1`) e o número de operações que ainda temos de efectuar (`cont`).

```

def fib_aux(f_n_2, f_n_1, cont):
    if cont == 0:
        return f_n_2 + f_n_1
    else:
        return fib_aux(f_n_1, f_n_2 + f_n_1, cont - 1)
  
```

Para utilizar esta função devemos indicar os dois primeiros termos da sequência de Fibonacci:

```

def fib(n):
    if n == 0:
        return 0
  
```

```
elif n == 1:  
    return 1  
else:  
    return fib_aux (0, 1, n - 2)
```

Esta função origina um processo iterativo linear em que não há duplicações de cálculos:

```
fib(5)  
fib_aux(0, 1, 3)  
fib-aux(1, 1, 2)  
fib-aux(1, 2, 1)  
fib-aux(2, 3, 0)  
5
```

Em programação imperativa, a função `fib` poderá ser definida do seguinte modo:

```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        f_n_2 = 0  
        f_n_1 = 1  
        cont = 0  
        while cont < n - 1:  
            f_n_2, f_n_1 = f_n_1, f_n_2 + f_n_1  
            cont = cont + 1  
        return f_n_1
```

Deste exemplo não devemos concluir que a recursão em árvore é um processo inútil. Na próxima seção apresentamos um outro problema para o qual a recursão em árvore corresponde ao método ideal para a sua resolução.



Figura 7.2: Torre de Hanói com três discos.

7.4.2 A torre de Hanói

Apresentamos um programa em Python para a solução de um *puzzle* chamado a Torre de Hanói. A *Torre de Hanói* é constituída por 3 postes verticais, nos quais podem ser colocados discos de diâmetros diferentes, furados no centro, variando o número de discos de *puzzle* para *puzzle*. O *puzzle* inicia-se com todos os discos num dos postes (tipicamente, o poste da esquerda), com o disco menor no topo e com os discos ordenados, de cima para baixo, por ordem crescente dos respectivos diâmetros, e a finalidade é movimentar todos os discos para um outro poste (tipicamente, o poste da direita), também ordenados por ordem crescente dos respectivos diâmetros, de acordo com as seguintes regras: (1) apenas se pode movimentar um disco de cada vez; (2) em cada poste, apenas se pode movimentar o disco de cima; (3) nunca se pode colocar um disco sobre outro de diâmetro menor.

Este *puzzle* está associado a uma lenda, segundo a qual alguns monges num mosteiro perto de Hanói estão a tentar resolver um destes *puzzles* com 64 discos, e no dia em que o completarem será o fim do mundo. Não nos devemos preocupar com o fim do mundo, pois se os monges apenas efectuarem movimentos perfeitos à taxa de 1 movimento por segundo, demorarão perto de mil milhões de anos para o resolver⁶.

Na Figura 7.2 apresentamos um exemplo das configurações inicial e final para a Torre de Hanói com três discos.

Suponhamos então que pretendíamos escrever um programa para resolver o *puzzle* da Torre de Hanói para um número n de discos (o valor de n será fornecido pelo utilizador). Para resolver o *puzzle* da Torre de Hanói com n discos ($n > 1$), teremos de efectuar basicamente três passos:

1. Movimentar $n - 1$ discos do poste da esquerda para o poste do centro

⁶Ver [Raphael, 1976], página 80.

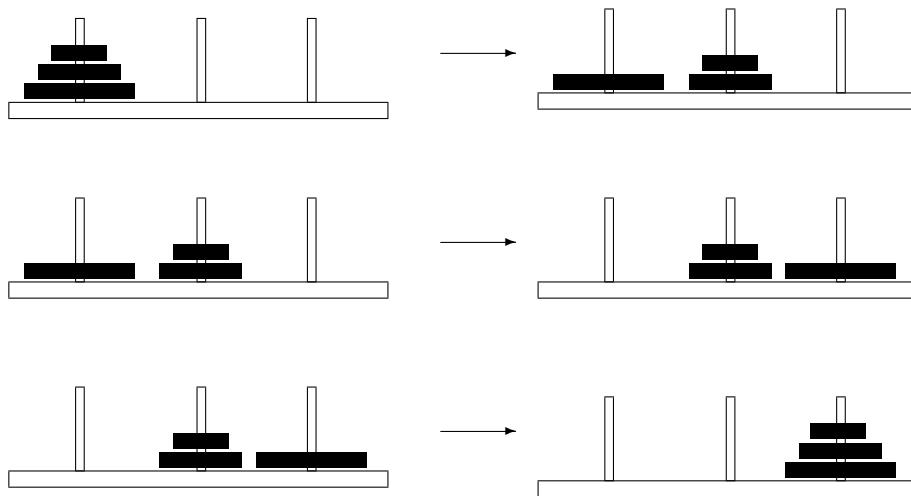


Figura 7.3: Solução da Torre de Hanói com três discos.

- (utilizado como poste auxiliar);
2. Movimentar o disco do poste da esquerda para o poste da direita;
 3. Movimentar os $n - 1$ discos do poste do centro para o poste da direita.

Estes passos encontram-se representados na Figura 7.3 para o caso de $n = 3$.

Com este método conseguimos reduzir o problema de movimentar n discos ao problema de movimentar $n - 1$ discos, ou seja, o problema da movimentação de n discos foi descrito em termos do mesmo problema, mas tendo um disco a menos. Temos aqui um exemplo típico de uma solução recursiva. Quando n for igual a 1, o problema é resolvido trivialmente, movendo o disco da origem para o destino.

Como primeira aproximação, podemos escrever a função `mova` para movimentar n discos. Esta função tem como argumentos o número de discos a mover e uma indicação de quais os postes de origem e destino dos discos, bem como qual o poste que deve ser usado como poste auxiliar:

```
def mova(n, origem, destino, aux):
    if n == 1:
        mova_disco(origem, destino)
```

```
mov(a(3, 'E', 'D', 'C')
```

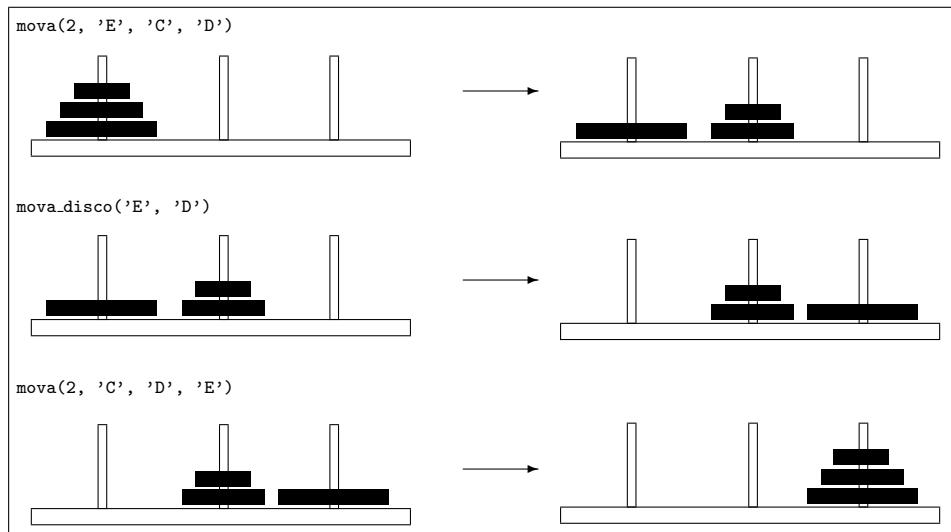


Figura 7.4: Subproblemas gerados por `mov(a(3, 'E', 'D', 'C'))`.

```
else:
    mov(a(n-1, origem, aux, destino)
        mov(a(1, origem, destino)
        mov(a(n-1, aux, destino, origem)
```

Antes de continuar, convém observar que a instrução composta que corresponde ao `else` é constituída por três instruções. Dentro destas instruções, as duas utilizações da função `mov(a)` têm os argumentos correspondentes aos postes, por ordem diferente, o que corresponde a resolver dois outros *puzzles* em que os postes de origem, de destino e auxiliares são diferentes. Na Figura 7.4 apresentamos as três expressões que são originadas por `mov(a(3, 'E', 'D', 'C'))`, ou seja, mova três discos do poste da esquerda para o poste da direita, utilizando o poste do centro como poste auxiliar, bem como uma representação dos diferentes subproblemas que estas resolvem.

Esta função reflecte o desenvolvimento do topo para a base: o primeiro passo para a solução de um problema consiste na identificação dos subproblemas que o constituem, bem como a determinação da sua inter-relação. Escreve-se então uma primeira aproximação da solução em termos destes subproblemas. No nosso

exemplo, o problema da movimentação de n discos foi descrito em termos de dois subproblemas: o problema da movimentação de um disco e o problema da movimentação de $n - 1$ discos, e daí a solução recursiva.

Podemos agora escrever a seguinte função que fornece a solução para o *puzzle* da Torre de Hanói para um número arbitrário de discos:

```
def hanoi():

    def move(n, origem, destino, aux):

        def move_disc(d, p):
            print(d, '→', p)

        if n == 1:
            move_disc(origem, destino)
        else:
            move(n-1, origem, aux, destino)
            move_disc(origem, destino)
            move(n-1, aux, destino, origem)

    n = eval(input('Quantos discos deseja considerar?\n? '))
    print('Solução do puzzle:')
    move(n, 'E', 'D', 'C')
```

Esta função permite originar a seguinte interacção:

```
>>> hanoi()
Quantos discos deseja considerar?
? 3
Solução do puzzle:
E → D
E → C
D → C
E → D
C → E
C → D
E → D
```

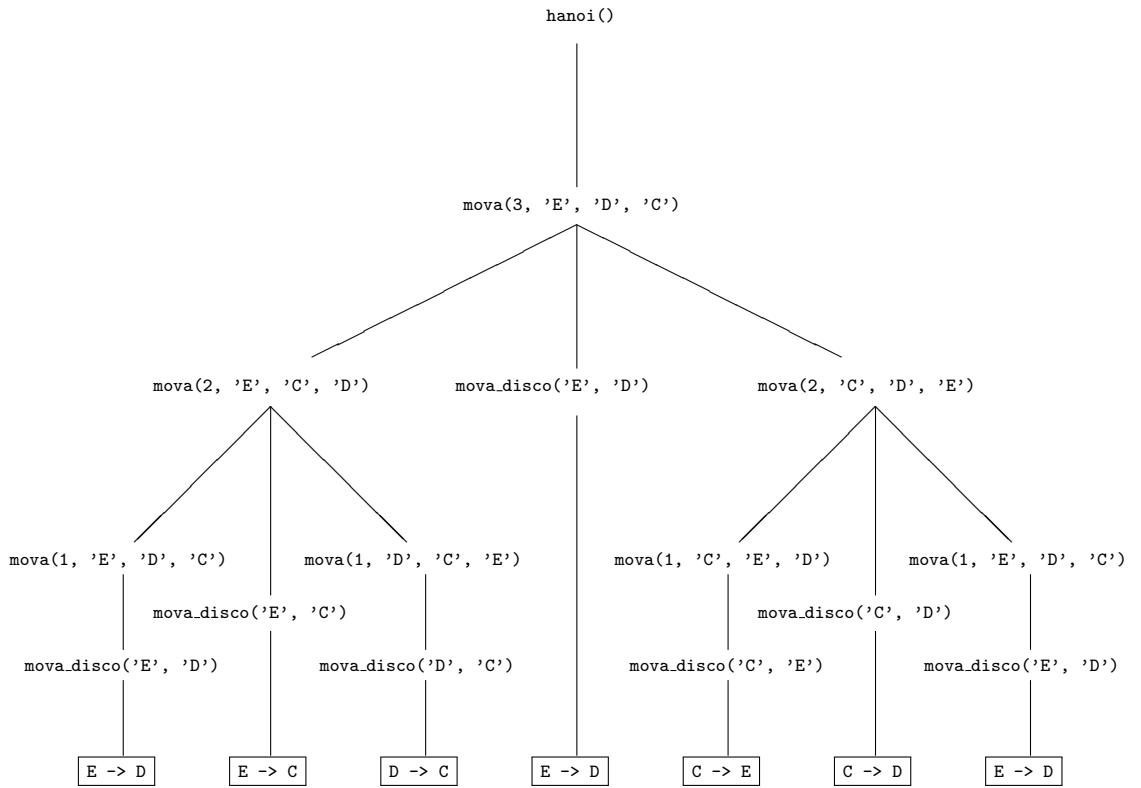


Figura 7.5: Árvore gerada por `hanoi()` com três discos

Na Figura 7.5 apresentamos a árvore gerada pela execução da função `hanoi()` quando o utilizador fornece o valor 3 para o número de discos. Nesta figura apresentam-se dentro de um rectângulo os valores que o Python escreve.

Estamos de novo perante uma função recursiva que origina um processo recursivo em árvore. Este exemplo mostra como a recursão pode dar origem a funções que são fáceis de escrever e de compreender.

7.5 Considerações sobre eficiência

Os exemplos apresentados neste capítulo mostram que os processos gerados por funções podem diferir drasticamente quanto à taxa a que consomem recursos computacionais. Assim, um dos aspectos que vamos ter de levar em linha de

conta quando escrevemos programas é a minimização dos recursos computacionais consumidos. Os recursos computacionais que aqui vamos considerar são o tempo e o espaço. O *tempo* diz respeito ao tempo que o nosso programa demora a executar, e o *espaço* diz respeito ao espaço de memória do computador usado pelo nosso programa.

Desta discussão e dos exemplos apresentados neste capítulo, podemos concluir que os processos recursivos lineares têm ordem $O(n)$, quer para o espaço, quer para o tempo, e que os processos iterativos lineares têm ordem $O(1)$ para o espaço e $O(n)$ para o tempo.

Tendo em atenção as preocupações sobre os recursos consumidos por um processo, apresentamos uma alternativa para o cálculo de potência, a qual gera um processo com ordem de crescimento inferior ao que apresentámos. Para compreender o novo método de cálculo de uma potência, repare-se que podemos definir potência, do seguinte modo:

$$x^n = \begin{cases} x & \text{se } n = 1 \\ x.(x^{n-1}) & \text{se } n \text{ for ímpar} \\ (x^{n/2})^2 & \text{se } n \text{ for par} \end{cases}$$

o que nos leva a escrever a seguinte função para o cálculo da potência:

```
def potencia_rapida(x, n):
    if n == 1:
        return x
    elif impar(n):
        return x * potencia_rapida(x, n - 1)
    else:
        return quadrado(potencia_rapida(x, n // 2))
```

Nesta função, `quadrado` e `impar` correspondem às funções:

```
def quadrado(x):
    return x * x

def impar(x):
    return (x % 2) == 1
```

Com este processo de cálculo, para calcular x^{2^n} precisamos apenas de mais uma multiplicação do que as que são necessárias para calcular x^n . Geramos assim um processo cuja ordem temporal é $O(\log_2(n))$. Para apreciar a vantagem desta função, notemos que, para calcular uma potência de expoente 1 000, a função `potencia` necessita de 1 000 multiplicações, ao passo que a função `potencia_rapida` apenas necessita de 14.

7.6 Notas finais

Neste capítulo apresentámos as motivações para estudar os processos gerados por procedimentos e caracterizámos alguns destes processos, nomeadamente os processos recursivos lineares, iterativos lineares e recursivos em árvore. Definimos a noção de ordem de crescimento e comparámos as taxas a que alguns processos consomem recursos.

7.7 Exercícios

1. Considere a seguinte função:

```
def m_p(x, y):
    def m_p_a(z):
        if z == 0:
            return 1
        else:
            return m_p_a(z - 1) * x
    return m_p_a(y)
```

- (a) Apresente a evolução do processo na avaliação de `m_p(2, 4)`.
- (b) A função gera um processo recursivo ou iterativo? Justifique a sua resposta.
- (c) Se a função gerar um processo recursivo, escreva uma função equivalente que gere um processo iterativo; se a função gerar um processo iterativo, escreva uma função equivalente que gere um processo recursivo.

2. Considere a função de Ackermann⁷:

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

- (a) Escreva em Python uma função para calcular o valor da função de Ackermann.
- (b) Siga o processo gerado pelo cálculo de $A(2, 2)$.
- (c) Qual o tipo de processo gerado pela sua função? Justifique a sua resposta.

3. Suponha que em Python não existia o operador de multiplicação, `*`, mas que existiam os operadores de adição, `+`, e divisão inteira, `//`. O produto de dois números inteiros positivos pode ser definido através da seguinte fórmula:

$$x.y = \begin{cases} 0 & \text{se } x = 0 \\ \frac{x}{2} \cdot (y + y) & \text{se } x \text{ é par} \\ y + (x - 1) \cdot y & \text{em caso contrário} \end{cases}$$

- (a) Defina a função `produto` que calcula o produto de dois números inteiros positivos, de acordo com esta definição.
- (b) A sua função é recursiva? Justifique.
- (c) A sua função gera um processo recursivo ou iterativo? Justifique a sua resposta.
- (d) Se o processo gerado era iterativo, transforme a função, de forma a que esta gere um processo recursivo. Se o processo gerado era recursivo, transforme a função, de forma a que esta gere um processo iterativo.

4. Considere a função g , definida para inteiros não negativos do seguinte modo:

$$g(n) = \begin{cases} 0 & \text{se } n = 0 \\ n - g(g(n - 1)) & \text{se } n > 0 \end{cases}$$

- (a) Escreva uma função recursiva em Python para calcular o valor de $g(n)$.

⁷Tal como apresentada em [Machtey e Young, 1978], página 24.

- (b) Siga o processo gerado por $g(3)$, indicando todos os cálculos efectuados.
- (c) Que tipo de processo é gerado por esta função?
5. O número de combinações de m objectos n a n pode ser dado pela seguinte fórmula:

$$C(m, n) = \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } m = n \\ 0 & \text{se } m < n \\ C(m - 1, n) + C(m - 1, n - 1) & \text{se } m > n, m > 0 \text{ e } n > 0 \end{cases}$$

- (a) Escreva uma função em Python para calcular o número de combinações de m objectos n a n .
- (b) Que tipo de processo é gerado pela sua função?

Capítulo 8

Ficheiros

'They're putting down their names,' the Gryphon whispered in reply, 'for fear they forget them before the end of the trial.'

Lewis Carroll, *Alice's Adventures in Wonderland*

Os programas que temos desenvolvido até aqui têm uma única fonte de dados, o teclado, e um único destino para os seus resultados, o ecrã. Para além disso, quando o programa termina, os dados usados e produzidos pelo programa desaparecem.

A maior parte dos programas desenvolvidos na vida real têm múltiplas fontes de dados e múltiplos destinos para os seus resultados. Muitos dos dados dos programas utilizados na vida real são *persistentes*, no sentido em que eles existem, independentemente da execução do programa, armazenados no disco do computador, numa memória USB ou num CD. A estrutura tipicamente utilizada para armazenar esta informação é chamada um ficheiro.

Um *ficheiro*¹ é um tipo estruturado de informação constituído por uma sequência de elementos, todos do mesmo tipo. Nos ficheiros que consideramos neste capítulo, os elementos são acedidos sequencialmente, ou seja, para aceder ao n -ésimo elemento do ficheiro, teremos primeiro de aceder aos $n - 1$ elementos que se encontram antes dele.

Os ficheiros diferem dos outros objectos computacionais considerados até aqui

¹Em inglês, “file”.

em dois aspectos: (1) os seus valores poderem existir independentemente de qualquer programa, um ficheiro pode existir antes do início da execução de um programa e manter a sua existência após o fim da sua execução; (2) um ficheiro encontra-se, em qualquer instante, num de dois estados possíveis, ou está a ser utilizado para a entrada de dados (estão a ser lidos valores do ficheiro, caso em que se diz que o ficheiro se encontra em *modo de leitura*) ou está a ser utilizado para a saída de dados (estão a ser escritos valores no ficheiro, caso em que se diz que o ficheiro se encontra em *modo de escrita*). Ao utilizarmos um ficheiro, temos de dizer ao Python em que modo queremos que esse ficheiro se encontre e qual a localização física dos ficheiros em que os dados se encontram.

8.1 O tipo ficheiro

Sendo os ficheiros entidades que existem fora do nosso programa, antes de utilizar um ficheiro é necessário identificar qual a localização física deste e o modo como o queremos utilizar, ou seja se queremos ler a informação contida no ficheiro ou se queremos escrever informação no ficheiro.

A operação através da qual identificamos a localização do ficheiro e o modo como o queremos utilizar é conhecida por *operação de abertura do ficheiro* e é realizada em Python recorrendo à função embutida `open`. A função `open` tem a seguinte sintaxe:

```
open(<expressão>, <modo>{, encoding = <tipo>})
```

Esta função tem dois argumentos obrigatórios e um opcional:

- o primeiro argumento, representado por `<expressão>`, é uma expressão cujo valor é uma cadeia de caracteres que corresponde ao nome externo do ficheiro;
- o segundo argumento, representado por `<modo>`, é uma expressão cujo valor é uma das cadeias de caracteres '`r`', '`w`' ou '`a`'.² Ou seja,

$\langle \text{modo} \rangle ::= 'r' \mid 'w' \mid 'a'$

²Existem outras alternativas que não são tratadas neste livro.

significando a primeira alternativa que o ficheiro é aberto para leitura³, a segunda alternativa que o ficheiro é aberto para escrita a partir do início do ficheiro⁴ e a terceira alternativa que o ficheiro é aberto para escrita a partir do fim do ficheiro⁵;

- o terceiro argumento, o qual é opcional, é da forma `encoding = <tipo>`, em que `tipo` é uma expressão que representa o tipo de codificação de caracteres utilizado no ficheiro.

O valor da função `open` corresponde à entidade no programa que está associada ao ficheiro.

8.2 Leitura de ficheiros

Ao abrir um ficheiro para leitura, está subentendido que esse ficheiro existe, pois queremos ler a informação que ele contém. Isto significa que se o Python for instruído para abrir para leitura um ficheiro que não existe, irá gerar um erro como mostra a seguinte interacção:

```
>>> f = open('nada', 'r')
builtins.IOError: [Errno 2] No such file or directory: 'nada'
```

Suponhamos que a directória (ou pasta) do nosso computador que contém os ficheiros utilizados pelo Python continha um ficheiro cujo nome é `teste.txt` e cujo o conteúdo corresponde ao seguinte texto:

```
Este é um teste
que mostra como o Python
lê ficheiros de caracteres
```

A execução pelo Python da instrução

```
t = open('teste.txt', 'r', encoding = 'UTF-16')
```

³,`r`, é a primeira letra da palavra inglesa “read” (lê).

⁴,`w`, é a primeira letra da palavra inglesa “write” (escreve).

⁵,`a`, é a primeira letra da palavra inglesa “append” (junta).

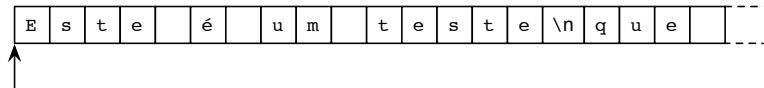


Figura 8.1: Representação do ficheiro `teste.txt`.

cria uma variável no nosso programa cujo nome é `t`, variável essa que está associada ao ficheiro `teste.txt`, o qual pode ser lido pelo Python, sabendo o Python que o texto neste ficheiro está codificado usando o código UTF-16.

Ao ler informação de um ficheiro, o Python mantém um indicador, o *indicador de leitura*, que indica qual o próximo elemento a ser lido do ficheiro (este indicador é representado nas nossas figuras por uma seta colocada imediatamente por baixo da posição em que se encontra no ficheiro). O indicador de leitura é colocado no início do ficheiro quando o ficheiro é aberto para leitura e movimenta-se no sentido do início para o fim do ficheiro sempre que se efectua uma leitura, sendo colocado imediatamente após o último símbolo lido, de cada vez que a leitura é feita.

Na Figura 8.1 mostramos parte do conteúdo do ficheiro `teste.txt`. Cada um dos elementos deste ficheiro é um carácter, e está representado na figura dentro de um quadrado. Este ficheiro corresponde a uma sequência de caracteres e contém caracteres que não são por nós visíveis quando o inspeccionamos num ecrã e que indicam o fim de cada uma das linhas. Apresentámos na Tabela 2.8 alguns destes caracteres, sendo o fim de linha representado por `\n`. Assim, no ficheiro, imediatamente após a cadeia de caracteres '`Este é um teste`', surge o carácter `\n` que corresponde ao fim da primeira linha do ficheiro.

A partir do momento que é criada uma variável, que designaremos por `<fich>`, associada a um ficheiro aberto para leitura, passam a existir quatro novas funções no nosso programa para efectuar operações sobre esse ficheiro⁶:

1. `<fich>.readline()`. Esta função lê a linha do ficheiro `<fich>` que se encontra imediatamente a seguir ao indicador de leitura, tendo como valor a cadeia de caracteres correspondente à linha que foi lida. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor '' (a cadeia de

⁶Os nomes destas funções seguem a sintaxe de `(nome composto)` apresentada na página 110, mas na realidade são um tipo de entidades diferentes. Este aspecto é abordado na Secção 11.3. Para os efeitos deste capítulo, iremos considerá-las como simples funções.

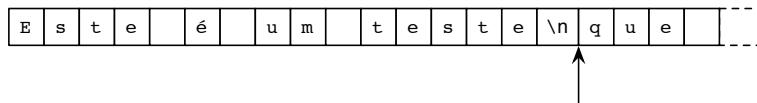


Figura 8.2: Ficheiro `teste.txt` após a execução de `t.readline()`.

caracteres vazia). No nosso exemplo, a função que lê uma linha do ficheiro corresponde a `t.readline()`.

2. `<fich>.readlines()`. Esta função lê todos os caracteres do ficheiro `<fich>` que se encontram depois do indicador de leitura, tendo como valor uma lista em que cada um dos elementos é a cadeia de caracteres correspondente a cada uma das linhas que foi lida. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor `[]` (a lista vazia). No nosso exemplo, esta função corresponde a `t.readlines()`.
3. `<fich>.read()`. Esta função lê todos os caracteres do ficheiro `<fich>` que se encontram depois do indicador de leitura, tendo como valor uma cadeia de caracteres contendo todos os caracteres lidos. Se o indicador de leitura se encontrar no fim do ficheiro, esta função tem o valor `''` (a cadeia de caracteres vazia). No nosso exemplo, esta função corresponde a `t.read()`.
4. `<fich>.close()`. Esta função fecha o ficheiro `<fich>`. A *operação de fecho de um ficheiro* corresponde a desfazer a ligação entre o programa e o ficheiro. No nosso exemplo, a função que fecha o ficheiro corresponde a `t.close()`.

Voltando ao nosso exemplo, suponhamos que abrimos o ficheiro `teste.txt` com a instrução `t = open('teste.txt', 'r', encoding = 'UTF-16')`. Se executarmos a instrução `ln1 = t.readline()`, a variável `ln1` passa a ter como valor a cadeia de caracteres `'Este é um teste\n'`, ficando o indicador de leitura como se mostra na Figura 8.2. Repare-se que o carácter de fim de linha faz parte da cadeia de caracteres lida.

A seguinte interacção mostra a utilização da função `t.readline()`. É importante notar que ao atingir o fim do ficheiro, a função `t.readline()` tem como valor a cadeia de caracteres vazia, `''`. Este aspecto é importante quando o nosso programa está a ler um ficheiro e necessita de determinar quando a leitura chega ao fim.

```
>>> t = open('teste.txt', 'r', encoding = 'UTF-16')
>>> ln1 = t.readline()
>>> ln1
'Este é um teste\n'
>>> ln2 = t.readline()
>>> ln2
'que mostra como o Python\n'
>>> ln3 = t.readline()
>>> ln3
'lê ficheiros de caracteres\n'
>>> ln4 = t.readline()
>>> ln4
 ''
>>> t.close()
```

A seguinte interacção mostra a utilização das funções disponíveis para ler ficheiros, indicando o comportamento de cada uma delas ao serem avaliadas com situações diferentes relativas ao indicador de leitura.

```
>>> f = open('teste.txt', 'r', encoding = 'UTF-16')
>>> l1 = f.readline()
>>> l1
'Este é um teste\n'
>>> l2 = f.read()
>>> l2
'que mostra como o Python\nlê ficheiros de caracteres\n'
>>> print(l2)
que mostra como o Python
lê ficheiros de caracteres

>>> f.close()
>>> g = open('teste.txt', 'r', encoding = 'UTF-16')
>>> lines = g.readlines()
>>> lines
['Este é um teste\n', 'que mostra como o Python\n',
 'lê ficheiros de caracteres\n']
```

Vamos agora analisar de uma forma mais detalhada os argumentos da função `open`. O nome do ficheiro utilizado por esta função corresponde à especificação completa do nome do ficheiro a utilizar. No exemplo que apresentámos, o ficheiro `teste.txt` existia na directória (ou pasta) do nosso computador que é utilizada por omissão pelo Python. Contudo, se este ficheiro não existisse na directória de omissão, mas sim numa directória chamada `exemplos` localizada na directória de omissão do Python, o nome do ficheiro a especificar na função `open` seria '`exemplos/teste.txt`'. Ou seja, o primeiro argumento da função `open` não é o nome de um ficheiro, mas sim a combinação de um caminho de directórias e do nome de um ficheiro. Embora diferentes sistemas operativos usem símbolos diferentes para especificar caminhos em directórias (o Mac OS e o Linux usam o símbolo `/`, ao passo que o Windows usa o símbolo `\`), na função `open` é sempre utilizado o símbolo `/`.

O argumento da função `open` associado à codificação de caracteres é ligeiramente mais complicado. Sabemos que os caracteres correspondem a símbolos e que estes são representados internamente em Python utilizando o *Unicode* (este aspecto foi discutido na Secção 4.3). Internamente ao Python, uma cadeia de caracteres é uma sequência de zeros e uns correspondente a representações de caracteres usando o *Unicode*. Contudo, um ficheiro existente no disco não é uma sequência de símbolos codificados em *Unicode*, mas apenas uma sequência de zeros e uns. Ao ler um ficheiro de texto existente num disco, o Python precisa de saber como “interpretar” a sequência de zeros e uns nele contida. Com base nesta informação, o Python descodifica a sequência de zeros e uns existente no disco, devolvendo uma sequência de caracteres em *Unicode*, que é identificada como uma cadeia de caracteres.

Para tornar as coisas ainda mais complicadas, não só a codificação de informação em disco é dependente do tipo de computador, mas também, dentro do mesmo computador existem normalmente ficheiros com diferentes tipos de codificação. Os sistemas operativos lidam com esta diversidade de codificações porque cada ficheiro contém informação sobre a codificação que este utiliza, informação essa que não está acessível ao Python. Por estas razões, ao abrir um ficheiro, é necessário dizer ao Python qual o tipo de codificação utilizada.

8.3 Escrita em ficheiros

De modo a escrevermos informação num ficheiro, teremos primeiro que efectuar a abertura do ficheiro com um dos modos '`w`' ou '`a`'. Ao abrir um ficheiro para escrita, se o ficheiro não existir, ele é criado pelo Python como um ficheiro sem elementos. Tal como no caso da leitura em ficheiros, ao escrever informação num ficheiro, o Python mantém um indicador, o *indicador de escrita*, que indica qual a posição do próximo elemento a ser escrito no ficheiro. Consoante o modo escolhido para a abertura do ficheiro, o Python coloca o indicador de escrita ou no início do ficheiro (ou seja, o ficheiro fica sem quaisquer elementos, e o seu antigo conteúdo, se existir, é perdido), se for utilizado o modo '`w`', ou no fim do ficheiro, se for utilizado o modo '`a`'.

De um modo semelhante ao que acontece quando abrimos um ficheiro em modo de leitura, a partir do momento que é criada uma variável, que designaremos por `<fich>`, associada a um ficheiro aberto para escrita, passam a existir as seguintes funções no nosso programa para efectuar operações sobre esse ficheiro:

1. `<fich>.write(<cadeia de caracteres>)`. Esta função escreve, a partir da posição do indicador de escrita, a `<cadeia de caracteres>` no ficheiro `<fich>`. O indicador de escrita é movimentado para a posição imediatamente a seguir à cadeia de caracteres escrita. Esta função devolve o número de caracteres escritos no ficheiro.
2. `<fich>.writelines(<sequência>)`, na qual `<sequência>` é um tuplo ou uma lista cujos elementos são cadeias de caracteres. Esta função escreve, a partir da posição do indicador de escrita, cada um dos elementos da `<sequência>` no ficheiro `<fich>`, não escrevendo o carácter de fim de linha entre os elementos escritos. O indicador de escrita é movimentado para a posição imediatamente a seguir à última cadeia de caracteres escrita. Esta função não devolve nenhum valor.
3. `<fich>.close()`. Esta função fecha o ficheiro `<fich>`.

A seguinte interacção mostra a utilização de operações de escrita e de leitura num ficheiro. Poderá parecer estranho a não utilização da indicação sobre a codificação dos caracteres utilizada no ficheiro. Contudo, como os ficheiros utilizados são criados pelo Python, a não indicação da codificação significa que esta será a codificação usada por omissão pelo Python.

```

>>> f1 = open('teste1', 'w')
>>> f1.write('abc')
3
>>> f1.write('def')
3
>>> f1.close()
>>> f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
>>> cont = f1.read()
>>> print(cont)
>>> abcdef
>>> cont # inspecção do conteúdo do ficheiro f1
'abcdef'
>>> f1.close()
>>> # 'teste1' (já existente) é aberto para escrita
>>> f1 = open('teste1', 'w')
>>> f1.close()
>>> f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
>>> cont = f1.read()
>>> cont # o conteúdo do ficheiro foi apagado
 '',
>>> f1.close()

```

A interacção anterior mostra que se um ficheiro existente é aberto para escrita, o seu conteúdo é apagado com a operação de abertura do ficheiro.

Após a abertura de um ficheiro para escrita, é também possível utilizar a função `print` apresentada na Secção 2.5.2, a qual, na realidade, tem uma sintaxe definida pelas seguintes expressões em notação BNF:

```

⟨escrita de dados⟩ ::= print() |
                      print(file = ⟨nome de ficheiro⟩) |
                      print(⟨expressões⟩) |
                      print(⟨expressões⟩, file = ⟨nome de ficheiro⟩)

⟨nome de ficheiro⟩ ::= ⟨nome⟩

```

Para as duas novas alternativas aqui apresentadas, a semântica desta função é definida do seguinte modo: ao encontrar a invocação da função `print(file = ⟨nome⟩)`, o Python escreve uma linha em branco no ficheiro `⟨nome⟩`; ao encontrar

a invocação da função `print(<exp1>, ... <expn>, file = <nome>)`, o Python avalia cada uma das expressões `<exp1> ... <expn>`, escrevendo-as na mesma linha do ficheiro `<nome>`, separadas por um espaço em branco e terminando com um salto de linha.

A seguinte interacção mostra a utilização da função `print` utilizando nomes de ficheiros:

```
>>> f1 = open('teste1', 'w') # 'teste1' é aberto para escrita
>>> f1.write('abc')
3
>>> print('cde', file = f1)
>>> print('fgh', 5, 5 * 5, file = f1)
>>> print('ijk', file = f1)
>>> f1.close()
>>> f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
>>> cont = f1.read()
>>> cont # conteúdo de 'teste1'
'abccde\nfgh 5 25\nijk\n'
>>> print(cont)
print('abccde\nfgh 5 25\nijk\n')
abccde
fgh 5 25
ijk

>>> f1.close()
>>> f1 = open('teste1', 'a') # 'teste1' é aberto para adição
>>> print(file = f1)
>>> print(file = f1)
>>> f1.write('lmn')
3
>>> f1.close()
>>> f1 = open('teste1', 'r') # 'teste1' é aberto para leitura
>>> cont = f1.read()
>>> cont # conteúdo de 'teste1'
'abccde\nfgh 5 25\nijk\n\n\nlmn'
>>> print(cont)
```

```
abccde  
fgh 5 25  
ijk
```

```
lmn  
>>> f1.close()
```

8.4 Notas finais

Apresentámos o conceito de ficheiro, considerando apenas ficheiros de texto. Um ficheiro corresponde a uma entidade que existe no computador independentemente da execução de um programa.

Para utilizar um ficheiro é necessário abrir o ficheiro, ou seja, associar uma entidade do nosso programa com um ficheiro físico existente no computador ou na rede a que o computador está ligado e dizer qual o tipo de operações a efectuar no ficheiro, a leitura ou a escrita de informação.

Depois de aberto um ficheiro apenas é possível efectuar operações de leitura ou de escrita, dependendo do modo como este foi aberto.

8.5 Exercícios

1. Escreva uma função em Python que recebe duas cadeias de caracteres, que correspondem a nomes de ficheiros. Recorrendo a listas, a sua função, lê o primeiro ficheiro, linha a linha, e calcula quantas vezes aparece cada uma das vogais. Após a leitura, a sua função escreve no segundo ficheiro, uma linha por vogal, indicando a vogal e o número de vezes que esta apareceu. Apenas conte as vogais que são letras minúsculas. Por exemplo, a execução de

```
>>> conta_vogais('testevogais.txt', 'restestavogais.txt')
```

poderá dar origem ao ficheiro:

```
a 41  
e 45
```

i 18
o 26
u 20

2. Escreva uma função em Python que que recebe três cadeias de caracteres, que correspondem a nomes de ficheiros. Os dois primeiros ficheiros, contêm números, ordenados por ordem crescente, contendo cada linha dos ficheiros apenas um número. O seu programa produz um ficheiro ordenado de números (contendo um número por linha) correspondente à junção dos números existentes nos dois ficheiros. Este ficheiro corresponde ao terceiro argumento da sua função. Para cada um dos ficheiros de entrada, o seu programa só pode ler uma linha de cada vez.
3. Escreva um programa em Python para formatar texto. O seu programa deve pedir ao utilizador o nome do ficheiro que contém o texto a ser formatado (este ficheiro, designado por ficheiro fonte, contém o texto propriamente dito e os comandos para o formatador de texto tal como se descrevem a seguir) e o nome do ficheiro onde o texto formatado será guardado (o ficheiro de destino). Após esta interacção deve ser iniciado o processamento do texto, utilizando os seguintes valores de omissão:
 - Cada página tem espaço para 66 linhas;
 - O número de linhas de texto em cada página é de 58;
 - A margem esquerda começa na coluna 9;
 - A margem direita acaba na coluna 79;
 - As linhas são geradas com espaçamento simples (isto é, não são inseridas linhas em branco entre as linhas de texto);
 - As páginas são numeradas automaticamente, sendo o número da página colocado no canto superior direito;
 - Os parágrafos (indicados no ficheiro fonte por uma linha que começa com um ou mais espaços em branco) são iniciados 8 espaços mais para a direita e separados da linha anterior por uma linha em branco;
 - Cada linha do texto começa na margem da esquerda e nunca ultrapassa a margem da direita.

Estes valores de omissão podem ser alterados através de comandos fornecidos ao programa, comandos esses que são inseridos em qualquer ponto

do texto. Cada comando é escrito numa linha individualizada que contém o carácter “\$” na primeira coluna. Os comandos possíveis são:

\$ nl

Este comando faz com que o texto comece numa nova linha, ou seja, origina uma nova linha no ficheiro de destino.

\$ es $\langle n \rangle$

Este comando faz com que o espaçamento entre as linhas do texto passe a ser de $\langle n \rangle$ linhas em branco (o símbolo não terminal $\langle n \rangle$ corresponde a um inteiro positivo).

\$ sp $\langle n \rangle$

Este comando insere $\langle n \rangle$ linhas em branco no texto, tendo em atenção o valor do espaçamento (o símbolo não terminal $\langle n \rangle$ corresponde a um inteiro positivo).

\$ ce $\langle \text{texto} \rangle$

Este comando causa um salto de linha e centra, na linha seguinte, a cadeia de caracteres representada pelo símbolo não terminal $\langle \text{texto} \rangle$.

\$ al

Depois da execução deste comando, todas as linhas produzidas pelo programa estão alinhadas, ou seja, as linhas começam na margem esquerda e acabam na margem direita, embora o número de caracteres possa variar de linha para linha.

Para conseguir este comportamento, o seu programa insere espaços adicionais entre as palavras da linha de modo a que esta termine na margem direita.

\$ na

Depois da execução deste comando, as linhas produzidas pelo programa não têm de estar alinhadas, ou seja, as linhas começam na margem da esquerda e podem acabar antes da margem da direita, desde que a palavra seguinte não caiba na presente linha. Neste caso, não se verifica a inserção de espaços adicionais entre palavras.

\$ me $\langle n \rangle$

Depois da execução deste comando, a margem da esquerda passa a começar na coluna correspondente ao símbolo não terminal $\langle n \rangle$. Se o valor de $\langle n \rangle$ for maior ou igual ao valor da margem da direita, é

originado um erro de execução e este comando é ignorado (o símbolo não terminal $\langle n \rangle$ corresponde a um inteiro positivo).

\$ md $\langle n \rangle$

Depois da execução deste comando, a margem da direita passa a acabar na coluna correspondente ao símbolo não terminal $\langle n \rangle$. Se o valor de $\langle n \rangle$ for menor ou igual ao valor da margem da esquerda, é originado um erro de execução e este comando é ignorado (o símbolo não terminal $\langle n \rangle$ corresponde a um inteiro positivo).

\$ pa

A execução deste comando causa um salto de página, excepto se, quando ele for executado, o formatador de texto se encontrar no início de uma página.

Sugestão: Utilize uma cadeia de caracteres para armazenar a linha que está a ser gerada para o ficheiro de destino. O seu programa deve ler palavras do ficheiro fonte e decidir se estas cabem ou não na linha a ser gerada. Em caso afirmativo, estas são adicionadas à linha a ser gerada, em caso contrário a linha é escrita no ficheiro de destino, eventualmente depois de algum processamento.

Capítulo 9

Dicionários

They were standing under a tree, each with an arm round the other's neck, and Alice knew which was which in a moment, because one of them had 'DUM' embroidered on his collar, and the other 'DEE.' 'I suppose they've each got "TWEEDLE" round at the back of the collar,' she said to herself.

Lewis Carroll, *Through the Looking Glass*

9.1 O tipo dicionário

Um *dicionário*, em Python designado por `dict`¹, em programação normalmente conhecido como *lista associativa* ou *tabela de dispersão*², é um tipo mutável contendo um conjunto de pares. O primeiro elemento de cada par tem o nome de *chave* e o segundo elemento do par corresponde ao *valor* associado com a chave. Num dicionário não podem existir dois pares distintos com a mesma chave. Esta definição de dicionário é semelhante à definição de função apresentada na página 74, contudo, a utilização que faremos dos dicionários corresponde à manipulação directa do conjunto de pares que caracteriza um dado dicionário (recorrendo à *definição por extensão*), em lugar da manipulação da expressão designatória que define esse dicionário.

Notemos também, que ao passo que as duas estruturas de informação que consi-

¹Do inglês “dictionary”.

²Do inglês, “hash table”.

derámos até agora, os tuplos e as listas, correspondem a sequências de elementos, significando isto que a ordem pela qual os elementos surgem na sequência é importante, os dicionários correspondem a conjuntos de elementos, o que significa que a ordem dos elementos num dicionário é irrelevante.

Utilizando a notação BNF, a representação externa de dicionários em Python é definida do seguinte modo:

```

⟨dicionário⟩ ::= {} | {⟨pares⟩}
⟨pares⟩ ::= ⟨par⟩ | ⟨par⟩, ⟨pares⟩
⟨par⟩ ::= ⟨chave⟩ : ⟨valor⟩
⟨chave⟩ ::= ⟨expressão⟩
⟨valor⟩ ::= ⟨expressão⟩ |
            ⟨tuplo⟩ |
            ⟨lista⟩ |
            ⟨dicionário⟩

```

Os elementos de um dicionário são representados dentro de chavetas, podendo corresponder a qualquer número de elementos, incluindo zero. O dicionário {} tem o nome de *dicionário vazio*.

O Python exige que a chave seja um elemento de um tipo imutável, tal como um número, uma cadeia de caracteres ou um tuplo. Esta restrição não pode ser representada em notação BNF. O Python não impõe restrições ao valor associado com uma chave.

São exemplos de dicionários, {'a':3, 'b':2, 'c':4}, {chr(56+1):12} (este dicionário é o mesmo que {'9':12}), {(1, 2):'amarelo', (2, 1):'azul'} e {1:'c', 'b':4}, este último exemplo mostra que as chaves de um dicionário podem ser de tipos diferentes. A entidade {'a':3, 'b':2, 'a':4} não é um dicionário pois tem dois pares com a mesma chave 'a'.³

É importante recordar que os dicionários são *conjuntos*, pelo que quando o Python mostra um dicionário ao utilizador, os seus elementos podem não aparecer pela mesma ordem pela qual foram escritos, como o demonstra a seguinte

³Na realidade, se fornecermos ao Python a entidade {'a':3, 'b':2, 'a':4}, o Python considera-a como o dicionário {'b':2, 'a':4}. Este aspecto está relacionado com a representação interna de dicionários, a qual pode ser consultada em [Cormen et al., 2009].

interacção:

```
>>> {'a':3, 'b':2, 'c':4}
{'a': 3, 'c': 4, 'b': 2}
```

Os elementos de um dicionário são referenciados através do conceito de nome indexado apresentado na página 117. Em oposição aos tuplos e às listas que, sendo sequências, os seus elementos são acedidos por um índice que correspondem à sua posição, os elementos de um dicionário são acedidos utilizando a sua chave como índice. Assim, sendo `d` um dicionário e `c` uma chave do dicionário `d`, o nome indexado `d[c]` corresponde ao valor associado à chave `c`. O acesso e a modificação dos elementos de um dicionário são ilustrados na seguinte interacção:

```
>>> ex_dic = {'a':3, 'b':2, 'c':4}
>>> ex_dic['a']
3
>>> ex_dic['d'] = 1
>>> ex_dic
{'a': 3, 'c': 4, 'b': 2, 'd': 1}
>>> ex_dic['a'] = ex_dic['a'] + 1
>>> ex_dic['a']
4
>>> ex_dic['j']
KeyError: 'j'
```

Na segunda linha, `ex_dic['a']` corresponde ao valor do dicionário associado à chave '`a`'. Na quarta linha, é definido um novo par no dicionário `ex_dic`, cuja chave é '`d`' e cujo valor associado é 1. Na sétima linha, o valor associado à chave '`a`' é aumentado em uma unidade. As últimas duas linhas mostram que se uma chave não existir num dicionário, uma referência a essa chave dá origem a um erro.

Sobre os dicionários podemos utilizar as funções embutidas apresentadas na Tabela 9.1 como se ilustra na interacção seguinte.

```
>>> ex_dic
{'a': 4, 'c': 4, 'b': 2, 'd': 1}
```

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
<code>c in d</code>	Chave e dicionário	True se a chave <i>c</i> pertence ao dicionário <i>d</i> ; False em caso contrário.
<code>c not in d</code>	Chave e dicionário	A negação do resultado da operação <code>c in d</code> .
<code>len(d)</code>	Dicionário	O número de elementos do dicionário <i>d</i> .

Tabela 9.1: Operações sobre dicionários em Python.

```
>>> 'f' in ex_dic
False
>>> 'a' in ex_dic
True
>>> len(ex_dic)
4
```

Analogamente ao que acontece com tuplos e listas, podemos utilizar um ciclo `for` para percorrer todos os elementos de um dicionário, como o mostra a seguinte interacção:

```
>>> inventores = {'Dean Kamen': 'Segway',
... 'Tim Berners-Lee': 'World Wide Web',
... 'Dan Bricklin': 'Spreadsheet'}
...
>>> for i in inventores:
...     print(i, ' inventou:', inventores[i])
...
Dean Kamen  inventou: Segway
Tim Berners-Lee  inventou: World Wide Web
Dan Bricklin  inventou: Spreadsheet
```

9.2 Frequênciа de letras num texto

Suponhamos que `Lusiadas.txt` é um ficheiro de texto, contendo o texto completo de *Os Lusíadas*. O seguinte programa lê a totalidade do ficheiro `Lusiadas.txt`, criando um dicionário em que as chaves são caracteres e cada carácter

está associado ao número de vezes que este aparece no texto de *Os Lusíadas*.

```
lus = open('Lusiadas.txt', 'r', encoding = 'UTF-16')
cont = {}
linha = lus.readline()
while linha != '': # enquanto o fim do ficheiro não for atingido
    for c in linha: # o dicionário é actualizado
        if c not in cont:
            cont[c] = 1
        else:
            cont[c] = cont[c] + 1
    linha = lus.readline()
lus.close()
```

Note-se que neste programa, o dicionário criado inclui uma chave associada ao carácter de salto de linha (\n), ao espaço em branco e a todos os símbolos de pontuação. Após a execução deste programa podemos gerar a interacção:

```
>>> cont['a']
29833
>>> cont['z']
913
>>> cont['\n']
9942
>>> cont[' ']
50449
>>> cont[',']
1618
```

Se desejarmos ignorar os saltos de linha, os espaços em branco e os símbolos de pontuação, podemos criar uma lista chamada pontuacao que contém os caracteres a ignorar⁴:

```
pontuacao = ['!', '"', "''", '(', ')', '‐', ',', '.', '\
';', ':', '?', '[', ']', '{', '}', ',', '\n']
```

⁴Note-se a a utilização de dois delimitadores para cadeias de caracteres, uma contendo as aspas e a outra a plica.

Usando esta lista, podemos alterar o ciclo `for` do pelo programa do seguinte modo:

```
for c in linha:
    if not c in pontuacao:
        if c not in cont :
            cont[c] = 1
        else:
            cont[c] = cont[c] + 1
```

No entanto, o nosso programa ainda conta separadamente as letras minúsculas e as letra maiúsculas que aparecem no texto, como o mostra a seguinte interacção:

```
>>> cont['a']
29833
>>> cont['A']
1222
```

Para resolver este problema podemos pensar em escrever uma função que transforma letras maiúsculas em minúsculas. Esta função recorre ao *Unicode*, calculando qual a letra minúscula correspondente a uma dada maiúscula. Para as letras não acentuadas, esta função é simples de escrever; para símbolos acentuados, esta função obriga-nos a determinar as representações de cada letra maiúscula acentuada, por exemplo “Á” e a sua minúscula correspondente “á”. De modo a que o nosso programa seja completo, teremos também que fazer o mesmo para letras acentuadas noutras línguas que não o português, por exemplo “Ü” e “ü”. Uma outra alternativa, a que vamos recorrer, corresponde a utilizar uma função embutida do Python, a qual para uma dada cadeia de caracteres, representada pela variável `s`, transforma essa cadeia de caracteres numa cadeia equivalente em que todos os caracteres correspondem a letras minúsculas. Esta função, sem argumentos tem o nome `s.lower()`⁵. Por exemplo:

```
>>> ex_cadeia = 'Exemplo DE CONVERSÃO'
>>> ex_cadeia.lower()
'exemplo de conversão'
```

⁵Novamente, surge-nos aqui uma situação semelhante à que apareceu nas funções de leitura e escrita em ficheiros, que o nome da função é um `{nome composto}` em que um dos seus constituintes é o nome de uma variável. Este aspecto é abordado na Secção 11.3.

Podemos agora apresentar a seguinte função que conta o número de letras existentes num ficheiro de texto, cujo nome lhe é fornecido como argumento:

```
def conta_letras(nome):
    fich = open(nome, 'r', encoding = 'UTF-16')
    cont = {}
    pontuacao = ['!', '"', "''", '(', ')', '‐', ',', '.', \
                 ';', ':', '?', '[', ']', '{', '}', ' ', '\n']
    linha = fich.readline()
    while linha != '': # enquanto o fim do ficheiro não for atingido
        for c in linha.lower(): # transformação em minúsculas
            if not c in pontuacao:
                if c not in cont:
                    cont[c] = 1
                else:
                    cont[c] = cont[c] + 1
        linha = fich.readline()
    fich.close()
    return cont
```

Obtendo a interacção:

```
>>> cont = conta_letras('Lusiadas.txt')
>>> cont
{'ê': 378, 'í': 520, 'ã': 1619, 'p': 5554, 'ó': 660, 'ú': 163,
 'ò': 1, 'u': 10617, 'è': 2, 'ô': 52, 'à': 1377, 'ç': 863, 'â': 79,
 'a': 31055, 'à': 158, 'c': 7070, 'b': 2392, 'e': 31581,
 'd': 12304, 'g': 3598, 'f': 3055, 'é': 823, 'h': 2583, 'j': 1023,
 'l': 6098, 'o': 27240, 'n': 13449, 'q': 4110, 'i': 12542,
 's': 20644, 'r': 16827, 'õ': 115, 't': 11929, 'v': 4259, 'y': 7,
 'x': 370, 'z': 919, 'ü': 14, 'm': 10912}
```

Suponhamos que após a execução do programa anterior, executávamos as seguintes instruções:

```
total_letras = 0
for c in cont:
    total_letras = total_letras + cont[c]
```

Com estas instruções calculamos o número total de letras no texto de *Os Lusíadas*, a partir do qual podemos calcular a frequência relativa de cada letra:

```
freq_rel = {}
for c in cont:
    freq_rel[c] = cont[c] / total_letras
```

Após a execução deste programa podemos gerar a interacção:

```
>>> total_letras
246962
>>> freq_rel['a']
0.12574809079939425
>>> freq_rel['z']
0.003721220268705307
```

9.3 Dicionários de dicionários

Suponhamos que desejávamos representar informação relativa à ficha académica de um aluno numa universidade, informação essa que deverá conter o número do aluno, o seu nome e o registo das disciplinas que frequentou, contendo para cada disciplina o ano lectivo em que o aluno esteve inscrito e a classificação obtida.

Comecemos por pensar em como representar a informação sobre as disciplinas frequentadas. Esta informação corresponde a uma coleção de entidades que associam o nome (ou a abreviatura) da disciplina às notas obtidas e o ano lectivo em que a nota foi obtida. Para isso, iremos utilizar um dicionário em que as chaves correspondem às abreviaturas das disciplinas e o seu valor ao registo das notas realizadas. Para um aluno que tenha frequentado as disciplinas de FP, AL, TC e SD, este dicionário terá a forma (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{'FP': <registo das classificações obtidas em FP>,
 'AL': <registo das classificações obtidas em AL>,
 'TC': <registo das classificações obtidas em TC>,
 'SD': <registo das classificações obtidas em SD>}
```

Consideremos agora a representação do registo das classificações obtidas numa dada disciplina. Um aluno pode frequentar uma disciplina mais do que uma vez. No caso de reprovão na disciplina, frequentará a disciplina até obter aprovação (ou prescrever); no caso de obter aprovação, poderá frequentar a disciplina para melhoria de nota no ano lectivo seguinte. Para representar o registo das classificações obtidas numa dada disciplina, iremos utilizar um dicionário em que cada par contém o ano lectivo em que a disciplina foi frequentada (uma cadeia de caracteres) e o valor corresponde à classificação obtida, um inteiro entre 10 e 20 ou REP. Ou seja, o valor associado a cada disciplina é por sua vez um dicionário! O seguinte dicionário pode corresponder às notas do aluno em consideração:

```
{'FP': {'2010/11': 12, '2011/12': 15},
 'AL': {'2010/11': 10},
 'TC': {'2010/11': 12},
 'SD': {'2010/11': 'REP', '2011/12': 13}}
```

Se o dicionário anterior tiver o nome `disc`, então podemos gerar a seguinte interacção:

```
>>> disc['FP']
{'2010/11': 12, '2011/12': 15}
>>> disc['FP']['2010/11']
12
```

O nome de um aluno é constituído por dois componentes, o nome próprio e o apelido. Assim, o nome de um aluno será também representado por um dicionário com duas chaves, `'nome_p'` e `'apelido'`, correspondentes, respectivamente ao nome próprio e ao apelido do aluno. Assim, o nome do aluno António Mega Bites é representado por:

```
{'nome_p': 'António', 'apelido': 'Mega Bites'}
```

Representaremos a informação sobre um aluno como um par pertencente a um dicionário cujas chaves correspondem aos números dos alunos e cujo valor associado é um outro dicionário com uma chave `'nome'` à qual está associada o nome do aluno e com uma chave `'disc'`, à qual está associada um dicionário com as classificações obtidas pelo aluno nos diferentes anos lectivos.

Por exemplo, se o aluno Nº. 12345, com o nome António Mega Bites, obteve as classificações indicadas acima, a sua entrada no dicionário será representada por (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{12345: {'nome': {'nome_p': 'António', 'apelido': 'Mega Bites'},  
         'disc': {'FP': {'2010/11': 12, '2011/12': 15},  
                  'AL': {'2010/11': 10},  
                  'TC': {'2010/11': 12},  
                  'SD': {'2010/11': 'REP', '2011/12': 13}}}}
```

Sendo `alunos` o dicionário com informação sobre todos os alunos da universidade, podemos obter a seguinte interacção:

```
>>> alunos[12345]['nome']  
{'nome_p': 'António', 'apelido': 'Mega Bites'}  
>>> alunos[12345]['nome']['apelido']  
'Mega Bites'  
>>> alunos[12345]['disc']['FP']['2010/11']  
12
```

Vamos agora desenvolver uma função utilizada pela secretaria dos registos académicos ao lançar as notas de uma dada disciplina. O professor responsável da disciplina envia à secretaria por correio electrónico um ficheiro de texto contendo na primeira linha a identificação da disciplina, na segunda linha o ano lectivo ao qual as notas correspondem, seguido de uma série de linhas, cada uma contendo o número de um aluno e a nota respectiva. Por exemplo, o seguinte ficheiro poderá corresponder às notas obtidas em FP no ano lectivo de 2012/13:

```
FP  
2012/13  
12345 12  
12346 REP  
12347 10  
12348 14  
12349 REP  
12350 16  
12351 14
```

A função `introduz_notas` recebe o ficheiro contendo as notas de uma disciplina e o dicionário contendo a informação sobre os alunos e efectua o lançamento das notas de uma determinada disciplina, num dado ano lectivo. O dicionário contendo a informação sobre os alunos é alterado por esta função.

```
def introduz_notas(origem, registos):
    f = open(origem, 'r')
    disc = f.readline()
    disc = disc[0:len(disc) - 1] # remove fim de linha
    ano_lectivo = f.readline()
    ano_lectivo = ano_lectivo[0:len(ano_lectivo) - 1] # idem

    info_aluno = f.readline()
    while info_aluno != '':
        num, nota = obtem_info(info_aluno)
        if num not in registos:
            print('O aluno', num, 'não existe')
        else:
            registos[num]['disc'][disc][ano_lectivo] = nota
        info_aluno = f.readline()
    f.close()
```

A função `obtem_info` recebe uma linha (uma cadeia de caracteres) correspondente a um número de aluno e a respectiva nota e devolve um tuplo contendo o número do aluno como um inteiro e a nota correspondente.

```
def obtem_info(linha):
    pos = 0
    num = ''
    nota = ''
    while linha[pos] != ' ': # obtém número
        num = num + linha[pos]
        pos = pos + 1
    num = eval(num)
    while linha[pos] == ' ': # salta sobre brancos
        pos = pos + 1
    while linha[pos] != '\n': # obtém nota
```

```

nota = nota + linha[pos]
pos = pos + 1
if nota != 'REP':
    nota = eval(nota)
return (num, nota)

```

9.4 Caminhos mais curtos em grafos

Como último exemplo da utilização de dicionários, apresentamos um algoritmo para calcular o caminho mais curto entre os nós de um grafo. Um *grafo* corresponde a uma descrição formal de um grande número de situações do mundo real. Um dos exemplos mais comuns da utilização de grafos corresponde à representação de um mapa com ligações entre cidades. Podemos considerar um mapa como sendo constituído por um conjunto de *nós*, os quais correspondem às cidades do mapa, e por um conjunto de *arcos*, os quais correspondem às ligações entre as cidades. Formalmente, um grafo corresponde a uma estrutura (N, A) , na qual N é um conjunto de entidades a que se dá o nome de *nós* e A é um conjunto de ligações, os *arcos*, entre os nós do grafo. Os arcos de um grafo podem ser *dirididos*, correspondentes, na analogia com os mapas, a estradas apenas com um sentido, e podem também ser *rotulados*, contendo, por exemplo, a distância entre duas cidades seguindo uma dada estrada ou o valor da portagem a pagar se a estrada correspondente for seguida.

Na Figura 9.1, apresentamos um grafo com seis nós, com os identificadores A, B, C, D, E e F. Neste grafo existe, por exemplo, um arco de A para B, com a distância 10, um arco de C para D, com a distância 15 e um arco de D para C, também com a distância 15. Neste grafo, por exemplo, não existe um arco do nó A para o nó F. Dado um grafo e dois nós desse grafo, n_1 e n_2 , define-se um *caminho* de n_1 e n_2 como a sequência de arcos que é necessário atravessar para ir do nó n_1 para o nó n_2 . Por exemplo, no grafo da Figura 9.1, um caminho de A para D corresponde ao arcos que ligam os nós A a B, B a C e C a D.

Uma aplicação comum em grafos corresponde a determinar a distância mais curta entre dois nós, ou seja, qual o caminho entre os dois nós em que o somatório das distâncias percorridas é menor. Um algoritmo clássico para determinar a distância mais curta entre dois nós de um grafo foi desenvolvido por Edsger

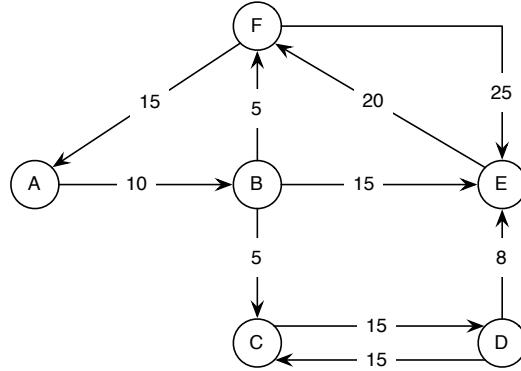


Figura 9.1: Exemplo de um grafo.

Dijkstra⁶ (1930–2002).

O algoritmo de Dijkstra recebe como argumentos um grafo e um nó desse grafo, a que se chama o *nó inicial*, e calcula a distância mínima do nó inicial a cada nó do grafo. O algoritmo começa por associar cada nó com a distância conhecida ao nó inicial. A esta associação chamamos a *distância calculada*. No início do algoritmo, apenas sabemos que a distância do nó inicial a si próprio é zero, podendo a distância do nó inicial a qualquer outro nó ser infinita (se não existirem ligações entre eles). Por exemplo, em relação ao grafo apresentado na Figura 9.1, sendo A o nó inicial, representaremos as distâncias calculadas iniciais por:

```
{'A':0, 'B':'inf', 'C':'inf', 'D':'inf', 'E':'inf', 'F':'inf'}
```

em que '*inf*' representa infinito. Na Figura 9.2 (a), apresentamos dentro de parênteses, junto a cada nó, a distância calculada pelo algoritmo. No início do algoritmo cria-se também uma lista contendo todos os nós do grafo.

O algoritmo percorre repetitivamente a lista dos nós, enquanto esta não for vazia, executando as seguintes ações:

1. Escolhe-se da lista de nós o nó com menor valor da distância calculada.
Se existir mais do que um nó nesta situação, escolhe-se arbitrariamente um deles;
2. Remove-se o nó escolhido da lista de nós;

⁶[Dijkstra, 1959].

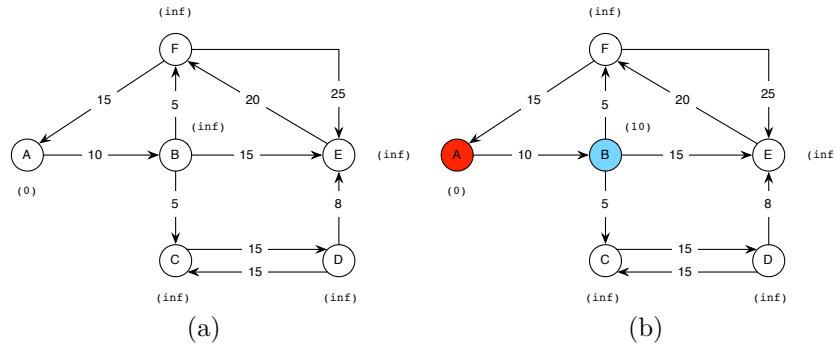


Figura 9.2: Dois primeiros passos no algoritmo de Dijkstra.

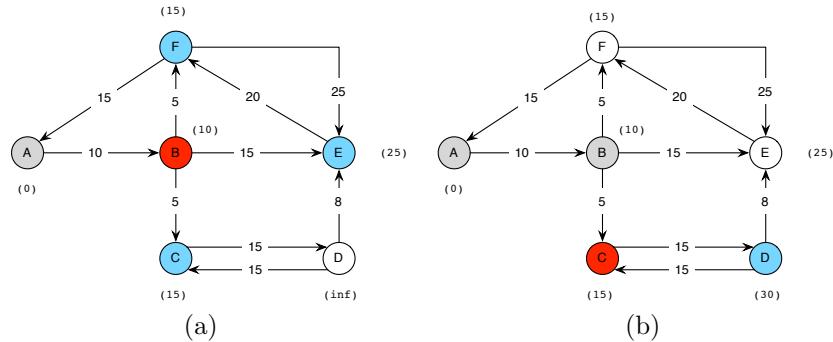


Figura 9.3: Terceiro e quarto passos no algoritmo de Dijkstra.

3. Actualiza-se a distância calculada dos nós adjacentes ao nó escolhido (os nós a que o nó escolhido está directamente ligado). Esta distância só é actualizada se a nova distância calculada for inferior à distância anteriormente calculada para o nó em causa.

Quando o algoritmo termina, todos os nós estão associados com a distância mínima ao nó original.

Na Figura 9.2 (b), mostramos o passo correspondente a seleccionar o primeiro nó, o nó A, actualizando-se a distância calculada para o nó B. Note-se que A não está ligado a F, mas F está ligado a A. Selecciona-se de seguida o nó B, actualizando-se as distâncias calculadas dos nós C, E e F (Figura 9.3 (a)). Existem agora dois nós com a distância calculada de 15, selecciona-se arbitrariamente o nó C e actualiza-se a distância calculada associada ao nó D (Fi-

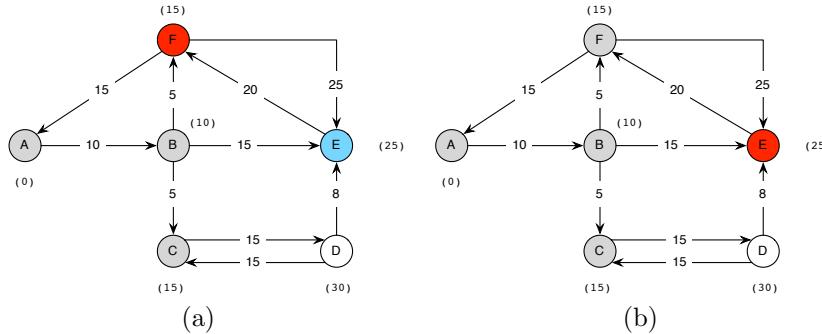


Figura 9.4: Quinto e sexto passos no algoritmo de Dijkstra.

gura 9.3 (b)). Escolhe-se de seguida o nó F , o qual não leva à actualização de nenhuma distância calculada (Figura 9.4 (a)). O nó E é então escolhido, não levando também à actualização das distâncias calculadas (Figura 9.4 (b)). Finalmente, o nó D é escolhido, não originando a actualização de nenhuma distância calculada.

De modo a poder escrever este algoritmo em Python, teremos que escolher uma representação para grafos. Iremos escolher uma representação que é conhecida por *lista de adjacências*⁷, na qual um grafo é representado por um dicionário em que as chaves correspondem aos nós e cada nó está associado a uma lista de tuplos, contendo os nós ligados ao nó em questão e as respectivas distâncias. No nosso exemplo, o nó B está ligado aos nós C , E e F , respectivamente, com as distâncias 5, 15 e 5, pelo que a sua representação será dada pelo par $'B': [('C', 5), ('E', 15), ('F', 5)]$. A representação do grafo da Figura 9.1, que designaremos por $g1$, é:

```
g1 = {'A': [('B', 10)], 'B': [('C', 5), ('E', 15), ('F', 5)],
      'C': [('D', 15)], 'D': [('C', 15), ('E', 8)], 'E': [('F', 20)],
      'F': [('A', 15), ('E', 25)]}
```

Assim, a função que implementa o algoritmo da distância mínima pode ser escrita em Python do seguinte modo:

```
def cdm(g, origem):
    nos = nos_do_grafo(g)
```

⁷Do inglês, “adjacency list”.

```

dc = dist_inicial(nos, origem)
while nos != []:
    n = dist_min(nos, dc) # selecciona o nó com menor distância
    remove(nos, n)
    for nn in g[n]: # para os nós a que nn está ligado
        actualiza_dist(n, nn[0], nn[1], dc)
return dc

```

A função `nos_do_grafo` devolve a lista dos nós do grafo que é seu argumento. Note-se a utilização de um ciclo `for` que, quando aplicado a um dicionário itera sobre as chaves do dicionário, tal como descrito na página 246.

```

def nos_do_grafo(g):
    res = []
    for el in g:
        res = res + [el]
    return res

```

A função `dist_inicial` recebe como argumentos a lista dos nós do grafo e o nó inicial e devolve o dicionário que corresponde à distância inicial dos nós do grafo à origem, ou seja, zero para a origem e '`inf`' para os restantes nós. Para o grafo do nosso exemplo, esta função devolve o dicionário apresentado na página 255.

```

def dist_inicial(nos, origem):
    res = {}
    for n in nos:
        if n == origem:
            res[n] = 0
        else:
            res[n] = 'inf'
    return res

```

A função `dist_min` recebe a lista dos nós em consideração (`nos`) e o dicionário com as distâncias calculadas (`dc`) e devolve o nó com menor distância e a função `remove` remove o nó que é seu argumento da lista dos nós. Note-se que esta função altera a lista `nos`.

```
def dist_min(nos, dc):
```

```

n_min = nos[0]      # o primeiro nó
min = dc[n_min]    # a distância associada ao primeiro nó

for n in nos:
    if menor(dc[n], min):
        min = dc[n]  # a distância associada ao nó n
        n_min = n
return n_min

def remove(nos, n):
    for nn in range(len(nos)):
        if nos[nn] == n:
            del nos[nn]
    return

```

A função `actualiza_dist` recebe um nó de partida (`no_partida`), um nó a que o `no_partida` está directamente ligado (`no_chegada`), a distância entre o `no_partida` e o `no_chegada` e as distâncias calculadas (`dc`) e actualiza, nas distâncias calculadas, a distância associada ao `no_chegada`.

```

def actualiza_dist(no_partida, no_chegada, dist, dc):
    dist_no_partida = dc[no_partida]
    if menor(soma(dist_no_partida, dist), dc[no_chegada]):
        dc[no_chegada] = dist_no_partida + dist

```

Finalmente, as funções `menor` e `soma`, respectivamente comparam e somam dois valores, tendo em atenção que um deles pode ser '`inf`'.

```

def menor(v1, v2):
    if v1 == 'inf':
        return False
    elif v2 == 'inf':
        return True
    else:
        return v1 < v2

```

```
def soma(v1, v2):
    if v1 == 'inf' or v2 == 'inf':
        return 'inf'
    else:
        return v1 + v2
```

Usando a função `cdm` e o grafo apresentado na Figura 9.1, podemos originar a seguinte interacção:

```
>>> cdm(g1, 'A')
{'A': 0, 'C': 15, 'B': 10, 'E': 25, 'D': 30, 'F': 15}
>>> cdm(g1, 'B')
{'A': 20, 'C': 5, 'B': 0, 'E': 15, 'D': 20, 'F': 5}
```

9.5 Notas finais

Neste capítulo apresentámos o tipo dicionário, o qual corresponde ao conceito de lista associativa ou tabela de dispersão. Um dicionário associa chaves a valores, sendo os elementos de um dicionário indexados pela respectiva chave. Um dos aspectos importantes na utilização de dicionários corresponde a garantir que o acesso aos elementos de um dicionário é feito a tempo constante. Para estudar o modo como os dicionários são representados internamente no computador aconselhamos a consulta de [Cormen et al., 2009].

Apresentámos, como aplicação de utilização de dicionários, o algoritmo do caminho mais curto em grafos. O tipo grafo é muito importante em informática, estando para além da matéria deste livro. A especificação do tipo grafo e algumas das suas aplicações podem ser consultadas em [Dale e Walker, 1996]. Algoritmos que manipulam grafos podem ser consultados em [McConnell, 2008] e em [Cormen et al., 2009].

9.6 Exercícios

1. Considere a seguinte variável:

```

teste = {'Portugal':{'Lisboa':[('Leonor', '1700-097'),
                               ('João', '1050-100')],
                  'Porto':[('Ana', '4150-036')]},
         'Estados Unidos':{'Miami':[('Nancy', '33136'),
                                   ('Fred', '33136')],
                           'Chicago':[('Cesar', '60661')]},
         'Reino Unido':{'London':[('Stuart', 'SW1H OBD')]}}

```

Qual o valor de cada um dos seguintes nomes? Se algum dos nomes originar um erro, explique a razão do erro.

- (a) `teste['Portugal']['Porto']`
 - (b) `teste['Portugal']['Porto'][0][0]`
 - (c) `teste['Estados Unidos']['Miami'][1]`
 - (d) `teste['Estados Unidos']['Miami'][1][0][0]`
 - (e) `teste['Estados Unidos']['Miami'][1][1][1]`
2. Escreva uma função em Python que que recebe uma cadeia de caracteres, que contém o nome de um ficheiro, lê esse ficheiro, linha a linha, e calcula quantas vezes aparece cada uma das vogais. Recorrendo a dicionários, a sua função deve devolver um dicionário cujas chaves são as vogais e os valores associados correspondem ao número de vezes que a vogal aparece no ficheiro. Apenas conte as vogais que são letras minúsculas. Por exemplo,

```

>>> conta_vogais('testevogais.txt')
{'a': 36, 'u': 19, 'e': 45, 'i': 16, 'o': 28}

```

3. Escreva uma função em Python que que recebe duas cadeias de caracteres, que correspondem a nomes de ficheiros. A sua função, lê o primeiro ficheiro, linha a linha, e calcula quantas vezes aparece cada uma das vogais. Após a leitura, a sua função escreve no segundo ficheiro, uma linha por vogal, indicando a vogal e o número de vezes que esta apareceu. Apenas conte as vogais que são letras minúsculas. Por exemplo, a execução de

```

>>> conta_vogais('testevogais.txt', 'restestavogais.txt')

```

poderá dar origem ao ficheiro:

e 45

i 18

o 26

u 20

4. Adicione ao processador de texto apresentado nos exercícios do Capítulo 8 a capacidade de produzir um índice alfabético. Para isso, algumas das palavras do seu texto devem ser marcadas com um símbolo especial, por exemplo, o símbolo “@”, o qual indica que a palavra que o segue deve ser colocada no índice alfabético, que contém, para cada palavra, a página em que esta aparece. Por exemplo, se no seu texto aparece @programação, então a palavra “programação” deve aparecer no índice alfabético juntamente com a indicação da página em que aparece. As palavras do índice alfabético aparecem ordenadas alfabeticamente. Tenha em atenção os sinais de pontuação que podem estar juntos com as palavras mas não devem aparecer no índice.

Sugestão: utilize um dicionário em que cada chave contém uma palavra a inserir no índice alfabético, associado à página, ou páginas, em que esta aparece. Mantenha este dicionário ordenado.

5. Escreva um programa que aceite como dado de entrada um ficheiro fonte do seu formatador de texto, e conte quantas palavras e quantos parágrafos o texto contém. Tenha cuidado para não contar os comandos do formataador de texto como palavras.

Capítulo 10

Abstracção de dados

'Not very nice alone' he interrupted, quite eagerly: 'but you've no idea what a difference it makes, mixing it with other things'

Lewis Carroll, *Through the Looking Glass*

No Capítulo 3 introduzimos o conceito de abstracção procedural e desde então temos utilizado a abstracção procedural na definição de novas funções. Já vimos que quase qualquer programa que efectue uma tarefa que não seja trivial, necessita de manipular entidades computacionais (dados) que correspondam a tipos estruturados de informação. Os programas que temos desenvolvido manipulam directamente instâncias dos tipos existentes em Python, sem criarem novos tipos de informação. Mesmo em situações em que faria sentido a criação de novos tipos de informação, por exemplo, no cálculo do caminho mais curto em grafos apresentado na Secção 9.4, recorremos a uma representação de grafos e não definimos o tipo grafo, como seria natural.

Um programa complexo requer normalmente a utilização de tipos de informação que não existem na linguagem de programação utilizada. Analogamente ao que referimos no início do Capítulo 3, não é possível que uma linguagem de programação forneça todos os tipos de dados que são necessários para o desenvolvimento de uma certa aplicação. Torna-se assim importante fornecer ao programador a capacidade de definir novos tipos de dados dentro de um programa e de utilizar esses tipos de dados como se fossem tipos embutidos da

linguagem de programação.

Neste capítulo, discutimos como criar tipos estruturados de informação que não estejam embutidos na nossa linguagem de programação, introduzindo o conceito de abstracção de dados.

10.1 A abstracção em programação

Sabemos que uma *abstracção* é uma descrição ou uma especificação simplificada de uma entidade que dá ênfase a certas propriedades dessa entidade e ignora outras. Uma boa abstracção específica as propriedades importantes e ignora os pormenores. Uma vez que as propriedades relevantes de uma entidade dependem da utilização a fazer com essa entidade, o termo “boa abstracção” está sempre associado a uma utilização particular da entidade.

A abstracção é um conceito essencial em programação. De facto, a actividade de programação pode ser considerada como a construção de abstracções que podem ser executadas por um computador.

Até aqui temos usado a abstracção procedural para a criação de funções. Usando a abstracção procedural, os pormenores da realização de uma função podem ser ignorados, fazendo com que uma função possa ser substituída por uma outra função com o mesmo comportamento global, utilizando um outro algoritmo, sem que os programas que utilizam essa função sejam afectados. A abstracção procedural permite pois separar o modo como uma função é utilizada do modo como essa função é realizada.

O conceito equivalente à abstracção procedural para dados (ou estruturas de informação) tem o nome de abstracção de dados. A *abstracção de dados* é uma metodologia que permite separar o modo como uma estrutura de informação é utilizada dos pormenores relacionados com o modo como essa estrutura de informação é construída a partir de outras estruturas de informação. Quando utilizamos um tipo de informação embutido em Python, por exemplo uma lista, não sabemos (nem queremos saber) qual o modo como o Python realiza internamente as listas. Para isso, recorremos a um conjunto de operações embutidas (apresentadas na Tabela 5.1) para escrever as nossas funções. Se numa versão posterior do Python, a representação interna das listas for alterada, os nossos programas não são afectados por essa alteração. A abstracção de dados permite

a obtenção de um comportamento semelhante para os dados criados no nosso programa. Analogamente ao que acontece quando recorremos à abstracção procedimental, com a abstracção de dados, podemos substituir uma realização particular da entidade correspondente a um dado sem ter de alterar o programa que utiliza essa entidade, desde que a nova realização da entidade apresente o mesmo comportamento genérico, ou seja, desde que a nova realização corresponda, na realidade, à mesma entidade abstracta.

10.2 Motivação: números complexos

A teoria dos tipos abstractos de informação considera que a definição de novos tipos de informação é feita em duas fases sequenciais, a primeira corresponde ao estudo das propriedades do tipo, e a segunda aborda os pormenores da realização do tipo numa linguagem de programação. A essência da abstracção de dados corresponde à separação das partes do programa que lidam com o modo como as entidades do tipo são *utilizadas* das partes que lidam com o modo como as entidades são *representadas*.

Para facilitar a nossa apresentação, vamos considerar um exemplo que servirá de suporte à nossa discussão. Suponhamos que desejávamos escrever um programa que lidava com números complexos. Os números complexos surgem em muitas formulações de problemas concretos e são entidades frequentemente usadas em Engenharia.

Os números complexos foram introduzidos, no século XVI, pelo matemático italiano Rafael Bombelli (1526–1572). Estes números surgiram da necessidade de calcular raízes quadradas de números negativos e são obtidos com a introdução do símbolo i , a *unidade imaginária*, que satisfaz a equação $i^2 = -1$. Um número complexo é um número da forma $a + bi$, em que tanto a , a *parte real*, como b , a *parte imaginária*, são números reais; um número complexo em que a parte real é nula chama-se um número *imaginário puro*. Sendo um número complexo constituído por dois números reais (à parte da unidade imaginária), pode-se estabelecer uma correspondência entre números complexos e pares de números reais, a qual está subjacente à representação de números complexos como pontos de um plano¹. Sabemos também que a soma, subtração, multiplicação e

¹Chamado *plano de Argand*, em honra ao matemático francês Jean-Robert Argand (1768–1822) que o introduziu em 1806.

divisão de números complexos são definidas do seguinte modo:

$$(a + b i) + (c + d i) = (a + c) + (b + d) i$$

$$(a + b i) - (c + d i) = (a - c) + (b - d) i$$

$$(a + b i) \cdot (c + d i) = (a c - b d) + (a d + b c) i$$

$$\frac{(a + b i)}{(c + d i)} = \frac{a c + b d}{c^2 + d^2} + \frac{b c - a d}{c^2 + d^2} i$$

operações que são óbvias, tendo em atenção que $i^2 = -1$.

Suponhamos que desejávamos escrever um programa que iria lidar com números complexos. Este programa deverá ser capaz de adicionar, subtrair, multiplicar e dividir números complexos, deve saber comparar números complexos, etc.

A hipótese de trabalharmos com números complexos com a parte real e a parte imaginária como entidades separadas é impensável pela complexidade que irá introduzir no nosso programa e no nosso raciocínio, pelo que esta hipótese será imediatamente posta de lado.

A primeira tentação de um programador inexperiente será a de começar por pensar em como representar números complexos com base nas estruturas de informação que conhece. Por exemplo, esse programador poderá decidir que um número complexo é representado por um tuplo, em que o primeiro elemento contém a parte real e o segundo elemento contém a parte imaginária. Assim o complexo $a + b i$ será representado pelo tuplo (a, b) . Esse programador, tendo duas variáveis que correspondem a números complexos, $c1$ e $c2$, irá naturalmente escrever a seguinte expressão para somar esses números, dando origem ao número complexo $c3$:

$$c3 = (c1[0] + c2[0], c1[1] + c2[1]).$$

Embora a expressão anterior esteja correcta, ela introduz uma complexidade desnecessária no nosso raciocínio. Em qualquer operação que efectuemos sobre números complexos temos que pensar simultaneamente na operação matemática e na representação que foi escolhida para complexos.

Suponhamos, como abordagem alternativa, e utilizando uma estratégia de pensamento positivo, que existem em Python as seguintes funções:

`cria_compl(r, i)` esta função recebe como argumentos dois números reais, *r* e *i*, e tem como valor o número complexo cuja parte real é *r* e cuja parte imaginária é *i*.

`p_real(c)` esta função recebe como argumento um número complexo, *c*, e tem como valor a parte real desse número.

`p_imag(c)` esta função recebe como argumento um número complexo, *c*, e tem como valor a parte imaginária desse número.

Com base nesta suposição, podemos escrever funções que efectuam operações aritméticas sobre números complexos, embora não saibamos como estes são representados. Assim, podemos escrever as seguintes funções que, respectivamente, adicionam, subtraem, multiplicam e dividem números complexos:

```
def soma_compl(c1, c2):
    p_r = p_real(c1) + p_real(c2)
    p_i = p_imag(c1) + p_imag(c2)
    return cria_compl(p_r, p_i)

def subtrai_compl(c1, c2):
    p_r = p_real(c1) - p_real(c2)
    p_i = p_imag(c1) - p_imag(c2)
    return cria_compl(p_r, p_i)

def multiplica_compl(c1, c2):
    p_r = p_real(c1) * p_real(c2) - p_imag(c1) * p_imag(c2)
    p_i = p_real(c1) * p_imag(c2) + p_imag(c1) * p_real(c2)
    return cria_compl(p_r, p_i)

def divide_compl(c1, c2):
    den = p_real(c2) * p_real(c2) + p_imag(c2) * p_imag(c2)
    p_r = (p_real(c1) * p_real(c2) + p_imag(c1) * p_imag(c2))/den
    p_i = (p_imag(c1) * p_real(c2) - p_real(c1) * p_imag(c2))/den
    return cria_compl(p_r, p_i)
```

Independentemente da forma como será feita a representação de números complexos, temos também interesse em dotar o nosso programa com a possibilidade de mostrar um número complexo tal como habitualmente o escrevemos:

o número complexo com parte real a e parte imaginária b é apresentado ao mundo exterior² como $a + bi$. Note-se que esta representação de complexos é exclusivamente “para os nossos olhos”, no sentido de que ela nos permite visualizar números complexos mas não interfere no modo como o programa lida com números complexos. A este tipo de representação chama-se *representação externa*, em oposição à *representação interna*, a qual é utilizada pelo Python para lidar com números complexos (e que é desconhecida neste momento).

Podemos escrever a seguinte função para produzir uma representação externa para números complexos:

```
def escreve_compl(c):
    if p_imag(c) >= 0:
        print(p_real(c), '+', p_imag(c), 'i')
    else:
        print(p_real(c), '-', abs(p_imag(c)), 'i')
```

De modo a que as funções anteriores possam ser utilizadas em Python é necessário que concretizemos as operações `cria_compl`, `p_real` e `p_imag`. Para isso, teremos de encontrar um modo de agregar a parte real com a parte imaginária de um complexo numa única entidade.

Suponhamos que decidimos representar um número complexo por um tuplo, em que o primeiro elemento contém a parte real e o segundo elemento contém a parte imaginária. Com base nesta representação, podemos escrever as funções `cria_compl`, `p_real` e `p_imag` do seguinte modo:

```
def cria_compl(r, i):
    return (r, i)

def p_real(c):
    return c[0]

def p_imag(c):
    return c[1]
```

A partir deste momento podemos utilizar números complexos, como o demonstra a seguinte interacção:

²Por “mundo exterior” entenda-se o mundo com que o Python comunica.

```
>>> c1 = cria_compl(3, 5)
>>> p_real(c1)
3
>>> p_imag(c1)
5
>>> escreve_compl(c1)
3 + 5 i
>>> c2 = cria_compl(1, -3)
>>> escreve_compl(c2)
1 - 3 i
>>> c3 = soma_compl(c1, c2)
>>> escreve_compl(c3)
4 + 2 i
>>> escreve_compl(subtrai_compl(cria_compl(2, -8), c3))
-2 - 10 i
```

Podemos agora imaginar que alguém objecta à nossa representação de números complexos como tuplos, argumentando que esta não é a mais adequada pois os índices 0 e 1, usados para aceder aos constituintes de um complexo, nada dizem sobre o constituinte acedido. Segundo essa argumentação, seria mais natural utilizar um dicionário para representar complexos, usando-se o índice '*r*' para a parte real e o índice '*i*' para a parte imaginária. Motivados por esta argumentação, podemos alterar a nossa representação, de modo a que um complexo seja representado por um dicionário. Podemos dizer que o número complexo $a + bi$ é representado pelo dicionário com uma chave '*r*', cujo valor é a e uma chave '*i*', cujo valor é b . Utilizando a notação $\Re[X]$ para indicar a representação interna da entidade X , podemos escrever, $\Re[a + bi] = \{ 'r': a, 'i': b \}$.

Com base na representação utilizando dicionários, podemos escrever as seguintes funções para realizar as operações que criam números complexos e seleccionam os seus componentes:

```
def cria_compl(r, i):
    return {'r':r, 'i':i}

def p_real(c):
    return c['r']
```

```
def p_imag(c):
    return c['i']
```

Utilizando esta representação, e recorrendo às operações definidas nas páginas 267 e seguintes, podemos replicar a interacção obtida com a representação de complexos através de tuplos (apresentada na página 268) sem qualquer alteração. Este comportamento revela a independência entre as funções que efectuam operações aritméticas sobre números complexos e a representação interna de números complexos. Este comportamento foi obtido através de uma separação clara entre as operações que manipulam números complexos e as operações que correspondem à definição de complexos (`cria_compl`, `p_real` e `p_imag`). Esta separação permite-nos alterar a representação de complexos sem ter de alterar o programa que lida com números complexos.

Esta é a essência da *abstracção de dados*, a separação entre as propriedades dos dados e os pormenores da realização dos dados numa linguagem de programação. Esta essência é traduzida pela separação das partes do programa que lidam com o modo como os dados são *utilizados* das partes que lidam com o modo como os dados são *representados*.

10.3 Tipos abstractos de informação

Nesta secção apresentamos os passos a seguir no processo de criação de novos tipos de informação. Sabemos que um tipo de informação é uma colecção de entidades, os elementos do tipo, conjuntamente com uma colecção de operações que podem ser efectuadas sobre essas entidades.

A metodologia que apresentamos tem o nome de metodologia dos *tipos abstractos de informação*³, e a sua essência é a separação das partes do programa que lidam com o modo como as entidades do tipo são utilizadas das partes que lidam com o modo como as entidades são representadas.

Na utilização da metodologia dos tipos abstractos de informação devem ser seguidos quatro passos sequenciais: (1) a identificação das operações básicas; (2) a axiomatização das operações básicas; (3) a escolha de uma representação

³Em inglês “abstract data types” ou ADTs.

para os elementos do tipo; e (4) a concretização das operações básicas para a representação escolhida.

Esta metodologia permite a definição de tipos de informação que são independentes da sua representação. Esta independência leva à designação destes tipos por tipos *abstractos* de informação. Exemplificamos estes passos para a definição do tipo complexo.

10.3.1 Identificação das operações básicas

Para definir a parte do programa que lida com o modo como os dados são *utilizados*, devemos identificar, para cada tipo de informação, o conjunto das operações básicas que podem ser efectuadas sobre os elementos desse tipo. É evidente que ao definir um tipo de informação não podemos definir todas as operações que manipulam os elementos do tipo. A ideia subjacente à criação de um novo tipo é a definição do *mínimo* possível de operações que permitam caracterizar o tipo. Estas operações são chamadas as *operações básicas* e dividem-se em seis grupos, os construtores, os selectores, os modificadores, os transformadores, os reconhecedores e os testes.

1. Os *construtores* são operações que permitem construir novos elementos do tipo. Em Python, os construtores para os tipos embutidos na linguagem correspondem à escrita da representação externa do tipo, por exemplo, `5` origina o inteiro `5` e `[2, 3, 5]` origina (constrói) uma lista com três elementos. Para o tipo complexo existe apenas um construtor, a operação `cria_compl`, a qual cria complexos a partir dos seus constituintes.
2. Os *selectores* são operações que permitem aceder (isto é, seleccionar) aos constituintes dos elementos do tipo. Em relação aos tipos embutidos em Python, os selectores são definidos através de uma notação específica para o tipo, por exemplo, um índice ou uma gama de índices no caso das listas. No caso das listas, a operação `len`, apresentada na Tabela 5.1, também corresponde a um selector pois selecciona o número de elementos de uma lista. Em relação ao tipo complexo, podemos seleccionar cada um dos seus componentes, existindo assim dois selectores, `p.real` e `p.imag`.
3. Os *modificadores* alteram destrutivamente os elementos do tipo. No caso das listas, a operação `del`, apresentada na Tabela 5.1, corresponde a um

modificador. No caso dos números complexos, dado que cada complexo é considerado como uma constante, não definimos modificadores.

4. Os *transformadores* transformam os elementos de um tipo em elementos de outro tipo. Em relação aos tipos embutidos em Python, as operações `round` e `int`, apresentadas na Tabela 2.4, correspondem a transformadores de reais para inteiros. Não definimos transformadores para números complexos.
5. Os *reconhecedores* são operações que identificam elementos do tipo. Os reconhecedores são de duas categorias. Por um lado, fazem a distinção entre os elementos do tipo e os elementos de qualquer outro tipo, reconhecendo explicitamente os elementos que pertencem ao tipo. Por outro lado, identificam elementos do tipo que se individualizam dos restantes por possuírem certas propriedades particulares. Para os tipos embutidos em Python, a operação `isinstance`, apresentada na página 121 corresponde a um reconhecedor. Em relação ao tipo complexo podemos identificar o reconhecedor, `complexo`, que reconhece números complexos, e os reconhecedores `compl_zero` e `imag_puro`⁴ que reconhecem números complexos com certas características.
6. Os *testes* são operações que efectuam comparações entre os elementos do tipo. A operação `==` permite reconhecer a igualdade entre os elementos dos tipos embutidos em Python. Em relação aos números complexos podemos identificar a operação `compl_iguais`⁵ que decide se dois números complexos são iguais.

O papel das operações básicas é construírem elementos do tipo (os construtores), seleccionarem componentes dos elementos do tipo (os selectores), alterarem os elementos do tipo (os modificadores), transformarem elementos do tipo em outros tipos (os transformadores) e responderem a perguntas sobre os elementos do tipo (os reconhecedores e os testes).

Nem todos estes grupos de operações têm que existir na definição de um tipo de informação, servindo a metodologia dos tipos abstractos de informação para nos guiar na decisão sobre quais os tipos de operações a definir. Uma decisão essencial a tomar, e que vai determinar a classificação das operações, consiste em

⁴Nenhuma destas operações foi ainda por nós definida nem realizada.

⁵Esta operação ainda não foi por nós definida nem realizada.

determinar se o tipo é uma entidade mutável ou imutável. Num tipo *mutável*, de que são exemplo as listas, podemos alterar permanentemente os seus elementos; por outro lado, num tipo *imutável*, de que são exemplo os tuplos, não é possível alterar os elementos do tipo. Existem tipos que são inherentemente mutáveis, por exemplo as contas bancárias apresentadas no Capítulo 11, e outros que são inherentemente imutáveis, por exemplo os números complexos por corresponderem a constantes. Para certos tipos, por exemplo as pilhas apresentadas no Capítulo 13 e as árvores apresentadas no Capítulo 14, é possível decidir se estes devem ser mutáveis ou imutáveis. Neste caso, algumas das operações básicas poderão ser classificadas como selectores ou como modificadores, consoante o tipo for considerado uma entidade imutável ou uma entidade mutável.

Ao conjunto das operações básicas para um dado tipo dá-se o nome de *assinatura do tipo*.

De modo a enfatizar que as operações básicas são independentes da linguagem de programação utilizada (na realidade, estas especificam de um modo abstracto o que é o tipo), é utilizada a notação matemática para as caracterizar.

Para o tipo complexo definimos as seguintes operações básicas:

1. *Construtores*:

- $\text{cria_compl} : \text{real} \times \text{real} \mapsto \text{complexo}$
 $\text{cria_compl}(r, i)$ tem como valor o número complexo cuja parte real é r e cuja parte imaginária é i .

2. *Selectores*:

- $p\text{-real} : \text{complexo} \mapsto \text{real}$
 $p\text{-real}(c)$ tem como valor a parte real do complexo c .
- $p\text{-imag} : \text{complexo} \mapsto \text{real}$
 $p\text{-imag}(c)$ tem como valor a parte imaginária do complexo c .

3. *Reconhecedores*:

- $\text{complexo} : \text{universal} \mapsto \text{lógico}$
 complexo(arg) tem o valor *verdadeiro* se arg é um número complexo e tem o valor *falso* em caso contrário.

- $compl_zero : complexo \mapsto lógico$
 $compl_zero(c)$ tem o valor *verdadeiro* se c é o complexo $0 + 0i$ e tem o valor *falso* em caso contrário.
- $imag_puro : complexo \mapsto lógico$
 $imag_puro(c)$ tem o valor *verdadeiro* se c é um imaginário puro, ou seja, um complexo da forma $0 + bi$, e tem o valor *falso* em caso contrário.

4. *Testes:*

- $compl_iguais : complexo \times complexo \mapsto lógico$
 $compl_iguais(c_1, c_2)$ tem o valor *verdadeiro* se c_1 e c_2 correspondem ao mesmo número complexo e tem o valor *falso* em caso contrário.

Assim, a assinatura do tipo complexo é:

$$\begin{aligned}
cria_compl &: real \times real \mapsto complexo \\
p_real &: complexo \mapsto real \\
p_imag &: complexo \mapsto real \\
complexo &: universal \mapsto lógico \\
compl_zero &: complexo \mapsto lógico \\
imag_puro &: complexo \mapsto lógico \\
compl_iguais &: complexo \times complexo \mapsto lógico
\end{aligned}$$

Deveremos ainda definir uma *representação externa* para complexos, por exemplo, podemos definir que a notação correspondente aos elementos do tipo complexo é da forma $a + bi$, em que a e b são reais. Com base nesta representação, devemos especificar os transformadores de entrada e de saída.

O *transformador de entrada* transforma a representação externa para as entidades abstractas na sua representação interna (seja ela qual for) e o *transformador de saída* transforma a representação interna das entidades na sua representação externa. Neste capítulo, não especificaremos o transformador de entrada para os tipos que definimos. O transformador de saída para números complexos,

ao qual chamamos `escreve_compl`, recebe uma entidade do tipo complexo e escreve essa entidade sob a forma $a + bi$.

Ao especificarmos as operações básicas e os transformadores de entrada e de saída para um dado tipo, estamos a criar uma extensão conceptual da nossa linguagem de programação como se o tipo fosse um tipo embutido. Podemos então escrever programas que manipulam entidades do novo tipo, em termos dos construtores, selectores, modificadores, transformadores, reconhecedores e testes, mesmo antes de termos escolhido uma representação para o tipo e de termos escrito funções que correspondam às operações básicas. Deste modo, obtemos uma verdadeira separação entre a utilização do tipo de informação e a sua realização através de um programa.

10.3.2 Axiomatização

A axiomatização especifica o modo como as operações básicas se relacionam entre si. Para o caso dos números complexos, sendo r e i números reais, esta axiomatização é dada por⁶:

$$\begin{aligned} & \text{complexo}(\text{cria_compl}(r, i)) = \text{verdadeiro} \\ & \text{compl_zero}(c) = \text{compl_iguais}(c, \text{cria_compl}(0, 0)) \\ & \text{imag_puro}(\text{cria_compl}(0, b)) = \text{verdadeiro} \\ & \text{p_real}(\text{cria_compl}(r, i)) = r \\ & \text{p_imag}(\text{cria_compl}(r, i)) = i \\ & \text{cria_compl}(\text{p_real}(c), \text{p_imag}(c)) = \begin{cases} c & \text{se } \text{complexo}(c) \\ \perp & \text{em caso contrário} \end{cases} \\ & \text{compl_iguais}(\text{cria_compl}(x, y), \text{cria_compl}(w, z)) = (x = w) \wedge (y = z) \end{aligned}$$

Neste passo especificam-se as relações obrigatoriamente existentes entre as operações básicas, para que estas definam o tipo de um modo coerente.

⁶O símbolo \perp representa indefinido.

10.3.3 Escolha da representação

O terceiro passo na definição de um tipo de informação consiste em escolher uma *representação* para os elementos do tipo em termos de outros tipos existentes.

No caso dos números complexos, iremos considerar que o complexo $a + bi$ é representado por um dicionário em que o valor da chave ' r ' é a parte real e o valor da chave ' i ' é a parte imaginária, assim $\Re[a+bi] = \{r:\Re[a], i:\Re[b]\}$.

10.3.4 Realização das operações básicas

O último passo na definição de um tipo de informação consiste em realizar as operações básicas definidas no primeiro passo em termos da representação definida no terceiro passo. É evidente que a realização destas operações básicas deve verificar a axiomatização definida no segundo passo.

Para os números complexos, definimos as seguintes funções que correspondem à realização das operações básicas:

1. *Construtores*:

```
def cria_compl(r, i):
    if numero(r) and numero(i):
        return {'r':r, 'i':i}
    else:
        raise ValueError ('cria_compl: argumento errado')
```

A função de tipo lógico, `numero` tem o valor `True` apenas se o seu argumento for um número (`int` ou `float`). Esta função é definida do seguinte modo:

```
def numero(x):
    return isinstance(x, (int, float))
```

Recordese da página 121, que a função `isinstance` determina se o tipo do seu primeiro argumento pertence ao tuplo que é seu segundo argumento.

2. *Selectores*:

```
def p_real(c):
    return c['r']

def p_imag(c):
    return c['i']
```

3. Reconhecedores:

```
def complexo(c):
    if isinstance(c, (dict)):
        if len(c) == 2 and 'r' in c and 'i' in c:
            if numero(c['r']) and numero(c['i']):
                return True
            else:
                return False
        else:
            return False
    else:
        return False
```

A função `complexo`, sendo um reconhecedor, pode receber qualquer tipo de argumento, devendo ter o valor `True` apenas se o seu argumento é um número complexo. Assim, o primeiro teste desta função verifica se o seu argumento é um dicionário, `isinstance(c, (dict))`. Se o fôr, sabemos que podemos calcular o seu comprimento, `len(c)`, e determinar se as chaves '`r`' e '`i`' existem no dicionário, `'r' in c and 'i' in c`. Só após todos estes testes deveremos verificar se os valores associados às chaves são números.

```
def compl_zero(c) :
    return zero(c['r']) and zero(c['i'])

def imag_puro(c):
    return zero(c['r'])
```

As funções `compl_zero` e `imag_puro` utilizam a função `zero` que recebe como argumento um número e tem o valor `True` se esse número for zero e o valor `False` em caso contrário. A definição da função `zero` é necessária,

uma vez que os componentes de um número complexo são números reais, e o teste para determinar se um número real é zero deve ser traduzido por um teste de proximidade em valor absoluto de zero⁷. Por esta razão, a função `zero` poderá ser definida como:

```
def zero(x):
    return abs(x) < 0.0001
```

em que 0.0001 corresponde ao valor do erro admissível.

4. Testes:

```
def compl_iguais(c1, c2) :
    return igual(c1['r'], c2['r']) and \
        igual(c1['i'], c2['i'])
```

A função `compl_iguais`, utiliza o predicado `igual` para testar a igualdade de dois números reais. Se 0.0001 corresponder ao erro admissível, esta função é definida por:

```
def igual(x, y):
    return abs(x - y) < 0.0001
```

No passo correspondente à realização das operações básicas, devemos também especificar os transformadores de entrada e de saída. Neste capítulo, apenas nos preocupamos com o transformador de saída, o qual é traduzido pela função `escreve_compl` apresentada na página 268.

10.4 Barreiras de abstracção

Depois de concluir todos os passos na definição de um tipo abstracto de informação (a definição de como os elementos do tipo são utilizados e a definição de como eles são representados, bem como a escrita de funções correspondentes às respectivas operações), podemos juntar o conjunto de funções correspondente ao tipo a um programa que utiliza o tipo como se este fosse um tipo embutido na linguagem. O programa acede a um conjunto de operações que são específicas

⁷Testes semelhantes foram utilizados nas Secções 3.4.5, 6.2.1 e 6.2.2.

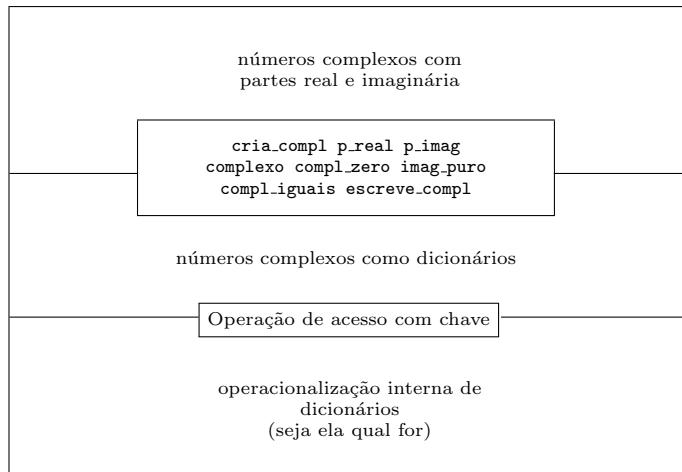


Figura 10.1: Camadas de abstracção na definição de complexos.

do tipo e que, na realidade, caracterizam o seu comportamento como tipo de informação. Qualquer manipulação efectuada sobre uma entidade de um dado tipo deve *apenas* recorrer às operações básicas para esse tipo.

Ao construir um novo tipo abstracto de informação estamos a criar uma nova camada conceptual na nossa linguagem, a qual corresponde ao tipo definido. Esta camada é separada da camada em que o tipo não existe por barreiras de abstracção. Conceptualmente, estas barreiras impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas. Na Figura 10.1 apresentamos as camadas e as barreiras de abstracção existentes num programa depois da definição do tipo complexo.

Em cada uma destas camadas, a barreira de abstracção separa os programas que usam a abstracção de dados (que estão situados acima da barreira) dos programas que realizam a abstracção de dados (que estão situados abaixo da barreira).

Para ajudar a compreender a necessidade da separação destas camadas, voltemos a considerar o exemplo dos números complexos. Suponhamos que, com base na representação utilizando dicionários, escrevímos as operações básicas para os números complexos. Podemos agora utilizar o módulo correspondente ao tipo complexo num programa que manipule números complexos. De acordo com a metodologia estabelecida para os tipos abstractos de informação, ape-

nas devemos aceder aos elementos do tipo complexo através das suas operações básicas (as operações disponíveis para uso público). Ou seja, embora possamos saber que os complexos são representados através de dicionários, não somos autorizados a manipulá-los através da referência à chave associada a cada elemento de um complexo.

Definindo os números complexos através de funções, como o fizemos na secção anterior, nada na linguagem nos impede de manipular directamente a sua representação. Ou seja, se um programador tiver conhecimento do modo como um tipo é representado, nada o impede de violar a regra estabelecida na metodologia e aceder directamente aos componentes do tipo através da manipulação da estrutura por que este é representado. Por exemplo, se $c1$ e $c2$ corresponderem a números complexos, esse programador poderá pensar em adicionar directamente $c1$ e $c2$, através de:

$$\{ 'r' : (c1['r'] + c2['r']), 'i' : (c1['i'] + c2['i']) \}.$$

No entanto, esta decisão corresponde a uma *má prática de programação*, a qual deve ser sempre evitada, pelas seguintes razões:

1. A manipulação directa da representação do tipo faz com que o programa seja dependente dessa representação. Suponhamos que, após o desenvolvimento do programa, decidímos alterar a representação de números complexos de dicionários para outra representação qualquer. No caso da regra da metodologia ter sido violada, ou seja se uma barreira de abstracção tiver sido “quebrada”, o programador teria que percorrer todo o programa e alterar todas as manipulações directas da estrutura que representa o tipo.
2. O programa torna-se mais difícil de escrever e de compreender, uma vez que a manipulação directa da estrutura subjacente faz com que se perca o nível de abstracção correspondente à utilização do tipo.

As linguagens de programação desenvolvidas antes do aparecimento da metodologia para os tipos abstractos de informação não possuem mecanismos para garantir que toda a utilização dos elementos de um dado tipo é efectuada recorrendo exclusivamente às operações específicas desse tipo. As linguagens de programação mais recentes (por exemplo, Ada, CLU, Mesa, Modula-2 e Python) fornecem mecanismos que garantem que as manipulações efectuadas sobre os

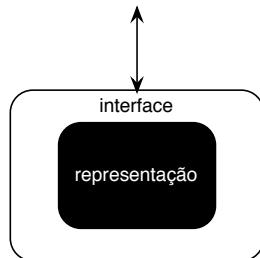


Figura 10.2: Módulo correspondente a um tipo abstracto de informação.

elementos de um tipo apenas utilizam as operações básicas desse tipo. Este comportamento é obtido através da utilização de dois conceitos chamados encapsulação da informação e anonimato da representação.

A *encapsulação da informação*⁸ corresponde ao conceito de que o conjunto de funções que corresponde ao tipo de informação engloba toda a informação referente ao tipo. Estas funções estão representadas dentro de um módulo que está protegido dos acessos exteriores. Este módulo comporta-se de certo modo como um bloco, com a diferença que “exporta” certas operações, permitindo apenas os acessos às operações exportadas. O *anonimato da representação*⁹ corresponde ao conceito de que este módulo guarda como segredo o modo escolhido para representar os elementos do tipo. O único acesso que o programa tem aos elementos do tipo é através das operações básicas definidas dentro do módulo que corresponde ao tipo. Na Figura 10.2 apresentamos de uma forma esquemática o módulo conceptual correspondente a um tipo abstracto de informação. Este módulo contém uma parte que engloba a representação do tipo, representação essa que está escondida do resto do programa, e um conjunto de funções que efectuam a comunicação (*interface*) entre o programa e a representação interna.

A vantagem das linguagens que incorporam os mecanismos da metodologia dos tipos abstractos de informação reside no facto de estas *proibirem efectivamente* a utilização de um tipo abstracto através da manipulação directa da sua representação: toda a informação relativa ao tipo está contida no módulo que o define, o qual guarda como segredo a representação escolhida para o tipo.

⁸Do inglês, “data encapsulation”.

⁹Do inglês, “information hiding”.

10.5 Objectos

De modo a poder garantir a encapsulação da informação e anonimato da representação em Python, precisamos de recorrer ao conceito de objecto. Um *objecto* é uma entidade computacional que apresenta, não só informação interna (a representação de um elemento de um tipo), como também um conjunto de operações que definem o seu comportamento.

A manipulação de objectos não é baseada em funções que calculam valores, mas sim no conceito de uma entidade que recebe solicitações e reage apropriadamente a essas solicitações, recorrendo à informação interna do objecto. As funções associadas a um objecto são vulgarmente designadas por *métodos*. É importante distinguir entre solicitações e métodos: uma solicitação é um pedido feito a um objecto; um método é a função utilizada pelo objecto para processar essa solicitação.

Para definir objectos em Python, criamos um módulo com o nome do objecto, associando-lhe os métodos correspondentes a esse objecto. Em notação BNF, um objecto em Python é definido do seguinte modo¹⁰:

```

⟨definição de objecto⟩ ::= class ⟨nome⟩ {⟨(nome)⟩} : [CR]
                                [TAB] ⟨definição de método⟩+
⟨definição de método⟩ ::= def ⟨nome⟩ (self{, ⟨parâmetros formais⟩}): [CR]
                                [TAB] ⟨corpo⟩

```

Os símbolos não terminais ⟨nome⟩, ⟨parâmetros formais⟩ e ⟨corpo⟩ foram definidos nos capítulos 2 e 3.

Nesta definição:

1. a palavra `class` (um símbolo terminal) indica que se trata da definição de um objecto. Existem duas alternativas para esta primeira linha: (1) pode apenas ser definido um nome, ou (2) pode ser fornecido um nome seguido de outro nome entre parênteses. Neste capítulo apenas consideramos a primeira alternativa, sendo a segunda considerada no Capítulo 11. O nome imediatamente após a palavra `class` representa o nome do objecto que está a ser definido;
2. A ⟨definição de método⟩ corresponde à definição de um método que está

¹⁰ Esta definição corresponde em Python a uma ⟨definição⟩.

associado ao objecto. Podem ser definidos tantos métodos quantos se desejar, mas existe sempre pelo menos um método cujo nome é `__init__` (este aspecto não está representado na nossa expressão em notação BNF);

3. Os parâmetros formais de um método contêm sempre, como primeiro elemento, um parâmetro com o nome `self`¹¹.

Os objectos contêm informação interna. Esta informação interna é caracterizada por uma ou mais variáveis. Os nomes destas variáveis utilizam a notação de `(nome composto)` apresentada na página 110, em que o primeiro nome é sempre `self`.

Para caracterizar a informação interna associada a um objecto correspondente a um número complexo, sabemos que precisamos de duas entidades, nomeadamente, a parte real e a parte imaginária desse número complexo. Vamos abandonar a ideia de utilizar um dicionário para representar números complexos, optando por utilizar duas variáveis separadas, `self.r` e `self.i`, hipótese que tínhamos abandonado à partida no início deste capítulo (página 266). A razão da nossa decisão prende-se com o facto destas duas variáveis existirem dentro do objecto que corresponde a um número complexo e consequentemente, embora sejam variáveis separadas, elas estão mutuamente ligadas por existirem dentro do mesmo objecto.

Consideremos a seguinte definição de um objecto com o nome `compl`¹² correspondente a um número complexo:

```
class compl:

    def __init__(self, real, imag):
        if isinstance(real, (int, float)) and \
           isinstance(imag, (int, float)):
            self.r = real
            self.i = imag
        else:
            raise ValueError ('complexo: argumento errado')
```

¹¹A palavra inglesa “self” corresponde à individualidade ou identidade de uma pessoa ou de uma coisa.

¹²Por simplicidade de apresentação, utilizamos o operador relacional `==` para comparar os elementos de um número complexo, quando, na realidade, deveríamos utilizar os predicados `zero` (apresentado na página 278) e `igual` (apresentado na página 278).

```

def p_real(self):
    return self.r

def p_imag(self):
    return self.i

def compl_zero(self):
    return self.r == 0 and self.i == 0

def imag_puro(self):
    return self.r == 0

def compl_iguais(self, outro):
    return self.r == outro.p_real() and \
           self.i == outro.p_imag()

def escreve(self):
    if self.i >= 0:
        print(self.r, '+', self.i, 'i')
    else:
        print(self.r, '-', abs(self.i), 'i')

```

A execução desta definição pelo Python dá origem à criação do objecto `compl`, estando este objecto associado a métodos que correspondem aos nomes `__init__`, `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

A partir do momento em que um objecto é criado, passa a existir uma nova função em Python cujo nome corresponde ao nome do objecto e cujos parâmetros correspondem a todos os parâmetros formais da função `__init__` que lhe está associada, com excepção do parâmetro `self`. O comportamento desta função é definido pela função `__init__` correspondente ao objecto. No nosso exemplo, esta é a função `compl` de dois argumentos. A chamada à função `compl(3, 2)` devolve o objecto correspondente ao número complexo $3 + 2i$. Podemos, assim, originar a seguinte interacção:

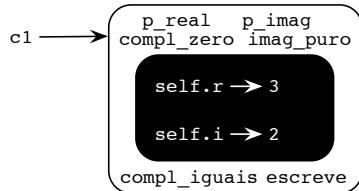


Figura 10.3: Objecto correspondente ao número complexo $3 + 2i$.

```

>>> compl
<class '__main__.compl'>
>>> c1 = compl(3, 2)
>>> c1
<__main__.compl object at 0x11fab90>
  
```

As duas primeiras linhas desta interacção mostram que `compl` é uma classe (um novo tipo de entidades computacionais que existem nos nossos programas) e as duas últimas linhas mostram que o nome `c1` está associado a um objecto, localizado na posição de memória `0x11fab90`.

O resultado da execução da instrução `c1 = compl(3, 2)`, foi a criação do objecto apresentado na Figura 10.3 e a associação do nome `c1` a este objecto. Este objecto “esconde” do exterior a representação do seu número complexo (dado pelos valores das variáveis `self.r` e `self.i`), fornecendo seis métodos para interactuar com o número complexo, cujos nomes são `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`.

Os métodos associados a um objecto correspondem a funções que manipulam as variáveis associadas ao objecto, variáveis essas que estão guardadas *dentro* do objecto, correspondendo a um ambiente local ao objecto, o qual existe enquanto o objecto correspondente existir. Para invocar os métodos associados a um objecto utiliza-se a notação de `<nome composto>` apresentada na página 110, em que o nome é composto pelo nome do objecto, seguido do nome do método¹³. Por exemplo `c1.p_real` é o nome do método que devolve a parte real do complexo correspondente a `c1`. Para invocar a execução de um método, utilizam-se todos os parâmetros formais da função correspondente ao método com excepção do parâmetro `self`. Assim, podemos originar a seguinte interacção:

¹³Note-se que as funções que manipulam ficheiros apresentadas no Capítulo 8 e a função `lower` apresentada na página 248 do Capítulo 9, na realidade correspondem a métodos.

```
>>> c1 = compl(3, 2)
>>> c1.p_real()
3
>>> c1.p_imag()
2
>>> c1.escreve()
3 + 2 i
>>> c2 = compl(2, -2)
>>> c2.escreve()
2 - 2 i
```

Na interacção anterior criámos dois objectos `c1` e `c2` correspondentes a números complexos. Cada um destes objectos armazena a parte inteira e a parte imaginária de um número complexo, apresentando comportamentos semelhantes, apenas diferenciados pela sua identidade, um corresponde ao complexo $3 + 2i$ e o outro ao complexo $2 - 2i$.

É importante fazer duas observações em relação à nossa definição do objecto `compl`. Em primeiro lugar, não definimos um reconhecedor correspondente à operação básica *complexo*. Na realidade, ao criar um objecto, o Python associa automaticamente o nome do objecto à entidade que o representa. A função embutida `type` devolve o tipo do seu argumento, como o mostra a seguinte interacção:

```
>>> c1 = compl(3, 5)
>>> type(c1)
<class '__main__.compl'>
```

A função `isinstance`, introduzida na página 276, permite determinar se uma dada entidade corresponde a um complexo:

```
>>> c1 = compl(9, 6)
>>> isinstance(c1, compl)
True
```

Existe assim, uma solicitação especial que pode ser aplicada a qualquer objecto e que pede a identificação do objecto. Por esta razão, ao criarmos um tipo abstracto de informação recorrendo a objectos, não necessitamos de escrever o reconhecedor que distingue os elementos do tipo dos restantes elementos.

A segunda observação respeita o método `compl_iguais`. Na nossa definição das operações básicas para complexos, definimos a função `compl_iguais` que permite decidir se dois complexos são iguais. Na nossa implementação, o método `compl_iguais` apenas tem um argumento correspondente a um complexo. Este método compara o complexo correspondente ao seu objecto (`self`) com outro complexo qualquer. Assim, por omissão, o primeiro argumento da comparação com outro complexo é o complexo correspondente ao objecto em questão, como a seguinte interacção ilustra:

```
>>> c1 = compl(9, 6)
>>> c2 = compl(7, 6)
>>> c1.compl_iguais(c2)
False
>>> c2.compl_iguais(c1)
False
```

Em programação com objectos existe uma abstracção chamada classe. Uma *classe* corresponde a uma infinidade de objectos com as mesmas variáveis e com o mesmo comportamento. É por esta razão, que na definição de um objecto o Python usa a palavra `class` (ver a definição apresentada na página 282). Por exemplo, os complexos correspondem a uma classe de objectos que armazenam uma parte real, `self.r`, e uma parte imaginária, `self.i`, e cujo comportamento é definido pelas funções `__init__`, `p_real`, `p_imag`, `compl_zero`, `imag_puro`, `compl_iguais` e `escreve`. Os complexos `c1` e `c2` criados na interacção anterior correspondem a instâncias do objecto `compl`, sendo conhecidos como *instâncias* da classe `compl`.

Uma classe é um potencial gerador de instâncias, objectos cujas variáveis e comportamento são definidos pela classe. São as instâncias que contêm os valores das variáveis associadas à classe e que reagem a solicitações – a classe apenas define como são caracterizadas as variáveis e o comportamento das suas instâncias.

As classes podem ser representadas graficamente como se indica na Figura 10.4. Uma classe corresponde a um rectângulo com três partes, contendo, respectivamente, o nome da classe, o nome das variáveis e o nome dos métodos. As instâncias são caracterizadas pelo seu nome (dentro de um rectângulo) e estão ligadas à classe correspondente por uma linha a tracejado.

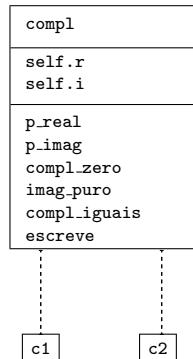


Figura 10.4: Representação da classe `compl` e duas de suas instâncias.

10.6 Notas finais

Neste capítulo, apresentámos a metodologia dos tipos abstractos de informação, que permite separar as propriedades abstractas de um tipo do modo como ele é realizado numa linguagem de programação. Esta separação permite melhorar a tarefa de desenvolvimento de programas, a facilidade de leitura de um programa e torna os programas independentes da representação escolhida para os tipos de informação. Esta metodologia foi introduzida por [Liskof e Zilles, 1974] e posteriormente desenvolvida por [Liskof e Guttag, 1986]. O livro [Dale e Walker, 1996] apresenta uma introdução detalhada aos tipos abstractos de informação.

De acordo com esta metodologia, sempre que criamos um novo tipo de informação devemos seguir os seguintes passos: (1) especificar as operações básicas para o tipo; (2) especificar as relações que as operações básicas têm de satisfazer; (3) escolher uma representação para o tipo; (4) realizar as operações básicas com base na representação escolhida.

Um tópico importante que foi discutido superficialmente neste capítulo tem a ver com garantir que as operações básicas que especificámos para o tipo na realidade caracterizam o tipo que estamos a criar. Esta questão é respondida fornecendo uma axiomatização para o tipo. Uma axiomatização corresponde a uma apresentação formal das propriedades do tipo. Exemplos de axiomatizações podem ser consultadas em [Hoare, 1972] e [Manna e Waldinger, 1985].

Os tipos abstractos de informação estão relacionados com o tópico estudado em matemática relacionado com a *teoria das categorias* [Asperti e Longo, 1991],

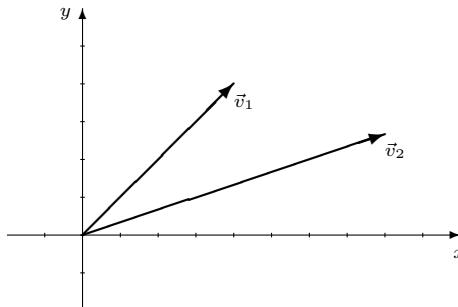


Figura 10.5: Exemplo de vectores.

[Simmons, 2011].

10.7 Exercícios

- Suponha que desejava criar o tipo vector em Python. Um vector num referencial cartesiano pode ser representado pelas coordenadas da sua extremidade (x, y) , estando a sua origem no ponto $(0, 0)$, ver Figura 10.5. Podemos considerar as seguintes operações básicas para vectores:

- *Construtor:*

$vector : real \times real \mapsto vector$

$vector(x, y)$ tem como valor o vector cuja extremidade é o ponto (x, y) .

- *Selectores:*

$abcissa : vector \mapsto real$

$abcissa(v)$ tem como valor a abcissa da extremidade do vector v .

$ordenada : vector \mapsto real$

$ordenada(v)$ tem como valor a ordenada da extremidade do vector v .

- *Reconhecedores:*

$vector : universal \mapsto lógico$

$vector(arg)$ tem valor *verdadeiro* apenas se arg é um vector.

$vector_nulo : vector \mapsto lógico$

$vector_nulo(v)$ tem valor *verdadeiro* apenas se v é o vector $(0, 0)$.

- *Teste:*

vectores_iguais : vector × vector ↪ lógico

vectores_iguais(v_1, v_2) tem valor verdadeiro apenas se os vectores v_1 e v_2 são iguais.

- (a) Defina uma representação para vectores utilizando tuplos.
 - (b) Escolha uma representação externa para vectores e escreva o transformador de saída.
 - (c) Implemente o tipo vector utilizando funções.
 - (d) Defina a classe `vector`.
2. Tendo em atenção as operações básicas sobre vectores da pergunta anterior, escreva funções em Python (quer usando a representação de funções, quer usando objectos) para:
 - (a) Somar dois vectores. A soma dos vectores representados pelos pontos (a, b) e (c, d) é dada pelo vector $(a + c, b + d)$.
 - (b) Calcular o produto escalar de dois vectores. O produto escalar dos vectores representados pelos pontos (a, b) e (c, d) é dado pelo real $a.c + b.d$.
 - (c) Determinar se um vector é colinear com o eixo dos xx .
 3. Suponha que pretendia representar pontos num espaço carteziano. Cada ponto é representado por duas coordenadas, a do eixo dos xx e a do eixo dos yy , ambas contendo valores reais.
 - (a) Especifique as operações básicas do tipo abstracto de informação `ponto`.
 - (b) Defina a classe `ponto` para representar pontos no espaço carteziano.
 - (c) Escreva uma função em Python que recebe duas variáveis do tipo `ponto` e que determina a distância entre esses pontos. A distância entre os pontos (x_1, y_1) e (x_2, y_2) é dada por $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. Suponha que já existem as funções `quadrado` e `sqrt`.
 - (d) Escreva uma função em Python que recebe como argumento um `ponto` e que determina o quadrante em que este se encontra. A sua função deve devolver um inteiro entre 1 e 4.

4. Suponha que quer representar o tempo, dividindo-o em horas e minutos.
- Especifique e implemente um tipo abstracto *tempo*. No seu tipo, o número de minutos está compreendido entre 0 e 59, e o número de horas apenas está limitado inferiormente a zero. Por exemplo 546:37 é um tempo válido.
 - Com base no tipo da alínea anterior, escreva as seguintes funções:
 - *depois* : *tempo* × *tempo* \mapsto lógico
depois(*t*₁, *t*₂) tem o valor *verdadeiro*, se *t*₁ corresponder a um instante de tempo posterior a *t*₂.
 - *num_minutos* : *tempo* \mapsto inteiro
num_minutos(*t*) tem como valor o número de minutos entre o momento 0 horas e 0 minutos e o tempo *t*.
5. (a) Especifique as operações básicas do tipo abstracto de informação *carta* o qual é caracterizado por um naipe (espadas, copas, ouros e paus) e por um valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K).
- Defina a classe **carta** correspondente ao tipo da alínea anterior.
 - Usando o tipo *carta*, defina uma função em Python que devolve uma lista em que cada elemento corresponde a uma carta de um baralho.
 - Usando o tipo *carta* e recorrendo à função `random()`, a qual produz um número aleatório no intervalo [0, 1[, escreva uma função, **baralha**, que recebe uma lista correspondente a um baralho de cartas e baralha aleatoriamente essas cartas, devolvendo a lista que corresponde às cartas baralhadas. SUGESTÃO: percorra sucessivamente as cartas do baralho trocando cada uma delas por uma outra carta seleccionada aleatoriamente.

Capítulo 11

Programação com objectos

And I haven't seen the two Messengers, either. They're both gone to the town. Just look along the road, and tell me if you can see either of them.

Lewis Carroll, *Through the Looking Glass*

No capítulo anterior introduzimos o conceito de objecto, uma entidade que contém um conjunto de variáveis internas e um conjunto de operações que manipulam essas variáveis. A abordagem que fizemos a objectos correspondeu a criar objectos que correspondem a constantes do tipo complexo. Neste capítulo começamos por apresentar um tipo de informação correspondente a objectos cujas variáveis internas variam ao longo do tempo, são tipos mutáveis, e analisamos qual a influência que este aspecto apresenta no seu comportamento.

A utilização de objectos em programação dá origem a um paradigma de programação conhecido por *programação com objectos*¹. Neste capítulo também abordamos alguns aspectos associados à programação com objectos.

¹Do inglês “object-oriented programming”. Em português, este tipo de programação também é conhecido por programação orientada a objectos, programação orientada por objectos, programação por objectos e programação orientada aos objectos.

11.1 O tipo conta bancária

Dizemos que uma entidade tem *estado* se o seu comportamento é influenciado pela sua história. Vamos ilustrar o conceito de entidade com estado através da manipulação de uma conta bancária. Para qualquer conta bancária, a resposta à questão “posso levantar 20 euros?” depende da história dos depósitos e dos levantamentos efectuados nessa conta. A situação de uma conta bancária pode ser caracterizada por um dos seguintes aspectos: o saldo num dado instante, ou a sequência de todos os movimentos na conta desde a sua criação. A situação da conta bancária é caracterizada computacionalmente através do conceito de estado. O estado de uma entidade é caracterizado por uma ou mais variáveis, as *variáveis de estado*, as quais mantêm informação sobre a história da entidade, de modo a poder determinar o comportamento da entidade.

Suponhamos que desejávamos modelar o comportamento de uma conta bancária. Por uma questão de simplificação, caracterizamos uma conta bancária apenas pelo seu saldo, ignorando todos os outros aspectos, que sabemos estarem associados a contas bancárias, como o número, o titular, o banco, a agência, etc.

Para modelar o comportamento de uma conta bancária, vamos criar uma classe, chamada `conta`, cujo estado interno é definido pela variável `saldo` e está associada a funções que efectuam depósitos, levantamentos e consultas de saldo. Esta classe, para além do construtor (`__init__`), apresenta um selector que devolve o valor do saldo (`consulta`) e dois modificadores, `deposito` e `levantamento`, que, respectivamente, correspondem às operações de depósito e de levantamento.

```
class conta:

    def __init__(self, quantia):
        self.saldo = quantia

    def consulta(self):
        return self.saldo

    def deposito(self, quantia):
        self.saldo = self.saldo + quantia
        return self.saldo
```

```
def levantamento(self, quantia):
    if self.saldo - quantia >= 0:
        self.saldo = self.saldo - quantia
        return self.saldo
    else:
        print('Saldo insuficiente')
```

Com esta classe podemos obter a seguinte interacção:

```
>>> conta_01 = conta(100)
>>> conta_01.deposito(50)
150
>>> conta_01.consulta()
150
>>> conta_01.levantamento(120)
30
>>> conta_01.levantamento(50)
Saldo insuficiente
>>> conta_02 = conta(250)
>>> conta_02.deposito(50)
300
>>> conta_02.consulta()
300
```

Uma conta bancária, tal como a que acabámos de definir, corresponde a uma entidade que não só possui um estado (o qual é caracterizado pelo seu saldo) mas também está associada a um conjunto de comportamentos (as funções que efectuam depósitos, levantamentos e consultas). A conta bancária evolui de modo diferente quando fazemos um depósito ou um levantamento.

Sabemos que em programação com objectos existe uma abstracção chamada classe. Uma *classe* corresponde a uma infinidade de objectos com as mesmas variáveis de estado e com o mesmo comportamento. Por exemplo, a conta bancária que definimos corresponde a uma classe de objectos cujo estado é definido pela variável `saldo` e cujo comportamento é definido pelas funções `levantamento`, `deposito` e `consulta`. As contas `conta_01` e `conta_02`, definidas na nossa interacção, correspondem a *instâncias* da classe `conta`. Como já vimos

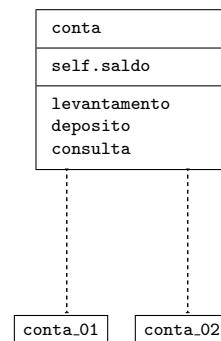


Figura 11.1: Representação da classe `conta` e suas instâncias.

no Capítulo 10, as classes e instâncias são representadas graficamente como se mostra na Figura 11.1.

Uma classe é um potencial gerador de instâncias, objectos cujas variáveis de estado e comportamento são definidos pela classe. São as instâncias que contêm o estado interno e que reagem a solicitações – a classe apenas define como é caracterizado o estado e o comportamento das suas instâncias.

Ao trabalhar com objectos, é importante distinguir entre identidade e igualdade. Considerando a classe `conta`, podemos definir as contas `conta_01` e `conta_02` do seguinte modo:

```

>>> conta_01 = conta(100)
>>> conta_02 = conta(100)
  
```

Será que os dois objectos `conta_01` e `conta_02` são iguais? Embora eles tenham sido definidos exactamente da mesma maneira, como instâncias da mesma classe, estes dois objectos têm estados distintos, como se demonstra pela seguinte interacção:

```

>>> conta_01 = conta(100)
>>> conta_02 = conta(100)
>>> conta_01.levantamento(25)
75
>>> conta_01.levantamento(35)
40
>>> conta_02.deposito(120)
  
```

```
220
>>> conta_01.consulta()
40
>>> conta_02.consulta()
220
```

Suponhamos agora que definíamos:

```
>>> conta_01 = conta(100)
>>> conta_02 = conta_01
```

Repare-se que agora `conta_01` e `conta_02` são o *mesmo* objecto (correspondem na realidade a uma conta conjunta). Por serem o mesmo objecto, qualquer alteração a uma destas contas reflecte-se na outra:

```
>>> conta_01 = conta(100)
>>> conta_02 = conta_01
>>> conta_01.consulta()
100
>>> conta_02.levantamento(30)
70
>>> conta_01.consulta()
70
```

11.2 Classes, subclasses e herança

Uma das vantagens da programação com objectos é a possibilidade de construir entidades reutilizáveis, ou seja, componentes que podem ser usados em programas diferentes através da especialização para necessidades específicas.

Para ilustrar o aspecto da reutilização, consideremos, de novo, a classe `conta`. Esta classe permite a criação de contas bancárias, com as quais podemos efectuar levantamentos e depósitos. Sabemos, no entanto, que qualquer banco oferece aos seus clientes a possibilidade de abrir diferentes tipos de contas, existem contas ordenado que permitem apresentar saldos negativos (até um certo limite), existem contas jovem que não permitem saldos negativos, mas que não impõem restrições mínimas para o saldo de abertura, mas, em contrapartida, não pagam

juros, e assim sucessivamente. Quando um cliente abre uma nova conta, tipicamente define as características da conta desejada: “quero uma conta em que não exista pagamento adicional pela utilização de cheques”, “quero uma conta que ofereça 2% de juros por ano”, etc.

Todos estes tipos de contas são “contas bancárias”, mas apresentam características (ou especializações) que variam de conta para conta. A programação tradicional, aplicada ao caso das contas bancárias, levaria à criação de funções em que o tipo de conta teria de ser testado quando uma operação de abertura, de levantamento ou de depósito era efectuada. Em programação com objectos, a abordagem seguida corresponde à criação de classes que especializam outras classes.

Comecemos por considerar uma classe de contas genéricas, `conta_gen`, a qual sabe efectuar levantamentos, depósitos e consultas ao saldo. Esta classe não impõe qualquer restrição durante o levantamento.

A classe `conta_gen`, correspondente a uma conta genérica, pode ser realizada através da seguinte definição:

```
class conta_gen:

    def __init__(self, quantia):
        self.saldo = quantia

    def consulta(self):
        return self.saldo

    def deposito(self, quantia):
        self.saldo = self.saldo + quantia
        return self.saldo

    def levantamento(self, quantia):
        self.saldo = self.saldo - quantia
        return self.saldo
```

Tendo definido a classe `conta_gen`, podemos definir a instância `conta_gen_01` e efectuar as seguintes operações:

```
>>> conta_gen_01 = conta_gen(20)
>>> conta_gen_01.consulta()
20
>>> conta_gen_01.levantamento(300)
-280
>>> conta_gen_01.deposito(400)
120
```

Suponhamos agora que estávamos interessados em criar os seguintes tipos de contas:

- *conta ordenado*: esta conta está associada à transferência mensal do ordenado do seu titular e está sujeita às seguintes restrições:
 - a sua abertura exige um saldo mínimo igual ao valor do ordenado a ser transferido para a conta.
 - permite saldos negativos, até um montante igual ao ordenado.
- *conta jovem*: conta feita especialmente para jovens e que está sujeita às seguintes restrições:
 - a sua abertura não exige um saldo mínimo.
 - não permite saldos negativos.

Para criar as classes `conta_ordenado` e `conta_jovem`, podemos pensar em duplicar o código associado à classe `conta_gen`, fazendo as adaptações necessárias nos respectivos métodos. No entanto, esta é uma má prática de programação, pois não só aumenta desnecessariamente a quantidade de código, mas também porque qualquer alteração realizada sobre a classe `conta_gen` não se propaga automaticamente às classes `conta_ordenado` e `conta_jovem`, as quais são contas.

Para evitar estes inconvenientes, em programação com objectos existe o conceito de subclasse. Diz-se que uma classe é uma *subclasse* de outra classe, se a primeira corresponder a uma especialização da segunda. Ou seja, o comportamento da subclasse corresponde ao comportamento da superclasse, excepto no caso em que comportamento específico está indicado para a subclasse. Diz-se que a subclasse *herda* o comportamento da superclasse, excepto quando este é explicitamente alterado na subclasse.

É na definição de subclasses que temos que considerar a parte opcional da ⟨definição de objecto⟩ apresentada na página 282, e reproduzida aqui para facilitar a leitura:

```
⟨definição de objecto⟩ ::= class <nome> {(<nome>)}: [CR]
[TAB] <definição de método>+
```

Ao considerar uma definição de classe da forma `class<nome1> (<nome2>)`, estamos a definir a classe `<nome1>` como uma subclasses de `<nome2>`. Neste caso, a classe `<nome1>` automaticamente “herda” todas as variáveis e métodos definidos na classe `<nome2>`.

Por exemplo, definindo a classe `conta_ordenado` como

```
class conta_ordenado(conta_gen),
```

se nada for dito em contrário, a classe `conta_ordenado` passa automaticamente a ter a variável `self.saldo` e os métodos `__init__`, `deposito`, `levantamento` e `consulta` definidos na classe `conta_gen`. Se, associado à classe `conta_ordenado`, for definido um método com o mesmo nome de um método da classe `conta_gen`, este método, na classe `conta_ordenado`, sobrepor-se ao método homónimo da classe `conta_gen`. Se forem definidos novos métodos na classe `conta_ordenado`, estes pertencem apenas a essa classe, não existindo na classe `conta_gen`.

Em resumo, as classes podem ter *subclasses* que correspondem a especializações dos seus elementos. As subclasses *herdam* as variáveis de estado e os métodos das superclasses, salvo indicação em contrário.

A classe `conta_ordenado` pode ser definida do seguinte modo:

```
class conta_ordenado (conta_gen):

    def __init__(self, quantia, ordenado):
        if quantia >= ordenado:
            self.saldo = quantia
            self.ordenado = ordenado
        else:
            print('O saldo deve ser maior que o ordenado')

    def levantamento(self, quantia):
```

```

    if quantia <= self.saldo + self.ordenado:
        self.saldo = self.saldo - quantia
        return self.saldo
    else:
        print('Saldo insuficiente')

```

Nesta classe, o método `__init__` é redefinido com um argumento adicional correspondente ao ordenado, o qual é guardado numa variável própria da classe `conta_ordenado`. O método `levantamento` é também redefinido, de modo a verificar que o saldo nunca desce abaixo do simétrico do ordenado. Os métodos `deposito` e `consulta` são herdados da classe `conta_gen`.

Analogamente, a classe `conta_jovem` é definida como uma subclasse de `conta_gen`, redefinindo apenas o método `levantamento`.

```

class conta_jovem(conta_gen):

    def levantamento(self, quantia):
        if quantia <= self.saldo:
            self.saldo = self.saldo - quantia
            return self.saldo
        else:
            print('Saldo insuficiente')

```

Estas classes permitem-nos efectuar a seguinte interacção:

```

>>> conta_ord_01 = conta_ordenado(300, 500)
0 saldo deve ser maior que o ordenado
>>> conta_ord_01 = conta_ordenado(800, 500)
>>> conta_ord_01.levantamento(500)
300
>>> conta_ord_01.levantamento(500)
-200
>>> conta_jov_01 = conta_jovem(50)
>>> conta_jov_01.consulta()
50
>>> conta_jov_01.levantamento(100)
Saldo insuficiente

```

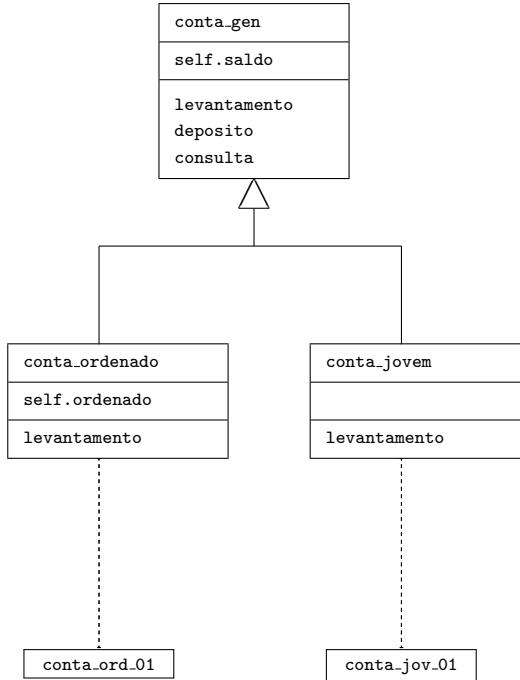


Figura 11.2: Hierarquia de contas e respectivas instâncias.

As classes correspondentes aos diferentes tipos de contas estão relacionadas entre si, podendo ser organizadas numa *hierarquia*. Nesta hierarquia, ao nível mais alto está a classe `conta_gen`, com duas subclasses, `conta_ordenado` e `conta_jovem`. Na Figura 11.2 mostramos esta hierarquia, bem como as instâncias `conta_ord_01` e `conta_jov_01`.

A importância da existência de várias classes e instâncias organizadas numa hierarquia, que estabelece relações entre classes e suas subclasses e instâncias, provém de uma forma de propagação associada, designada por *herança*. A herança consiste na transmissão das características duma classe às suas subclasses e instâncias. Isto significa, por exemplo, que todos os métodos associados à classe `conta_gen` são herdados por todas as subclasses e instâncias dessa classe, excepto se forem explicitamente alterados numa das subclasses.

Suponhamos agora que desejamos criar uma conta jovem que está protegida por um código de segurança, normalmente conhecido por PIN². Esta nova conta,

²Do inglês “Personal Identification Number”.

sendo uma `conta_jovem`, irá herdar as características desta classe. A nova classe que vamos definir, `conta_jovem_com_pin` terá um estado interno constituído por três variáveis, uma delas corresponde ao código de acesso, `pin`, a outra será uma `conta_jovem`, a qual só é acedida após o fornecimento do código de acesso correcto, finalmente, a terceira variável, `contador`, regista o número de tentativas falhadas de acesso à conta, bloqueando o acesso à conta assim que este número atingir o valor 3. Deste modo, o método que cria contas da classe `conta_jovem_com_pin` é definido por:

```
def __init__(self, quantia, codigo):
    self.conta = conta_jovem(quantia)
    self.pin = codigo
    self.contador = 0
```

Para aceder a uma conta do tipo `conta_jovem_com_pin` utilizamos o método `acede`, o qual, sempre que é fornecido um PIN errado aumenta em uma unidade o valor do contador de tentativas de acesso incorrectas. Ao ser efectuado um acesso com o PIN correcto, os valor do contador de tentativas incorrectas volta a ser considerado zero. O método `acede` devolve `True` se é efectuado um acesso com o PIN correcto e devolve `False` em caso contrário.

```
def acede(self, pin):
    if self.contador == 3:
        print('Conta bloqueada')
        return False
    elif pin == self.pin:
        self.contador = 0
        return True
    else:
        self.contador = self.contador + 1
        if self.contador == 3:
            print('Conta bloqueada')
        else:
            print('PIN incorrecto')
            print('Tem mais', 3-self.contador, 'tentativas')
    return False
```

Os métodos `levantamento`, `deposito` e `consulta` utilizam o método `acede`

para verificar a correcção do PIN e, em caso de este acesso estar correcto, utilizam o método correspondente da conta `conta_jovem` para aceder à conta `self.conta`. Note-se que `self.conta` é a instância da classe `conta_jovem` que está armazenada na instância da classe `conta_jovem_com_pin` que está a ser manipulada.

Finalmente, a classe `conta_jovem_com_pin` apresenta um método que não existe na classe `conta_jovem` e que corresponde à acção de alteração do PIN da conta.

A classe `conta_jovem_com_pin` é definida do seguinte modo:

```
class conta_jovem_com_pin (conta_jovem):

    def __init__(self, quantia, codigo):
        self.conta = conta_jovem(quantia)
        self.pin = codigo
        self.contador = 0

    def levantamento(self, quantia, pin):
        if self.acede(pin):
            return self.conta.levantamento(quantia)

    def deposito(self, quantia, pin):
        if self.acede(pin):
            return self.conta.deposito(quantia)

    def altera_codigo(self, pin):
        if self.acede(pin):
            novopin = input('Introduza o novo PIN\n--> ')
            print('Para verificação')
            verifica = input('Volte a introduzir o novo PIN\n--> ')
            if novopin == verifica:
                self.pin = novopin
                print('PIN alterado')
            else:
                print('Operação sem sucesso')
```

```
def consulta(self, pin):
    if self.acede(pin):
        return self.conta.consulta()

def acede(self, pin):
    if self.contador == 3:
        print('Conta bloqueada')
        return False
    elif pin == self.pin:
        self.contador = 0
        return True
    else:
        self.contador = self.contador + 1
        if self.contador == 3:
            print('Conta bloqueada')
        else:
            print('PIN incorrecto')
            print('Tem mais', 3-self.contador, 'tentativas')
    return False
```

Com esta classe, podemos originar a seguinte interacção:

```
>>> conta_jcp_01 = conta_jovem_com_pin(120, '1234')
>>> conta_jcp_01.levantamento(30, 'abcd')
PIN incorrecto
Tem mais 2 tentativas
>>> conta_jcp_01.levantamento(30, 'ABCD')
PIN incorrecto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
90
>>> conta_jcp_01.altera_codigo('abcd')
PIN incorrecto
Tem mais 2 tentativas
>>> conta_jcp_01.altera_codigo('1234')
Introduza o novo PIN
```

```
--> abcd
Para verificação
Volte a introduzir o novo PIN
--> abcd
PIN alterado
>>> conta_jcp_01.levantamento(30, 'abcd')
60
>>> conta_jcp_01.levantamento(30, '1234')
PIN incorrecto
Tem mais 2 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
PIN incorrecto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
Conta bloqueada
```

11.3 Objectos em Python

O Python é uma linguagem baseada em objectos. Embora a nossa utilização do Python tenha fundamentalmente correspondido à utilização da sua faceta imperativa, na realidade, todos os tipos de informação embutidos em Python correspondem a classes. Recorrendo à função `type`, introduzida na página 286, podemos obter a seguinte interacção que revela que os tipos embutidos com que temos trabalhado não são mais do que objectos pertencentes a determinadas classes:

```
>>> type(2)
<class 'int'>
>>> type(True)
<class 'bool'>
>>> type(3.5)
<class 'float'>
>>> type('abc')
<class 'str'>
>>> type([1, 2, 3])
<class 'list'>
```

Recordemos a função que lê uma linha de um ficheiro, `<fich>.readline()`, apresentada na Secção 8.2. Dissemos na altura que o nome desta função podia ser tratado como um nome composto. O que na realidade se passa é que ao abrirmos um ficheiro, estamos a criar um objecto, objecto esse em que um dos métodos tem o nome de `readline`.

11.4 Polimorfismo

Já vimos que muitas das operações do Python são operações sobrecarregadas, ou seja, aplicam-se a vários tipos de informação. Por exemplo, a operação `+` pode ser aplicada a inteiros, a reais, a tuplos, a cadeias de caracteres, a listas e a dicionários.

Diz-se que uma operação é *polimórfica*, ou que apresenta a propriedade do *polimorfismo*³, quando é possível definir funções diferentes que usam a mesma operação para lidar com tipos de dados diferentes. Consideremos a operação de adição, representada em Python por “`+`”. Esta operação aplica-se tanto a inteiros como a reais. Recorde-se da página 43, que dissemos que internamente ao Python existiam duas operações, que representámos por `+Z` e `+R`, que, respectivamente, são invocadas quando os argumentos da operação `+` são números inteiros ou são números reais. Esta capacidade do Python de associar a mesma representação externa de uma operação, `+`, a diferentes operações internas corresponde a uma faceta do polimorfismo⁴. Esta faceta do polimorfismo dá origem a operações sobrecarregadas.

Após definirmos o tipo correspondente a números complexos, no Capítulo 10, estaremos certamente interessados em escrever funções que executem operações aritméticas sobre complexos. Por exemplo, a soma de complexos pode ser definida por:

```
def soma_compl(c1, c2):
    r = c1.p_real() + c2.p_real()
    i = c1.p_imag() + c2.p_imag()
    return compl(r, i)
```

³A palavra “polimorfismo” é derivada do grego e significa apresentar múltiplas formas.

⁴O tipo de polimorfismo que definimos nesta secção corresponde ao conceito inglês de “subtype polymorphism”.

com a qual podemos gerar a interacção:

```
>>> c1 = compl(9, 6)
>>> c2 = compl(7, 6)
>>> c3 = soma_compl(c1, c2)
>>> c3.escreve()
16 + 12 i
```

No entanto, quando trabalhamos com números complexos em matemática usamos o símbolo da operação de adição, $+$, para representar a soma de números complexos. Através da propriedade do polimorfismo, o Python permite especificar que a operação $+$ também pode ser aplicada a números complexos e instruir o computador em como aplicar a operação “ $+$ ” a números complexos.

Sabemos que em Python, todos os tipos embutidos correspondem a classes. Um dos métodos que existe numa classe tem o nome de `__add__`, recebendo dois argumentos, `self` e outro elemento do mesmo tipo. A sua utilização é semelhante à da função `compl_iguais` que apresentámos na página 284, como o mostra a seguinte interacção:

```
>>> a = 5
>>> a.__add__(2)
7
```

A invocação deste método pode ser feita através da representação externa “ $+$ ”, ou seja, sempre que o Python encontra a operação $+$, invoca automaticamente o método `__add__`, aplicado às instâncias envolvidas. Assim, se associado à classe `compl`, definirmos o método

```
def __add__(self, outro):
    r = self.p_real() + outro.p_real()
    i = self.p_imag() + outro.p_imag()
    return compl(r, i)
```

podemos originar a interacção

```
>>> c1 = compl(2, 4)
>>> c2 = compl(5, 10)
```

```
>>> c3 = c1 + c2
>>> c3.escreve()
7 + 14 i
```

Ou seja, usando o polimorfismo, criámos uma nova forma da operação `+` que sabe somar complexos. A “interface” da operação para somar complexos passa a ser o operador `+`, sendo este operador transformado na operação de somar complexos quando os seus argumentos são números complexos.

De um modo análogo, existem métodos embutidos, com os nomes `__sub__`, `__mul__` e `__truediv__` que estão associados às representações das operações `-`, `*` e `/`. O método `__eq__` está associado à operação `==`. Existe também um método, `__repr__` que transforma a representação interna de uma instância da classe numa cadeia de caracteres que corresponde à sua representação externa. Esta representação externa é usada directamente pela função `print`.

Sabendo como definir em Python operações aritméticas e a representação externa, podemos modificar a classe `compl` com os seguintes métodos. Repare-se que com o método `__eq__` deixamos de precisar do método `compl_iguais` e que com o método `__repr__` deixamos de precisar do método `escreve` para gerar a representação externa de complexos⁵. Em relação à classe `compl`, definimos como operações de alto nível, funções que adicionam, subtraem, multiplicam e dividem complexos.

```
class compl:

    def __init__(self, real, imag):
        if isinstance(real, (int, float)) and \
           isinstance(imag, (int, float)):
            self.r = real
            self.i = imag
        else:
            raise ValueError ('complexo: argumento errado')

    def p_real(self):
        return self.r
```

⁵No método `__repr__` utilizamos a função embutida `str`, a qual foi apresentada na Tabela 4.3, que recebe uma constante de qualquer tipo e tem como valor a cadeia de caracteres correspondente a essa constante.

```
def p_imag(self):
    return self.i

def compl_zero(self):
    return self.r == 0 and self.i == 0

def __eq__(self, outro):
    return self.r == outro.p_real() and \
           self.i == outro.p_imag()

def __add__(self, outro):
    r = self.p_real() + outro.p_real()
    i = self.p_imag() + outro.p_imag()
    return compl(r, i)

def __sub__(self, outro):
    r = self.p_real() - outro.p_real()
    i = self.p_imag() - outro.p_imag()
    return compl(r, i)

def __mul__(self, outro):
    r = self.p_real() * outro.p_real() - \
        self.p_imag() * outro.p_imag()
    i = self.p_real() * outro.p_imag() + \
        self.p_imag() * outro.p_real()
    return compl(r, i)

def __truediv__(self, outro):
    try:
        den = outro.p_real() * outro.p_real() + \
              outro.p_imag() * outro.p_imag()
        r = (self.p_real() * outro.p_real() + \
              self.p_imag() * outro.p_imag()) / den
        i = (self.p_real() * outro.p_imag() - \
              self.p_imag() * outro.p_real()) / den
        return compl(r, i)
    except ZeroDivisionError:
        raise ValueError("Divisão por zero")
```

```

        self.p_imag() * outro.p_imag()) / den
        i = (self.p_imag() * outro.p_real() - \
              self.p_real() * outro.p_imag()) / den
        return compl(r, i)
    except ZeroDivisionError:
        print('complexo: divisão por zero')

def __repr__(self):
    if self.p_imag() >= 0:
        return str(self.p_real()) + '+' + \
               str(self.p_imag()) + 'i'
    else:
        return str(self.p_real()) + '-' + \
               str(abs(self.p_imag())) + 'i'

```

Podendo agora gerar a seguinte interacção:

```

>>> c1 = compl(2, 5)
>>> c1
2+5i
>>> c2 = compl(-9, -7)
>>> c2
-9-7i
>>> c1 + c2
-7-2i
>>> c1 * c2
17-59i
>>> c3 = compl(0, 0)
>>> c1 / c3
complexo: divisão por zero
>>> c1 == c2
False
>>> c4 = compl(2, 5)
>>> c1 == c4
True

```

11.5 Notas finais

Neste capítulo apresentámos um estilo de programação, conhecido como programação com objectos, que é centrado em objectos, entidades que possuem um estado interno e reagem a mensagens. Os objectos correspondem a entidades, tais como contas bancárias, livros, estudantes, etc. O conceito de objecto agrupa as funções que manipulam dados (os métodos) com os dados que representam o estado do objecto.

O conceito de objecto foi introduzido com a linguagem SIMULA⁶, foi vulgarizado em 1984 pelo sistema operativo do Macintosh, foi adoptado pelo sistema operativo Windows, uma década mais tarde, e está hoje na base de várias linguagens de programação, por exemplo, o C++, o CLOS, o Java e o Python.

11.6 Exercícios

- Defina uma classe em Python, chamada `estacionamento`, que simula o funcionamento de um parque de estacionamento. A classe `estacionamento` recebe um inteiro que determina a lotação do parque e devolve um objecto com os seguintes métodos: `entra()`, corresponde à entrada de um carro; `sai()`, corresponde à saída de um carro; `lugares()` indica o número de lugares livres no estacionamento. Por exemplo,

```
>>> ist = estacionamento(20)
>>> ist.lugares()
20
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.entra()
>>> ist.sai()
>>> ist.lugares()
17
```

- Defina uma classe que corresponde a uma urna de uma votação. A sua classe deve receber a lista dos possíveis candidatos e manter como estado

⁶[Dahl e Nygaard, 1967].

interno o número de votos em cada candidato. Esta classe pode receber um voto num dos possíveis candidatos, aumentando o número de votos nesse candidato em um. Deve também permitir apresentar os resultados da votação.

3. Considere a função de Ackermann apresentada nos exercícios do Capítulo 7. Como pode verificar, a sua função calcula várias vezes o mesmo valor. Para evitar este problema, podemos definir uma classe, `mem_A`, cujo estado interno contém informação sobre os valores de `A` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `A` para os inteiros que são seus argumentos. Por exemplo,

```
>>> A = mem_A()  
>>> A.val(2, 2)  
7
```

Defina a classe `mem_A`.

Capítulo 12

O desenvolvimento de programas

*'First, the fish must be caught,'
That is easy: a baby, I think could have caught it.
'Next, the fish must be bought,'
That is easy: a penny, I think, would have bought it.
'Now, cook me the fish!'
That is easy, and will not take more than a minute.
'Let it lie in a dish!'
That is easy, because it already is in it.*

Lewis Carroll, *Through the Looking Glass*

A finalidade deste capítulo é a apresentação sumária das várias fases por que passa o desenvolvimento de um programa, fornecendo uma visão global da actividade de programação.

A expressão “desenvolvimento de um programa” é frequentemente considerada como sinónimo de programação, ou de codificação, isto é, a escrita de instruções utilizando uma linguagem de programação. Contudo, bastante trabalho preparatório deve anteceder a programação de qualquer solução potencial para o problema que se pretende resolver. Este trabalho preparatório é constituído por fases como a definição exacta do que se pretende fazer, removendo ambiguidades e incertezas que possam estar contidas nos objectivos a atingir, a decisão do processo a utilizar para a solução do problema e o delineamento da solução utilizando uma linguagem adequada. Se este trabalho preparatório for bem feito, a

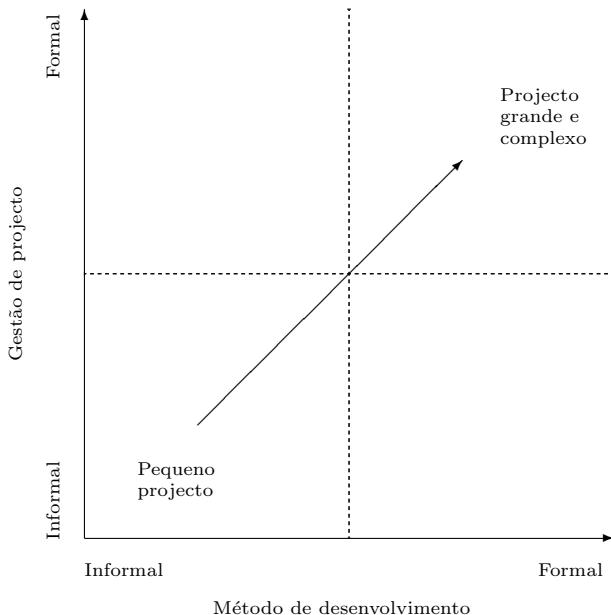


Figura 12.1: O problema da dimensão do programa.

fase de programação, que parece a mais importante para muitas pessoas, torna-se relativamente fácil e pouco criativa. Tal como se despendeu muito trabalho antes de começar a programar, muito trabalho terá de ser feito desde a fase de programação até o programa estar completo. Terão de se detectar e corrigir todos os erros, testar exaustivamente o programa e consolidar a documentação. Mesmo depois de o programa estar completamente terminado, existe trabalho a fazer relativamente à manutenção do programa.

O ponto fundamental nesta discussão é que o desenvolvimento de um programa é uma actividade complexa, constituída por várias fases individualizadas, sendo todas elas importantes, e cada uma delas contribuindo para a solução do problema. É evidente que, quanto mais complexo for o programa, mais complicada é a actividade do seu desenvolvimento.

Desde os anos 60 que a comunidade informática tem dedicado um grande esforço à caracterização e regulamentação da actividade de desenvolvimento de programas complexos. Este trabalho deu origem a uma subdisciplina da informática, a *engenharia da programação*¹ que estuda as metodologias para o desenvolvi-

¹Do inglês “software engineering”.

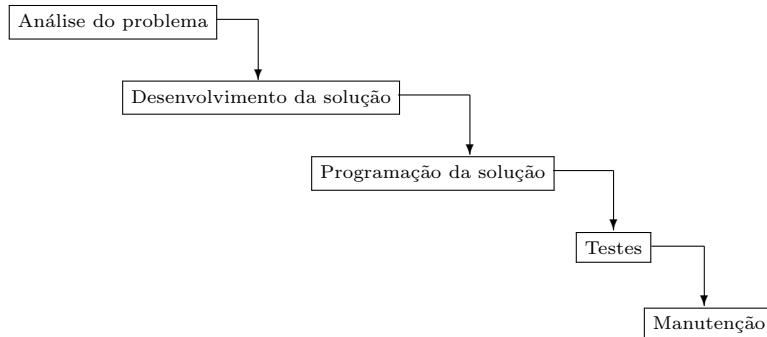


Figura 12.2: Modelo de cascata.

mento de programas. A finalidade da engenharia da programação é a de criar metodologias para desenvolver programas de qualidade a baixo custo. Nesta disciplina faz-se a distinção clara entre o conceito de *programa* (conjunto de instruções escritas numa linguagem de programação) e sistema computacional, entendido como “*software*”. A sociedade americana IEEE, “Institute of Electrical and Electronics Engineers”, definiu o termo *sistema computacional* como sendo “uma colecção de programas, funções, regras, documentação e dados associados” para transmitir de modo inequívoco que a actividade de programação não se limita à produção de código para ser executado por um computador, mas que inclui toda a documentação e dados associados a esse código.

Existem diferentes metodologias para diferentes tipos de problemas a resolver e diferentes dimensões do programa final. Um dos problemas associados ao desenvolvimento de programas é a dimensão do produto final. Os métodos utilizados para o desenvolvimento de pequenos programas não podem ser adaptados directamente a programas grandes e complexos (Figura 12.1). Quanto maior for a complexidade do programa e a dimensão da equipa associada ao seu desenvolvimento, maior é o nível de formalização necessário para os métodos de desenvolvimento e para a gestão do projecto associado ao seu desenvolvimento.

Uma das metodologias mais usadas no desenvolvimento de programas é baseada no modelo da cascata². De acordo com o *modelo da cascata*, o desenvolvimento de um programa passa por cinco fases distintas que, embora sejam executadas sequencialmente, estão intimamente interligadas (Figura 12.2): a análise do pro-

²Do inglês “waterfall method” (ver [Boehm, 1981]).

blema, o desenvolvimento da solução, a programação da solução, os testes e a manutenção. Em todas estas fases é produzida documentação que descreve as decisões tomadas e que serve de apoio ao desenvolvimento das fases subsequentes.

Durante o desenvolvimento de um programa é bom ter sempre em mente a chamada *Lei de Murphy*:

1. Tudo é mais difícil do que parece.
2. Tudo demora mais tempo do que pensamos.
3. Se algo puder correr mal, irá correr mal, no pior dos momentos possíveis.

12.1 A análise do problema

Um programa é desenvolvido para satisfazer uma necessidade reconhecida por um utilizador ou por um conjunto de utilizadores. Neste capítulo, o utilizador (ou utilizadores) é designado por “cliente”. As necessidades do cliente são, de um modo geral, apresentadas com lacunas, imperfeições, ambiguidades e até, por vezes, contradições.

Durante a fase da análise do problema, o programador (ou, de um modo mais preciso, o analista – a pessoa que analisa o problema) estuda o problema, juntamente com o cliente, para determinar exactamente *o que tem de ser feito*, mesmo antes de pensar *como os objectivos vão ser atingidos*. Esta fase parece ser de tal maneira óbvia que muitos programadores a ignoram completamente, não perdendo tempo a tentar compreender o problema que se propõem resolver. Como resultado da não consideração desta fase, começam a desenvolver um programa mal concebido, o qual pode representar uma solução incorrecta do problema.

Esta fase processa-se antes de se começar a pensar na solução do problema, mais concretamente, em como resolver o problema. Pretende-se determinar claramente quais as especificações do problema e saber exactamente quais os objectivos a atingir. Esta fase envolve duas entidades com características diferentes, o programador e o cliente. De um modo geral, o programador não tem conhecimento sobre o domínio de aplicação do cliente e o cliente não domina os

aspectos técnicos associados à programação. Este aspecto cria um problema de comunicação que tem de ser resolvido durante a análise do problema.

O trabalho desenvolvido nesta fase inclui um conjunto de interacções entre o programador e o cliente, na qual, progressivamente, ambas as partes aumentam a sua compreensão sobre o que tem que ser feito. Estas interacções são baseadas num documento evolutivo que vai descrevendo, cada vez de modo mais detalhado, quais as características do trabalho a desenvolver.

O resultado desta fase é a criação de um documento – o documento de *análise dos requisitos* – especificando claramente, do ponto de vista informático (mas de modo que sejam perfeitamente entendidos pelo cliente), o que faz o programa, estudo das alternativas para o desenvolvimento do programa e riscos envolvidos no desenvolvimento. Para além deste documento, são também resultados da fase de análise do problema um planeamento pormenorizado para o desenvolvimento com o faseamento das fases seguintes, os custos estimados, etc.

O documento de *análise de requisitos* (conhecido frequentemente por SRS³) embora não especifique *como* é que o programa vai abordar a solução do problema, tem duas finalidades importantes. Por um lado, para o cliente, serve de garantia escrita do que vai ser feito. Por outro lado, para o programador, serve como definição dos objectivos a atingir.

12.2 O desenvolvimento da solução

Uma vez sabido *o que* deve ser feito, durante o desenvolvimento da solução determina-se *como* deve ser feito. Esta é uma fase predominantemente criativa, em que se desenvolve um algoritmo que constitui a solução do problema a resolver.

O desenvolvimento deste algoritmo deve ser feito sem ligação a uma linguagem de programação particular, pensando apenas em termos das operações e dos tipos de informação que vão ser necessários. Normalmente, os algoritmos são desenvolvidos utilizando linguagens semelhantes às linguagens de programação nas quais se admitem certas descrições em língua natural. Pretende-se, deste modo, descrever rigorosamente como vai ser resolvido o problema sem se entrar, no entanto, nos pormenores inerentes a uma linguagem de programação.

³Do inglês, “System Requirements Specifications”.

A ideia chave a utilizar durante esta fase é a *abstracção*. Em qualquer instante deve separar-se o problema que está a ser abordado dos pormenores irrelevantes para a sua solução.

As metodologias a seguir durante esta fase são o *desenvolvimento do topo para a base*⁴ e a *refinação por passos*⁵, as quais têm como finalidade a diminuição da complexidade do problema a resolver, originando também algoritmos mais legíveis e fáceis de compreender. Segundo estas metodologias, o primeiro passo para a solução de um problema consiste em identificar os principais subproblemas que constituem o problema a resolver, e em determinar qual a relação entre esses subproblemas. Depois de concluída esta primeira fase, desenvolve-se uma primeira aproximação do algoritmo, aproximação essa que é definida em termos dos subproblemas identificados e das relações entre eles. Depois deve repetir-se sucessivamente este processo para cada um dos subproblemas. Quando se encontrar um subproblema cuja solução é trivial, deve-se então escrever o algoritmo para esse subproblema.

Esta metodologia para o desenvolvimento de um algoritmo tem duas vantagens: o controle da complexidade e a modularidade da solução. Quanto ao *controle da complexidade*, devemos notar que em cada instante durante o desenvolvimento do algoritmo estamos a tentar resolver um subproblema único, sem nos preocuparmos com os pormenores da solução, mas apenas com os seus aspectos fundamentais. Quanto à *modularidade da solução*, o algoritmo resultante será constituído por módulos, cada módulo correspondente a um subproblema cuja função é perfeitamente definida.

Seguindo esta metodologia, obtém-se um algoritmo que é fácil de compreender, de modificar e de corrigir. O algoritmo é fácil de compreender, pois é expresso em termos das divisões naturais do problema a resolver. Se pretendermos alterar qualquer aspecto do algoritmo, podemos determinar facilmente qual o módulo do algoritmo (ou seja, o subproblema) que é afectado pela alteração, e só teremos de considerar esse módulo durante o processo de modificação. Os algoritmos tornam-se assim mais fáceis de modificar. Analogamente, se detectarmos um erro na concepção do algoritmo, apenas teremos de considerar o subproblema em que o erro foi detectado.

O resultado desta fase é um documento, o *documento de concepção* (conhecido

⁴Do inglês, “top down design”.

⁵Do inglês, “stepwise refinement”.

frequentemente por SDD⁶, em que se descreve pormenorizadamente a solução do problema, as decisões que foram tomadas para o seu desenvolvimento e as decisões que têm de ser adiadas para a fase da programação da solução. O documento de concepção está normalmente organizado em dois subdocumentos, a concepção global e a concepção pormenorizada. O documento de *concepção global* identifica os módulos principais do sistema, as suas especificações de alto nível, o modo de interacção entre os módulos, os dados que recebem e os resultados que produzem. O documento de *concepção pormenorizada* especifica o algoritmo utilizado por cada um dos módulos e os tipos de informação que estes manipulam.

A fase de desenvolvimento da solução termina com uma verificação formal dos documentos produzidos.

12.3 A programação da solução

The sooner you start coding your program the longer it is going to take.
[Ledgard, 1975]

Antes de iniciarmos esta secção, será importante ponderarmos a citação de Henri Ledgard: “*Quanto mais cedo começares a escrever o teu programa mais tempo demorarás*”. Com esta frase, Ledgard pretende dizer que o desenvolvimento de um programa sem um período prévio de meditação e planeamento, em relação ao que deve ser feito e como fazê-lo, leva a situações caóticas, que para serem corrigidas requerem mais tempo do que o tempo despendido num planeamento cuidado do algoritmo.

Só depois de termos definido claramente o problema a resolver e de termos desenvolvido cuidadosamente um algoritmo para a sua solução, poderemos iniciar a fase de programação, ou seja, a escrita do algoritmo desenvolvido, recorrendo a uma linguagem de programação.

O primeiro problema a resolver, nesta fase, será a escolha da linguagem de programação a utilizar. A escolha da linguagem de programação é ditada por duas considerações fundamentais:

⁶Do inglês “Software Design Description”.

1. *As linguagens existentes (ou potencialmente existentes) no computador que vai ser utilizado.* De facto, esta é uma limitação essencial. Apenas poderemos utilizar as linguagens que se encontram à nossa disposição.
2. *A natureza do problema a resolver.* Algumas linguagens são mais adequadas à resolução de problemas envolvendo fundamentalmente cálculos numéricos, ao passo que outras linguagens são mais adequadas à resolução de problemas envolvendo manipulações simbólicas. Assim, tendo a possibilidade de escolha entre várias linguagens, não fará sentido, por exemplo, a utilização de uma linguagem de carácter numérico para a solução de um problema de carácter simbólico.

Uma vez decidida qual a linguagem de programação a utilizar, e tendo já uma descrição do algoritmo, a geração das instruções do programa é relativamente fácil. O programador terá de decidir como representar os tipos de informação necessários e escrever as respectivas operações. Em seguida, traduzirá as instruções do seu algoritmo para instruções escritas na linguagem de programação a utilizar. O objectivo desta fase é a concretização dos documentos de concepção.

O resultado desta fase é um programa escrito na linguagem escolhida, com comentários que descrevem o funcionamento das funções e os tipos de informação utilizados, bem como um documento que complementa a descrição do algoritmo produzido na fase anterior e que descreve as decisões tomadas quanto à representação dos tipos de informação.

Juntamente com estes documentos são apresentados os resultados dos testes que foram efectuados pelo programador para cada um dos módulos que compõem o sistema (os chamados *testes de módulo* ou *testes unitários*). Paralelamente, nesta fase desenvolve-se a documentação de utilização do programa.

12.3.1 A depuração

Anyone who believes his or her program will run correctly the first time is either a fool, an optimist, or a novice programmer.
[Schneider et al., 1978]

Na fase de depuração (do verbo depurar, tornar puro, em inglês, conhecida

por “*debugging*”⁷) o programador detecta, localiza e corrige os erros existentes no programa desenvolvido. Estes erros fazem com que o programa produza resultados incorrectos ou não produza quaisquer resultados. A fase de depuração pode ser a fase mais demorada no desenvolvimento de um programa.

Existem muitas razões para justificar a grande quantidade de tempo e esforço despendidos normalmente na fase de depuração, mas duas são de importância primordial. Em primeiro lugar, a facilidade de detecção de erros num programa está directamente relacionada com a clareza da estrutura do programa. A utilização de abstracção diminui a complexidade de um programa, facilitando a sua depuração. Em segundo lugar, as técnicas de depuração não são normalmente ensinadas do mesmo modo sistemático que as técnicas de desenvolvimento de programas. A fase de depuração é, em grande parte dos casos, seguida sem método, fazendo tentativas cegas, tentando alguma coisa, qualquer coisa, pois não se sabe como deve ser abordada.

Os erros de um programa podem ser de dois tipos distintos, erros de natureza sintáctica e erros de natureza semântica.

A depuração sintáctica

Os erros de *natureza sintáctica* são os erros mais comuns em programação, e são os mais fáceis de localizar e de corrigir. Um erro sintáctico resulta da não conformidade de um constituinte do programa com as regras sintácticas da linguagem de programação. Os erros sintácticos podem ser causados por erros de ortografia ao escrevermos o programa ou por um lapso na representação da estrutura de uma instrução.

Os erros de natureza sintáctica são detectados pelo processador da linguagem, o qual produz mensagens de erro, indicando qual a instrução mais provável em que o erro se encontra e, normalmente, qual o tipo de erro verificado. A tendência actual em linguagens de programação é produzir mensagens de erro que auxiliem, tanto quanto possível, o programador.

A correção dos erros sintáticos normalmente não origina grandes modificações no programa.

A fase de depuração sintáctica termina quando o processador da linguagem não

⁷Ver a nota de rodapé na página 27.

encontra quaisquer erros de natureza sintáctica no programa. O programador inexperiente tende a ficar eufórico quando isto acontece, não tendo a noção de que a verdadeira fase de depuração vai então começar.

A depuração semântica

Os *erros semânticos* resultam do facto de o programa, sintacticamente correcto, ter um significado para o computador que é diferente do significado que o programador desejava que ele tivesse. Os erros de natureza semântica podem causar a interrupção da execução do programa, ciclos infinitos de execução, a produção de resultados errados, etc.

Quando, durante a fase de depuração semântica, se descobre a existência de um erro cuja localização ou origem não é facilmente detectável, o programador deverá recorrer metodicamente às seguintes técnicas de depuração (ou a uma combinação delas):

1. Utilização de *programas destinados à depuração*. Certos processadores de linguagens fornecem programas especiais que permitem fazer o rastreio⁸ automático do programa, inspecionar o estado do programa quando o erro se verificou, alterar valores de variáveis, modificar instruções, etc.
2. Utilização da técnica da *depuração da base para o topo*⁹. Utilizando esta técnica, testam-se, em primeiro lugar, os módulos (por módulo entenda-se uma função correspondente a um subproblema) que estão ao nível mais baixo, isto é, que não são decomponíveis em subproblemas, e só quando um nível está completamente testado se passa ao nível imediatamente acima. Assim, quando se aborda a depuração de um nível tem-se a garantia de que não há erros em nenhum dos níveis que ele utiliza e, portanto, que os erros que surgirem dependem *apenas* das instruções nesse nível.

Depois de detectado um erro de natureza semântica, terá de se proceder à sua correcção. A correcção de um erro de natureza semântica poderá variar desde casos extremamente simples a casos que podem levar a uma revisão completa

⁸Do dicionário da Porto Editora: rastear *v.t.* seguir o rasto de; rastreio *s.m.* acto de rastrear; rastro *s.m.* vestígio que alguém, algum animal ou alguma coisa deixou no solo ou no ar, quando passou.

⁹Do inglês, “bottom-up debugging”.

da fase de desenvolvimento da solução e, consequentemente, à criação de um novo programa.

12.3.2 A finalização da documentação

O desenvolvimento da documentação do programa deve começar simultaneamente com a formulação do problema (fase 1) e continuar à medida que se desenvolve a solução (fase 2) e se escreve o programa (fase 3). Nesta secção, vamos descrever o conteúdo da documentação que deve estar associada a um programa no instante em que ele é entregue à equipa de testes (ver Secção 12.4). A documentação de um programa é de dois tipos: a documentação destinada aos utilizadores do programa, chamada documentação de utilização, e a documentação destinada às pessoas que irão fazer a manutenção do programa, a documentação técnica.

A documentação de utilização

A *documentação de utilização* tem a finalidade de fornecer ao utilizador a informação necessária para a correcta utilização do programa. Esta documentação inclui normalmente o seguinte:

1. *Uma descrição do que o programa faz.* Nesta descrição deve estar incluída a área geral de aplicação do programa e uma descrição precisa do seu comportamento. Esta descrição deve ser bastante semelhante à descrição desenvolvida durante a fase de análise do problema.
2. *Uma descrição do processo de utilização do programa.* Deve ser explicado claramente ao utilizador do programa o que ele deve fazer, de modo a poder utilizar o programa.
3. *Uma descrição da informação necessária ao bom funcionamento do programa.* Uma vez o programa em execução, vai ser necessário fornecer-lhe certa informação para ser manipulada. A forma dessa informação é descrita nesta parte da documentação. Esta descrição pode incluir a forma em que os ficheiros com dados devem ser fornecidos ao computador (se for caso disso), o tipo de comandos que o programa espera receber durante o seu funcionamento, etc.

4. *Uma descrição da informação produzida pelo programa, incluindo por vezes a explicação de mensagens de erro.*
5. *Uma descrição, em termos não técnicos, das limitações do programa.*

A documentação técnica

A *documentação técnica* fornece ao programador que irá modificar o programa a informação necessária para a compreensão do programa. A documentação técnica é constituída por duas partes, a documentação externa e a documentação interna.

A parte da documentação técnica que constitui a *documentação externa* descreve o algoritmo desenvolvido na fase 2 (desenvolvimento da solução), a estrutura do programa, as principais funções que o constituem e a interligação entre elas. Deve ainda descrever os tipos de informação utilizados no algoritmo e a justificação para a escolha de tais tipos.

A *documentação interna* é constituída pelos comentários do programa. Os *comentários* são linhas ou anotações que se inserem num programa e que descrevem, em língua natural, o significado de cada uma das partes do programa. Cada linguagem de programação tem a sua notação própria para a criação de comentários; por exemplo, em Python um comentário é tudo o que aparece numa linha após o carácter `#`. Para exemplificar a utilização de comentários em Python, apresenta-se de seguida a função `factorial`, juntamente com um comentário:

```
# Calcula o factorial de um número.  
# Não testa se o número é negativo  
  
def factorial(n):  
    fact = 1  
    for i in range(n, 0, -1):  
        fact = fact * i  
    return fact
```

Os comentários podem auxiliar tremendamente a compreensão de um programa, e por isso a sua colocação deve ser criteriosamente estudada. Os comentários

devem identificar secções do programa e devem explicar claramente o objectivo dessas secções e o funcionamento do algoritmo respectivo. É importante não sobrecarregar o programa com comentários, pois isso dificulta a sua leitura. Certos programadores inexperientes tendem por vezes a colocar um comentário antes de cada instrução, explicando o que é que ela faz. Os comentários devem ser escritos no programa, à medida que o programa vai sendo desenvolvido, e devem reflectir os pensamentos do programador ao longo do desenvolvimento do programa. Comentários escritos depois de o programa terminado tendem a ser superficiais e inadequados.

Uma boa documentação é essencial para a utilização e a manutenção de um programa. Sem a documentação para o utilizador, um programa, por excepcional que seja, não tem utilidade, pois ninguém o sabe usar. Por outro lado, uma boa documentação técnica é fundamental para a manutenção de um programa. Podemos decidir modificar as suas características ou desejar corrigir um erro que é descoberto muito depois do desenvolvimento do programa ter terminado. Para grandes programas, estas modificações são virtualmente impossíveis sem uma boa documentação técnica.

Finalmente, uma boa documentação pode ter fins didácticos. Ao tentarmos desenvolver um programa para uma dada aplicação, podemos aprender bastante ao estudarmos a documentação de um programa semelhante.

12.4 A fase de testes

Depois de o processo de depuração semântica aparentemente terminado, isto é, depois de o programa ser executado e produzir resultados correctos, poderá ser posto em causa se o programa resolve o problema para que foi proposto para todos os valores possíveis dos dados. Para garantir a resposta afirmativa a esta questão, o programa é entregue a uma equipa de testes, a qual deverá voltar a verificar os testes de cada um dos módulos executados na fase anterior e, simultaneamente, verificar se todos os módulos em conjunto correspondem à solução acordada com o cliente. A equipa de testes deverá criar uma série de casos de teste para o sistema global (tal como foi descrito no documento de concepção global) e testar o bom funcionamento do programa, para todos estes casos.

Os *casos de teste* deverão ser escolhidos criteriosamente, de modo a testarem todos os caminhos, ou rastos, possíveis através do algoritmo. Por exemplo, suponhamos que a seguinte função recebe três números correspondentes aos comprimentos dos lados de um triângulo e decide se o triângulo é equilátero, isósceles ou escaleno:

```
def classifica(l1, l2, l3):
    if l1 == l2 == l3:
        return 'Equilátero'
    elif (l1 == l2) or (l1 == l3) or (l2 == l3):
        return 'Isósceles'
    else:
        return 'Escaleno'
```

Embora esta função esteja conceptualmente correcta, existem muitas situações que esta não verifica, nomeadamente, se $l1$, $l2$ e $l3$ correspondem aos lados de um triângulo. Para isso terão de verificar as seguintes condições:

1. As variáveis $l1$, $l2$ e $l3$ têm valores numéricos.
2. Nenhuma das variáveis $l1$, $l2$ e $l3$ é negativa.
3. Nenhuma das variáveis $l1$, $l2$ e $l3$ é nula.
4. A soma de quaisquer duas destas variáveis é maior do que a terceira (num triângulo, qualquer lado é menor do que a soma dos outros dois).

Para além destas condições, os casos de teste deverão incluir valores que testam triângulos isósceles, equiláteros e escalenos.

Devemos notar que, para programas complexos, é impossível testar completamente o programa para todas as combinações de dados e portanto, embora estes programas sejam testados de um modo sistemático e criterioso, existe sempre a possibilidade da existência de erros não detectados pelo programador. Como disse Edsger Dijkstra (1930–2002), uma das figuras mais influentes do Século XX no que respeita a programação estruturada, o processo de testar um programa pode ser utilizado para mostrar a presença de erros, mas nunca para mostrar a sua ausência! (“*Program testing can be used to show the presence of bugs, but never to show their absence!*” [Dahl et al., 1972], página 6).

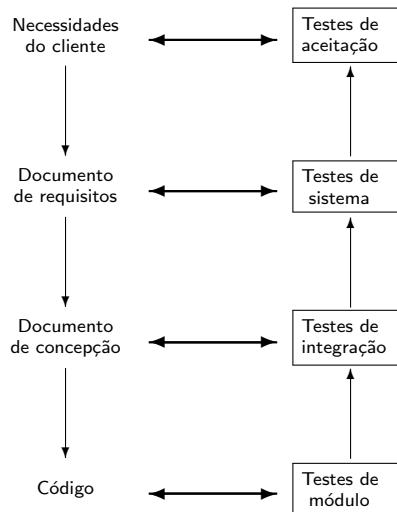


Figura 12.3: Níveis de testes.

Em resumo, os testes de um programa são efectuados a vários níveis (Figura 12.3). Os *testes de módulo* são efectuados pelo programador durante a fase da programação da solução. Os *testes de integração* são efectuados pela equipa de testes, tendo em atenção o documento de concepção. Após a execução, com sucesso, dos testes de integração, a equipa de testes deverá também verificar se o sistema está de acordo com o documento dos requisitos, efectuando *testes de sistema*. Finalmente, após a entrega, o cliente efectua *testes de aceitação* para verificar se o sistema está de acordo com as suas necessidades.

Existem métodos para demonstrar formalmente a correcção semântica de um programa, discutidos, por exemplo, em [Dijkstra, 1976], [Hoare, 1972] e [Wirth, 1973], mas a sua aplicabilidade ainda se limita a programas simples e pequenos.

12.5 A manutenção

Esta fase decorre depois de o programa ter sido considerado terminado, e tem duas facetas distintas. Por um lado, consiste na verificação constante da possibilidade de alterações nas especificações do problema, e, no caso de alteração de especificações, na alteração correspondente do programa. Por outro lado, consiste na correcção dos eventuais erros descobertos durante o funcionamento

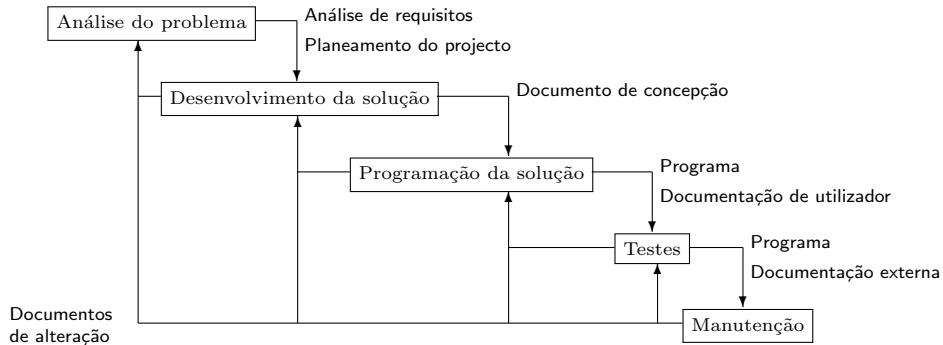


Figura 12.4: Modelo de cascata e documentos associados.

do programa. Em qualquer dos casos, uma alteração no programa obriga a uma correspondente alteração na documentação. Um programa com erros de documentação ou com a documentação desactualizada pode ser pior do que um programa sem documentação, porque encoraja o programador a seguir falsas pistas.

Segundo Frederick P. Brooks, Jr.¹⁰, o custo de manutenção de um programa é superior a 40% do custo total do seu desenvolvimento. Este facto apela a um desenvolvimento cuidado do algoritmo e a uma boa documentação, aspectos que podem diminuir consideravelmente o custo de manutenção.

12.6 Notas finais

Neste capítulo apresentámos de um modo muito sumário as fases por que passa o desenvolvimento de um programa. Seguimos o modelo da cascata, segundo o qual o desenvolvimento de um programa passa por cinco fases sequenciais. Discutimos que a actividade desenvolvida em cada uma destas fases pode levar à detecção de deficiências em qualquer das fases anteriores, que deve então ser repetida.

Esta constatação leva a uma reformulação do modelo apresentado, a qual se indica na Figura 12.4. Nesta figura mostramos que o trabalho desenvolvido em qualquer fase pode levar a um retrocesso para fases anteriores. Indicamos também os principais documentos produzidos em cada uma das fases. Na fase da manutenção podem verificar-se grandes alterações ao programa, o que pode

¹⁰Ver [Brooks, 1975].

levar a um retrocesso para qualquer uma das fases de desenvolvimento.

O assunto que abordámos neste capítulo insere-se no campo da Engenharia Informática a que se chama *Engenharia da Programação*. Informação adicional sobre este assunto poderá ser consultada em [Jalote, 1997] ou [Sommerville, 1996]. A Engenharia da Programação é uma das áreas da Engenharia Informática em que existe mais regulamentação sobre as metodologias a utilizar. Em [Moore, 1998] encontra-se uma perspectiva global das normas existentes nesta disciplina.

Capítulo 13

Estruturas lineares

“The time has come,” the walrus said, “to talk of many things: Of shoes and ships - and sealing wax - of cabbages and kings”

Lewis Carroll, *Alice’s Adventures in Wonderland*

Neste capítulo apresentamos duas estruturas de informação, as pilhas e as filas, que partilham com os tuplos, com as cadeias de caracteres e com as listas a propriedade dos seus elementos estarem organizados numa sequência. Estas estruturas de informação são conhecidas por *estruturas lineares*.

13.1 Pilhas

As *pilhas*¹ são estruturas de informação constituídas por uma sequência de elementos. Os elementos da pilha são retirados pela ordem inversa pela qual foram colocados — os elementos são adicionados ao topo da pilha e são também retirados do topo da pilha. As pilhas correspondem a um comportamento que vulgarmente é designado por LIFO². Numa pilha não podemos aceder nem inspecionar nenhum elemento, para além do elemento no topo da pilha. As pilhas correspondem a pilhas de objectos físicos, por exemplo uma pilha de tabuleiros numa cantina ou uma pilha de livros.

¹Em inglês “stack”.

²Do inglês “Last In First Out”.

O conceito de pilha é muito comum em informática. Por exemplo, a sequência de ambientes locais criados durante a chamada a funções corresponde ao conceito de pilha.

Uma pilha tem um elemento que se distingue de todos os outros: o elemento do topo da pilha. Apenas o elemento do topo da pilha pode ser acedido ou removido. Nenhum elemento pode ser adicionado, removido ou inspeccionado, a não ser com o recurso a operações que colocam no topo da pilha, removem do topo da pilha e inspeccionam o elemento no topo da pilha.

13.1.1 Operações básicas para pilhas

Vamos considerar as operações básicas para o tipo de informação pilha. Podemos considerar as pilhas através de duas ópticas diferentes. Numa óptica *funcional*, as pilhas são entidades imutáveis. Neste caso, as operações básicas constroem pilhas devolvendo a pilha criada e seleccionam componentes de pilhas sem as alterar. Numa óptica *imperativa*, baseada em objectos, as pilhas são entidades mutáveis, existindo certas operações que alteram a pilha de um modo permanente. Para cada um destes casos, iremos considerar cada um dos grupos de operações a definir para o tipo abstracto de informação *pilha*.

Pilhas como entidades imutáveis

As pilhas como entidades imutáveis apresentam as seguintes operações básicas:

1. *Construtores*. Os construtores são operações que constroem pilhas.

Os construtores para o tipo pilha incluem uma operação que gera pilhas a partir do nada, à qual chamaremos *nova_pilha*. Uma operação semelhante deverá existir sempre que se define um *tipo dinâmico*, ou seja, um tipo cujo número de constituintes existentes nos seus elementos pode aumentar ou diminuir com a sua manipulação.

Um outro construtor será a operação que recebe um elemento e uma pilha, e que insere esse elemento na pilha, ou seja, constrói a pilha que resulta da inserção de um novo elemento. O que interessa capturar neste construtor é o facto de apenas podermos inserir elementos no topo da pilha. Tendo isto em atenção, definimos um construtor chamado *empurra*, que recebe como

argumentos uma pilha e um elemento e que produz a pilha resultante da inserção do elemento no topo da pilha.

2. *Selectores*. Os selectores são operações que seleccionam partes de pilhas.

Deveremos ter um selector que indica o elemento no topo da pilha e um selector que retira o elemento no topo da pilha.

Deste modo, a operação *topo* recebe como argumento uma pilha e devolve o elemento no topo da pilha. Se a pilha for vazia, esta operação é indefinida.

A operação *tira* recebe como argumento uma pilha e devolve a pilha sem o elemento que se encontrava no topo da pilha. Se a pilha for vazia, esta operação é indefinida.

3. *Reconhecedores*. Os reconhecedores são operações que identificam tipos de pilhas.

A operação *pilha* recebe como argumento um elemento de um tipo qualquer e decide se este pertence ao tipo pilha.

A operação *pilha_vazia* recebe como argumento uma pilha e decide se esta corresponde à pilha vazia (a pilha gerada por *nova_pilha*).

4. *Testes*. Os testes são operações que relacionam pilhas entre si.

A operação *pilhas_iguais* recebe como argumentos duas pilhas e decide se estas são ou não iguais.

Em resumo, o tipo pilha, como entidade imutável, tem as seguintes operações básicas (estas operações referem o tipo *elemento*, que corresponde ao tipo dos elementos da pilha):

1. *Construtores*:

- *nova_pilha* : $\{\} \mapsto \text{pilha}$

nova_pilha() tem como valor uma pilha sem elementos.

- *empurra* : $\text{pilha} \times \text{elemento} \mapsto \text{pilha}$

empurra(pilha, elm) tem como valor a pilha que resulta de inserir o elemento *elm* no topo da pilha *pilha*.

2. *Selectores*:

— *topo* : *pilha* \mapsto *elemento*

topo(pilha) tem como valor o elemento que se encontra no topo da pilha *pilha*. Se a pilha não contiver elementos, o valor desta operação é indefinido.

— *tira* : *pilha* \mapsto *pilha*

tira(pilha) tem como valor a pilha que resulta de remover o elemento que se encontra no topo da pilha *pilha*. Se a pilha não contiver elementos, o valor desta operação é indefinido.

3. *Reconhecedores*:

— *pilha* : *universal* \mapsto *lógico*

pilha(arg) tem o valor *verdadeiro*, se *arg* é uma pilha, e tem o valor *falso*, em caso contrário.

— *pilha_vazia* : *pilha* \mapsto *lógico*

pilha_vazia(pilha) tem o valor *verdadeiro*, se *pilha* é a pilha vazia, e tem o valor *falso*, em caso contrário.

4. *Testes*:

— *pilhas_iguais* : *pilha* \times *pilha* \mapsto *lógico*

pilhas_iguais(pilha_1, pilha_2) tem o valor *verdadeiro*, se *pilha_1* é igual a *pilha_2*, e tem o valor *falso*, em caso contrário.

Pilhas como entidades mutáveis

As pilhas como entidades mutáveis apresentam as seguintes operações básicas:

1. *Construtores*. Usando entidades mutáveis, os construtores para o tipo *pilha* incluem apenas a operação que gera pilhas a partir do nada, a qual designamos por *nova_pilha*.

2. *Modificadores*. Existem dois modificadores para pilhas como entidades mutáveis:

A operação *empurra* recebe como argumentos uma pilha e um elemento e modifica a pilha de modo a que esta corresponda à pilha resultante de inserir o elemento no topo da pilha.

A operação *tira* recebe como argumento uma pilha e modifica a pilha, retirando-lhe o elemento no topo da pilha. Se a pilha for vazia, esta operação é indefinida.

3. *Selectores*. Existe apenas um selector, a operação *topo*, que recebe como argumento uma pilha e devolve o elemento no topo da pilha. Se a pilha for vazia, esta operação é indefinida.
4. *Reconhecedores*. Teremos os mesmos reconhecedores que no caso das pilhas como entidades imutáveis.
5. *Testes*. Teremos os mesmos testes que no caso das pilhas como entidades imutáveis.

Em resumo, o tipo pilha, como entidade mutável, tem as seguintes operações básicas (estas operações referem o tipo *elemento*, que corresponde ao tipo dos elementos da pilha). Considerámos que os modificadores, para além de alterarem a pilha que é seu elemento, devolvem a pilha alterada. Deste modo, podemos garantir uma axiomatização única para pilhas como entidades mutáveis e como entidades imutáveis, a qual é apresentada na Secção 13.1.2.

1. *Construtores*:

- *nova_pilha* : $\{\}$ \mapsto *pilha*
nova_pilha() tem como valor uma pilha sem elementos.

2. *Modificadores*:

- *empurra* : *pilha* \times *elemento* \mapsto *pilha*
empurra(pilha, elm) modifica a *pilha* de modo a que esta corresponda à pilha resultante de inserir o elemento *elm* no topo da *pilha*. Tem como valor a pilha que resulta de inserir o elemento *elm* no topo da pilha *pilha*.
- *tira* : *pilha* \mapsto *pilha*
tira(pilha) modifica a *pilha*, retirando-lhe o elemento que se encontra no topo da *pilha*. Tem como valor a pilha que resulta de remover o elemento que se encontra no topo da pilha *pilha*. Se a pilha não contiver elementos, o valor desta operação é indefinido.

3. *Selectores*:

3
5
7
====

Figura 13.1: Representação externa para a pilha com os elementos 3 5 7 (em que 3 está no topo da pilha).

— *topo* : *pilha* \mapsto *elemento*

topo(pilha) tem como valor o elemento que se encontra no topo da pilha *pilha*. Se a pilha não contiver elementos, o valor desta operação é indefinido.

4. Reconhecedores:

— *pilha* : *universal* \mapsto *lógico*

pilha(arg) tem o valor *verdadeiro*, se *arg* é uma pilha, e tem o valor *falso*, em caso contrário.

— *pilha_vazia* : *pilha* \mapsto *lógico*

pilha_vazia(pilha) tem o valor *verdadeiro*, se *pilha* é a pilha vazia, e tem o valor *falso*, em caso contrário.

5. Testes:

— *pilhas_iguais* : *pilha* \times *pilha* \mapsto *lógico*

pilhas_iguais(pilha_1, pilha_2) tem o valor *verdadeiro*, se *pilha_1* é igual a *pilha_2*, e tem o valor *falso*, em caso contrário.

Independentemente do tipo de entidades que utilizamos para pilhas, entidades imutáveis ou mutáveis, devemos ainda estabelecer uma representação externa para pilhas. Vamos convencionar que uma pilha será apresentada ao mundo exterior com a sequência dos elementos que a constituem, em linhas separadas, começando com o elemento no topo da pilha e terminando com a cadeia de caracteres “====”. Assim, a pilha com os elementos 3 5 7 (em que 3 está no topo da pilha) será representada como se indica na Figura 13.1. Seguindo a nossa convenção, a pilha vazia será representada por “====”.

13.1.2 Axiomatização

Entre as operações básicas para pilhas devem verificar-se as seguintes relações:

$$\begin{aligned}
 & \text{pilha}(\text{nova_pilha}()) = \text{verdadeiro} \\
 & \text{pilha}(\text{empurra}(p, e)) = \text{verdadeiro} \\
 & \text{pilha}(\text{tira}(p)) = \begin{cases} \text{verdadeiro} & \text{se } p \text{ não for vazia} \\ \perp & \text{em caso contrário} \end{cases} \\
 & \text{pilha_vazia}(\text{nova_pilha}()) = \text{verdadeiro} \\
 & \text{pilha_vazia}(\text{empurra}(p, e)) = \text{falso} \\
 & \text{topo}(\text{empurra}(p, e)) = e \\
 & \text{tira}(\text{empurra}(p, e)) = p \\
 & \text{pilhas_iguais}(p_1, p_2) = \begin{cases} \text{pilha_vazia}(p_1) & \text{se } \text{pilha_vazia}(p_2) \\ \text{pilha_vazia}(p_2) & \text{se } \text{pilha_vazia}(p_1) \\ \text{topo}(p_1) = \text{topo}(p_2) & \text{se } \text{tira}(p_1) = \text{tira}(p_2) \\ \text{falso} & \text{em caso contrário} \end{cases}
 \end{aligned}$$

13.1.3 Representação de pilhas

Representaremos pilhas recorrendo a listas:

1. Uma pilha vazia é representada pela lista vazia.
2. Uma pilha não vazia é representada por uma lista em que o primeiro elemento corresponde ao elemento no topo da pilha, o segundo elemento ao elemento imediatamente a seguir ao elemento no topo da pilha e assim sucessivamente.

Na Figura 13.2 apresentamos a nossa representação interna para pilhas.

13.1.4 Realização das operações básicas

Com base na representação escolhida para as pilhas, apresentamos a realização das operações básicas para duas alternativas, pilhas como funções (dando origem

$$\begin{aligned}\mathfrak{R}[==] &= [] \\ \mathfrak{R} \left[\begin{array}{c} X_1 \\ X_2 \\ \dots \\ X_n \\ == \end{array} \right] &= [\mathfrak{R}[X_1], \mathfrak{R}[X_2], \dots, \mathfrak{R}[X_n]]\end{aligned}$$

Figura 13.2: Representação interna de pilhas.

a um tipo imutável) e pilhas como objectos (dando origem a um tipo mutável).

Pilhas como entidades imutáveis

Antes de apresentar a realização de pilhas, vamos introduzir a instrução `try-except`. Esta instrução tem a finalidade de “apanhar” os erros que são detectados durante a execução de um grupo de instruções (tarefa que também é conhecida como *lidar com exceções* à execução de um grupo de instruções), permitindo ao programador que o seu programa tome medidas excepcionais na presença destes erros.

A sintaxe da instrução `try-except` é definida pela seguinte expressão em notação BNF³:

```

⟨instrução try-except⟩ ::= try: [CR]
                           ⟨instrução composta⟩
                           ⟨excepções⟩*
                           {finally: [CR]
                            ⟨instrução composta⟩}

⟨excepções⟩ ::= except ⟨excepção⟩: [CR]
                           ⟨instrução composta⟩
⟨excepção⟩ ::= ⟨nomes⟩

```

O símbolo não terminal `⟨nomes⟩` foi definido na página 76. A utilização de nomes

³Esta não é uma definição completa da instrução `try-except`.

nesta instrução diz respeito a nomes que correspondem à identificação de vários tipos de erros que podem surgir num programa, alguns dos quais se mostram na Tabela 3.3, apresentada na página 92.

Ao encontrar a instrução:

```
try:  
    ⟨instruçõest⟩  
except⟨excepção1⟩:  
    ⟨instruções1⟩  
:  
except⟨excepçãon⟩:  
    ⟨instruçõesn⟩  
finally:  
    ⟨instruçõesf⟩
```

O Python começa por executar as instruções correspondentes a ⟨instruções_t⟩. Se estas instruções forem executadas sem gerar nenhum erro, a execução da instrução **try-except** está terminada. Se for gerado um erro de execução durante a execução das instruções correspondentes a ⟨instruções_t⟩, em vez de interromper a execução do programa, mostrando o erro ao utilizador, como é o comportamento normal do Python, este vai percorrer as excepções, começando por ⟨excepção₁⟩ ... ⟨excepção_n⟩, por esta ordem. Quando for encontrada a primeira excepção que corresponda ao erro gerado, digamos, ⟨excepção_i⟩, o Python executa as instruções associadas a esta excepção, ⟨instruções_i⟩, terminando a execução da instrução **try-except**. Se nenhuma das excepções corresponder ao erro detectado, são executadas as instruções ⟨instruções_f⟩, terminando a execução da instrução **try-except**. Para exemplificar o funcionamento desta instrução, consideremos a interacção, na qual, de dois modos diferentes, tentamos aceder a um elemento de uma lista usando um índice que não existe na lista.

```
>>> teste = [1, 2]  
>>> teste[2] > 0  
IndexError: list index out of range  
  
>>> try:  
...     teste[2] > 0  
... except IndexError:
```

```

...     print('Enganou-se no índice')
...
Enganou-se no índice

```

Na realização das pilhas utilizamos a instrução `try-except` para detectar possíveis erros na utilização das pilhas. Por exemplo, se for solicitado o elemento no topo de uma pilha vazia, em lugar de deixar o Python gerar um erro correspondente a `IndexError`, o que iria de certo modo revelar a representação interna das pilhas, geramos uma mensagem apropriada.

Como alternativa, poderíamos ter escrito, por exemplo, a função `topo` como:

```

def topo(pilha):
    if pilha == []:
        raise ValueError ('A pilha não tem elementos')
    else:
        return pilha[0]

```

Na nossa implementação de pilhas, iremos utilizar a instrução `try-except`. As seguintes funções criam o tipo pilha em Python recorrendo a funções:

```

def nova_pilha():
    return []

def empurra(pilha, elemento):
    return [elemento] + pilha

def topo(pilha):
    try:
        return pilha[0]
    except IndexError:
        raise ValueError ('A pilha não tem elementos')

def tira(pilha):
    try:
        return pilha[1:]
    
```

```

except IndexError:
    raise ValueError ('A pilha não tem elementos')

def pilha(x):
    if x == []:
        return True
    elif isinstance(x, list):
        return True
    else:
        return False

def pilha_vazia(pilha):
    return pilha == []

def pilhas_iguais(p1, p2):
    return p1 == p2

def mostra_pilha(pilha):
    if pilha != []:
        for e in pilha:
            print(e)
    print('===')

```

A classe pilha

Com base na representação da Figura 13.2, podemos, alternativamente, definir pilhas como entidades mutáveis recorrendo a objectos. Quando definimos as pilhas como entidades mutáveis, as operações `empurra` e `tira` alteram a pilha que é seu argumento, devolvendo a pilha que resulta da aplicação da operação. Estas operações modificam o estado interno do objecto que corresponde à pilha. A classe `pilha` está associada aos seguintes métodos:

```
class pilha:
```

```
def __init__ (self):
    self.p = []

def empurra(self, elemento):
    self.p = [elemento] + self.p
    return self

def tira (self):
    try:
        del(self.p[0])
        return self
    except IndexError:
        print('A pilha não tem elementos')

def topo (self):
    try:
        return self.p[0]
    except IndexError:
        print('A pilha não tem elementos')

def pilha_vazia (self):
    return self.p == []

def __repr__ (self):
    if self.p != []:
        rep = ''
        for e in self.p:
            rep = rep + ' ' + str(e) + '\n'
        rep = rep + '===='
        return rep
    else:
        return '===='
```

Ao definir a classe `pilha`, propositadamente, não definimos o método correspondente ao teste `pilhas_iguais`. Este método iria comparar uma pilha que lhe é fornecida como argumento com a pilha que corresponde ao estado interno de uma instância do tipo `pilha`. Como este método não tem acesso à representação interna da pilha que lhe é fornecida como argumento, a implementação deste teste exigia que a pilha fornecida como argumento fosse “desmarchada” para comparar os seus elementos com a pilha em que corresponde ao objecto em causa e depois fosse “construída” de novo, introduzindo um peso adicional na operação. Por esta razão, decidimos não definir este método.

Com a classe `pilha`, podemos gerar a seguinte interacção:

```
>>> p1 = pilha()
>>> p1
=====
>>> p1.empurra(3)
3
=====
>>> p1.empurra(2).empurra(5)
5
2
3
=====
>>> p1.tira()
2
3
=====
```

13.2 Balanceamento de parêntesis

Ao longo do livro temos vindo a utilizar expressões que contêm parêntesis. Um dos aspectos para a correcção destas expressões corresponde ao facto dos seus parêntesis estarem balanceados, ou seja cada parênteses que é aberto tem um parênteses fechado correspondente e os pares de parêntesis estão correctamente encadeados. Por exemplo, as expressões `(2 + 3 * 5` e `(2 + 3)) * 5` estão sintacticamente incorrectas pois na primeira não existe parênteses a fechar “)”

e na segunda não existe um abrir parênteses correspondente a “)”. O conceito de balanceamento de parêntesis estende-se naturalmente à utilização de outros tipos de parêntesis, “[” e “]” e “{” e “}”.

As pilhas são estruturas de informação adequadas a avaliar o balanceamento de parêntesis numa expressão: percorrendo uma expressão da esquerda para a direita, sempre que é encontrado um abrir parêntesis este é colocado na pilha, o que significa que a pilha contém a sequência dos parêntesis que foram abertos, do mais recente para o mais antigo; sempre que é encontrado um fechar parênteses, consulta-se o elemento no topo da pilha, se este corresponder a um abrir parênteses do mesmo tipo (“(”, “[” ou “{”), então foi encontrado um par de parêntesis balanceados e o elemento no topo da pilha é retirado, em caso contrário podemos afirmar que os parêntesis não estão平衡ados.

O seguinte programa efectua a verificação do balanceamento de parêntesis numa expressão. É importante notar que, neste programa, qualquer símbolo que não seja um parênteses é ignorado. Isto significa que o programa pode também ser utilizado para verificar o balanceamento de uma cadeia de caracteres apenas contendo parêntesis. Como segunda observação, notemos que se for detectado que os parêntesis não estão平衡ados, o ciclo de verificação termina imediatamente, pois não interessa continuar a processar a expressão.

Apresentamos a função que testa o balanceamento de parêntesis utilizando pilhas como entidades imutáveis.

```
def balanceados():
    balanc = True
    pars = nova_pilha()
    exp = input('Escreva uma expressão\n--> ')
    for c in exp:
        if c in [‘(’, ‘[’, ‘{’]:
            pars = empurra(pars, c)
        elif c in [‘)’, ‘]’, ‘}’]:
            if not pilha_vazia(pars):
                outro = topo(pars)
                if (c == ‘)’ and outro == ‘(’) or \
                   (c == ‘]’ and outro == ‘[’) or \
                   (c == ‘}’ and outro == ‘{’):
                    pars = tira(pars)
```

```

        else:
            balanc = False
            break
    else:
        balanc = False
        break
if balanc and pilha_vazia(pars):
    print('parentesis correctos')
else:
    print('parentesis não balanceados')

```

Com este programa, geramos a seguinte interacção:

```

>>> balanceados()
Escreva uma expressão
2 * [(5 + 7) * a + b]
parentesis correctos
>>> balanceados()
Escreva uma expressão
3)
parentesis não balanceados
>>> balanceados()
Escreva uma expressão
({{[()]}()}())
parentesis correctos

```

13.3 Expressões em notação pós-fixa

Apresentamos uma aplicação de pilhas, a avaliação de expressões em notação pós-fixa⁴, uma notação em que o operador é escrito após os operandos. Por exemplo, a operação $2+8$ é escrita em notação pós-fixa como $2\ 8\ +$. Expressões complexas são escritas, utilizando repetidamente este método. Por exemplo, $(4+3 \times 6)+5 \times 8$ é escrita

4 3 6 × + 5 8 × +

⁴Conhecida em inglês por “Reverse Polish notation”, a qual foi inventada por [Burks et al., 1954].

A notação pós-fixa apresenta a vantagem de não necessitar da especificação da precedência entre operadores e de não necessitar da utilização de parêntesis.

Para avaliar uma expressão em notação pós-fixa, percorremos a expressão da esquerda para a direita. Ao encontrar um operador, aplicamos esse operador aos dois últimos operandos percorridos e colocamos o seu valor na expressão (substituindo os operandos e o operador). Este processo é repetido, até que a expressão seja reduzida a um único valor, o qual representa o valor da expressão. Por exemplo, para calcular o valor da expressão em notação pós-fixa $4 \ 3 \ 6 \times + \ 5 \ 8 \times +$, teremos de efectuar os seguintes passos:

$4 \ 3 \ 6 \times + \ 5 \ 8 \times +$

$4 \ 18 \ + \ 5 \ 8 \times +$

$22 \ 5 \ 8 \times +$

$22 \ 40 \ +$

62

Este processo de avaliação pode ser traduzido através de um algoritmo muito simples que recorre a uma pilha, a qual contém os operandos que foram encontrados ao percorrer a expressão. Utilizando este algoritmo começamos com uma pilha vazia (no início do algoritmo não encontrámos nenhum operando). Percorremos a expressão da esquerda para a direita. Sempre que encontramos um operando, este é colocado na pilha. Sempre que encontramos um operador, este é aplicado ao elemento no topo da pilha (o segundo operando), o qual é retirado da pilha, e ao novo elemento do topo da pilha (o primeiro operando), o qual também é retirado da pilha, e o resultado é colocado na pilha. A pilha tem assim o papel de “memorizar” os operandos que foram encontrados.

Apresentamos na Figura 13.3 a sequência de passos necessários para avaliar a expressão

$4 \ 3 \ 6 \times + \ 5 \ 8 \times +,$

recorrendo a este algoritmo, bem como a indicação da pilha gerada. A seta indica qual o componente da expressão que está a ser analisado.

A seguinte função efectua a avaliação de expressões em notação pós-fixa, usando a classe `pilha`:

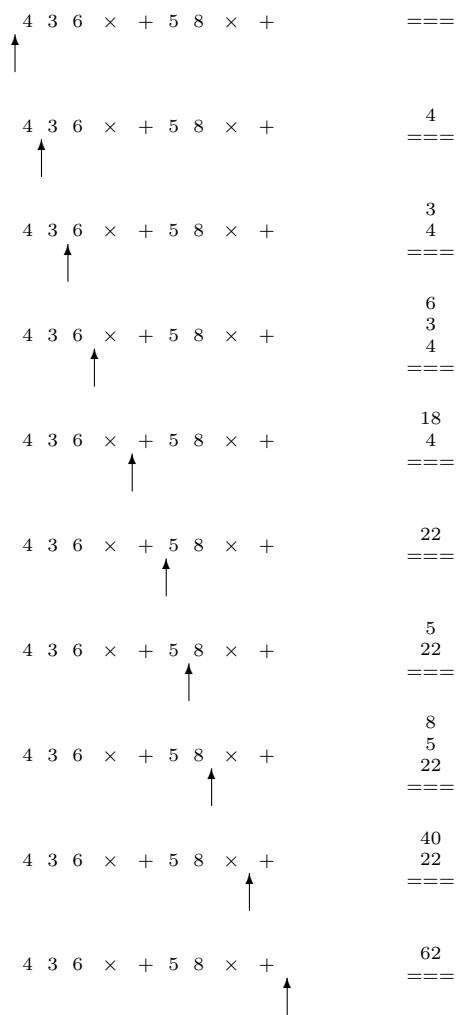


Figura 13.3: Sequência de passos necessários para se avaliar a expressão $4\ 3\ 6\ \times\ +\ 5\ 8\ \times\ +$.

```

def avalia_posfixa():

    exp = input('Escreva uma expressão\n-> ')
    operandos = pilha()

    i = 0
    while i < len(exp):
        simbolo = ''
        while i < len(exp) and exp[i] != ' ': # obtém o símbolo
            simbolo = simbolo + exp[i]
            i = i + 1
        while i < len(exp) and exp[i] == ' ': # salta brancos
            i = i + 1
        if simbolo in ['+', '-', '*', '/']:
            op2 = operandos.topo()
            operandos.tira()
            op1 = operandos.topo()
            operandos.tira()
            if simbolo == '+':
                res = op1 + op2
            elif simbolo == '-':
                res = op1 - op2
            elif simbolo == '*':
                res = op1 * op2
            else:
                res = op1 / op2
            operandos.empurra(res)
        else:
            operandos.empurra(eval(simbolo))

    res = operandos.topo()
    print('=', res)

```

Com esta função podemos gerar a seguinte interacção:

```

>>> avalia_posfixa()
Escreva uma expressão

```

```
-> 2 3 *
= 6
>>> avalia_posfixa()
Escreva uma expressão
-> 2 3 4 * +
= 14
>>> avalia_posfixa()
Escreva uma expressão
-> 4 3 6 * + 5 8 * +
= 62
```

13.4 Filas

As filas estão omnipresentes no nosso dia-a-dia. Começamos o dia em filas de transito, normalmente deparamo-nos filas quando vamos almoçar, existem filas em caixas de supermercados, em bilheteiras e em muitas outras situações do nosso quotidiano. Uma fila pode corresponder a uma fila física de entidades ou de pessoas ou pode corresponder a uma fila conceptual em que os elementos têm uma indicação da sua ordem na fila, por exemplo as filas geradas numa loja que possui uma máquina para atribuir números de ordem a clientes.

Em programação, as *filas*⁵ são estruturas de informação constituídas por uma sequência de elementos. Os elementos da fila são retirados pela ordem em que foram colocados. Uma fila tem dois elementos que se distinguem dos restantes, o elemento do início da fila, o qual pode ser retirado da fila, e o elemento do fim da fila, a seguir ao qual serão adicionados novos elementos à fila. Apenas um elemento da fila pode ser acedido ou removido, o elemento “mais antigo” da fila. Nenhum elemento pode ser adicionado, removido ou inspeccionado a não ser com o recurso a operações que colocam no fim da fila, removem do início da fila e inspeccionam o elemento no início da fila. As filas correspondem a um comportamento que vulgarmente é designado por FIFO⁶.

⁵Em inglês “queue”.

⁶Do inglês “First In First Out”.

13.4.1 Operações básicas para filas

Consideremos as operações básicas para o tipo de informação fila como tipo mutável:

1. *Construtores.* Sendo as filas estruturas dinâmicas, os construtores para o tipo fila incluem uma operação que gera filas a partir do nada, a qual designamos por *nova-fila*.
2. *Selectores.* Iremos ter um selector que indica qual o elemento no início da fila e um selector que indica o número de elementos da fila.

A operação *inicio* recebe como argumento uma fila e devolve o primeiro elemento da fila. Esta operação não altera a fila. Se a fila for vazia, esta operação é indefinida.

A operação *comprimento* recebe como argumento uma fila e devolve o número de elementos da fila.

3. *Modificadores.* Existem dois modificadores para o tipo fila, uma operação que introduz novos elementos numa fila e uma operação que retira elementos de uma fila.

A operação *coloca* recebe como argumentos uma fila e um elemento e altera a fila para a que resulta da inserção do elemento no fim da fila. O valor de *coloca* é a fila resultante.

A operação *retira* recebe como argumento uma fila e altera a fila para a que resulta da remoção do elemento no início da fila. Esta operação devolve a fila sem o elemento no início da fila. Se a fila for vazia, esta operação é indefinida.

4. *Transformadores.* Recorde-se da página 272 que um *transformador* é uma operação básica de um tipo abstracto de informação que transforma o tipo noutro tipo. São exemplos de transformadores as funções embutidas *str* e *int*.

Definimos a operação *fila-para-lista* que recebe uma fila e que produz uma lista com os mesmos elementos que a fila, e na mesma ordem, estando o primeiro elemento da fila na primeira posição da lista.

5. *Reconhecedores.*

A operação *fila* recebe como argumento um elemento de um tipo qualquer e decide se este pertence ou não ao tipo *fila*.

A operação *fila_vazia* recebe como argumento uma fila e decide se esta corresponde à fila vazia (a fila gerada por *nova_fila*).

6. Testes.

A operação *filas_iguais* recebe como argumento duas filas e decide se estas são ou não iguais.

Em resumo, o tipo *fila* tem as seguintes operações básicas (estas operações referem o tipo *elemento*, que corresponde ao tipo dos elementos da fila):

1. Construtores:

- *nova_fila* : $\{\}$ \mapsto *fila*
nova_fila() tem como valor uma fila sem elementos.

2. Selectores:

- *inicio* : *fila* \mapsto *elemento*
inicio(fila) tem como valor o elemento que se encontra no início da fila *fila*. Se a fila não tiver elementos, o valor desta operação é indefinido.
- *comprimento* : *fila* \mapsto \mathbb{N}_0
comprimento(fila) tem como valor o número de elementos da fila.

3. Modificadores:

- *coloca* : *fila* \times *elemento* \mapsto *fila*
coloca(fila, elm) altera de forma permanente a *fila* para a fila que resulta em inserir *elm* no fim da *fila*. Tem como valor a fila que resulta de inserir o elemento *elm* no fim da fila *fila*.
- *retira* : *fila* \mapsto *fila*
retira(fila) altera de forma permanente a *fila* para a fila que resulta em remover o elemento no início da *fila*. Tem como valor a fila que resulta de remover o elemento que se encontra no início da fila *fila*. Se a fila não contiver elementos, o valor desta operação é indefinido.

4. Transformadores:

— $\text{fila_para_lista} : \text{fila} \mapsto \text{lista}$

$\text{fila_para_lista}(\text{fila})$ devolve a lista com os mesmos elementos que fila , e na mesma ordem, estando o primeiro elemento da fila na primeira posição da lista.

5. *Reconhecedores:*

— $\text{fila} : \text{universal} \mapsto \text{lógico}$

$\text{fila}(\text{arg})$ tem o valor *verdadeiro*, se arg é uma fila, e tem o valor *falso*, em caso contrário.

— $\text{fila_vazia} : \text{fila} \mapsto \text{lógico}$

$\text{fila_vazia}(\text{fila})$ tem o valor *verdadeiro*, se fila é a fila vazia, e tem o valor *falso*, em caso contrário.

6. *Testes:*

— $\text{filas_iguais} : \text{fila} \times \text{fila} \mapsto \text{lógico}$

$\text{filas_iguais}(\text{fila}_1, \text{fila}_2)$ tem o valor *verdadeiro*, se fila_1 é igual a fila_2 , e tem o valor *falso*, em caso contrário.

Devemos ainda estabelecer uma representação externa para filas. Vamos convencionar que uma fila será apresentada ao mundo exterior com a sequência dos elementos que a constituem, separados por espaços em branco. A totalidade dos elementos da fila será apresentada dentro de “<”, indicando a posição de inserção e remoção de elementos. Assim, a fila com os elementos 3 5 7 (em que 3 está no início da fila) será representada por:

< 3 5 7 <

Seguindo a nossa convenção, a fila vazia será representada por < <.

13.4.2 Axiomatização

Entre as operações básicas para filas devem verificar-se as seguintes relações:

$$\text{fila}(\text{nova_fila}()) = \text{verdadeiro}$$

$$\text{fila}(\text{coloca}(f, e)) = \text{verdadeiro}$$

$$\begin{aligned}
fila(retira(f)) &= \begin{cases} verdadeiro & \text{se } f \text{ não for vazia} \\ \perp & \text{em caso contrário} \end{cases} \\
fila_vazia(nova_fila()) &= verdadeiro \\
fila_vazia(coloca(f, e)) &= falso \\
inicio(coloca(f, e)) &= \begin{cases} inicio(f) & \text{se } f \text{ não for vazia} \\ e & \text{em caso contrário} \end{cases} \\
retira(coloca(f, e)) &= \begin{cases} coloca(retira(f), e) & \text{se } f \text{ não for vazia} \\ nova_fila() & \text{em caso contrário} \end{cases} \\
comprimento(nova_fila()) &= 0 \\
comprimento(coloca(f, e)) &= 1 + comprimento(f) \\
fila_iguais(f_1, f_2) &= \begin{cases} fila_vazia(f_1) & \text{se } fila_vazia(f_2) \\ fila_vazia(f_2) & \text{se } fila_vazia(f_1) \\ inicio(f_1) = inicio(f_2) & \text{se } fila_iguais(retira(f_1), \\ & \quad retira(f_2)) \\ falso & \text{em caso contrário} \end{cases}
\end{aligned}$$

13.4.3 Representação de filas

Representaremos filas recorrendo a listas. A representação de uma fila é definida do seguinte modo:

1. $\mathfrak{R}[< <] = []$
2. $\mathfrak{R}[< X_i \dots X_f <] = [\mathfrak{R}[X_i], \dots, \mathfrak{R}[X_f]]$

13.4.4 A classe fila

Definimos filas recorrendo a objectos. O estado interno de uma fila é definido através da variável `self.f` a qual contém a representação da fila como descrita na Secção 13.4.3.

Na classe `fila` definimos um transformador, `fila_para_lista`, que produz uma lista contendo todos os elementos da fila. Com a definição deste transformador, deixamos de ter os problemas discutidos na página 345 e, por essa razão,

podemos definir o método `filas_iguais`.

A classe `fila` é definida através dos seguintes métodos:

```
class fila:

    def __init__(self):
        self.f = []

    def inicio(self):
        try:
            return self.f[0]
        except IndexError:
            print('inicio: a fila não tem elementos')

    def comprimento(self):
        return len(self.f)

    def coloca(self, elemento):
        self.f = self.f + [elemento]
        return self

    def retira(self):
        try:
            del(self.f[0])
            return self
        except IndexError:
            print('retira: a fila não tem elementos')

    def fila_para_lista(self):
        lst = []
        for i in range(len(self.f)):
            lst = lst + [self.f[i]]
        return lst
```

```

def fila_vazia(self):
    return self.f == []

def filas_iguais(self, outra):
    outra_lista = outra.fila_para_lista()
    if len(self.f) != len(outra_lista):
        return False
    else:
        for i in range(len(self.f)):
            if self.f[i] != outra_lista[i]:
                return False
    return True

def __repr__(self):
    f = '< '
    if self.f != []:
        for i in range(len(self.f)):
            f = f + self.f[i].__repr__() + ' '
    f = f + '<'
    return f

```

Com esta classe podemos gerar a seguinte interacção:

```

>>> f = fila()
>>> f.retira()
retira: a fila não tem elementos
>>> f.inicio()
inicio: a fila não tem elementos
>>> f.coloca(1)
< 1 <
>>> f.coloca(2).coloca(3)
< 1 2 3 <
>>> f.comprimento()
3
>>> f.inicio()
1

```

```
>>> f.retira()  
< 2 3 <
```

13.5 Simulação de um supermercado

Uma das áreas de aplicação da informática, conhecida por *simulação*, consiste em “*imitar*” a evolução do comportamento de um processo do mundo real. Para efectuar uma simulação é necessário construir um modelo do comportamento do processo, o qual captura as características essenciais do processo e especifica o seu comportamento, estudando depois a evolução do processo ao longo do tempo. Um exemplo sofisticado de simulação corresponde a um simulador de voo, através do qual os pilotos são treinados a lidar com situações adversas que podem ocorrer durante um voo.

Um programa de simulação tem pois como finalidade o estudo de uma situação hipotética do mundo real, avaliando o comportamento de um sistema ou de um processo em determinadas situações. Através da alteração de variáveis associadas à simulação, é possível prever o comportamento do sistema sob situações diversas.

Apresentamos um programa de simulação do tempo de espera nas caixas de um supermercado. Este programa corresponde a uma aplicação do tipo fila para representar a fila associada a cada caixa. As variáveis de simulação do nosso programa correspondem ao número de caixas abertas, a uma medida da afluência dos clientes ao supermercado e a uma medida do número médio de compras previsto para os clientes. Alterando os valores destas variáveis é possível obter uma ideia do tempo que os clientes esperam para ser atendidos, eventualmente decidindo quantas caixas devem estar em funcionamento, em função da afluência e do número expectável de compras dos clientes.

O programa utiliza as seguintes variáveis para a simulação:

- **afluencia.** Caracteriza a afluência dos clientes ao supermercado. O seu valor corresponde a um inteiro entre 1 e 100, em que 1 representa a afluência mínima e 100 representa a afluência máxima.
- **apetencia.** Caracteriza a apetência dos clientes para fazerem compras. O seu valor corresponde a um inteiro entre 1 e 100, em que 1 representa

a apetência mínima e 100 representa a apetência máxima. Por exemplo, no final do mês assume-se que as pessoas fazem as compras do mês pelo que a sua apetência para fazer compras será maior do que no meio do mês. Quanto maior for a apetência para compras maior será o número de compras associadas a cada cliente.

- **n_caixas**. Representa o número de caixas que estão abertas durante a simulação.
- **ciclos**. Esta variável define o tempo que demora a simulação. Este tempo é caracterizado por um inteiro. Em cada ciclo, poderá ou não ser inserido um cliente numa fila de uma caixa, dependendo do valor da **afluência** e são verificados quais os clientes nas filas das caixas cujas compras já foram todas processadas.

Antes de apresentar o programa que efectua a simulação, vamos discutir os tipos de informação que este utiliza:

- *O tipo cliente* corresponde a um cliente do supermercado. Cada cliente é caracterizado por um certo número de compras (**self.items**) e pelo instante (número do ciclo) em que foi colocado numa das filas de pagamento (este instante corresponde à variável **self.entrada**). A classe **cliente** é definida do seguinte modo:

```
class cliente:

    def __init__(self, items, entrada):
        self.items = items
        self.entrada = entrada

    def artigos(self):
        return self.items

    def tempo_entrada(self):
        return self.entrada
```

```
def __repr__(self):
    return '[' + str(self.items) + \
        ':' + str(self.entrada) + ']'
```

- O tipo *caixa* corresponde a uma caixa aberta no supermercado. Este tipo é caracterizado por uma fila, tal como foi apresentada na Secção 13.4, a qual corresponde à fila dos clientes na caixa, e pela a seguinte informação adicional:

- O número da caixa;
- O instante a partir do qual a caixa está pronta para atender um cliente na fila. Se a caixa estiver a atender um cliente, esse instante corresponde ao instante em que o cliente deixa de ser atendido, se a fila da caixa não tem clientes, então esse instante é o instante actual;
- O número total de clientes que foi atendido pela caixa. Este valor é aumentado em uma unidade sempre que a caixa termina o atendimento de um cliente;
- O número total de produtos processados pela caixa. Este número é contabilizado sempre que a caixa termina o atendimento de um cliente, somando o número de produtos que o cliente comprou ao total de produtos processados pela caixa;
- O tempo total de espera (medido em número de ciclos) associado a todos os clientes que foram atendidos pela caixa. Este número é contabilizado sempre que a caixa termina o atendimento de um cliente, somando o tempo de espera desse cliente ao tempo total de espera associado à caixa;
- O número de produtos que a caixa processa por unidade de tempo (entenda-se por ciclo). Este valor, entre 1 e 5, é calculado aleatoriamente sempre que uma caixa é aberta.

Em programação, é comum necessitar-se de gerar aleatoriamente números dentro de um certo intervalo. As sequências de *números aleatórios* produzidas por um programa não são aleatórias no verdadeiro sentido da palavra, na medida em que é possível prever qual será a sequência gerada, e por isso são denominadas *pseudo-aleatórias*. Contudo, os números gerados podem ser considerados aleatórios porque não apresentam qualquer correlação entre si. Estas sequências de

números aleatórios são repetidas sempre que o programa é executado de novo. Este facto não representa um inconveniente mas sim uma vantagem, uma vez que permite duplicar as condições de execução de um programa, o que é extremamente útil para a sua depuração.

Suponhamos então que pretendíamos gerar aleatoriamente números inteiros no intervalo $[1, n]$. A ideia básica na geração de números aleatórios consiste em gerar números uniformemente distribuídos no intervalo $[0, 1[$. Multiplicando o número gerado por n , obtemos um número real no intervalo $[0, n[$, finalmente desprezando a parte decimal do número gerado e adicionando 1 ao resultado, obtém-se um número inteiro no intervalo $[1, n]$.

Em Python, a função `random()`, localizada na biblioteca `random`, gera números aleatórios no intervalo $[0, 1[$. No nosso programa de simulação utilizamos a função `random`.

O tipo caixa é realizado através da classe `caixa`, a qual apresenta o seguinte comportamento para alguns dos seus métodos (o comportamento dos restantes métodos é trivial):

- `muda_info_caixa`, recebe dois argumentos, o número de uma caixa, `nb_caixa`, e um inteiro, `inst`, e actualiza o instante de tempo (`inst`) em que a caixa `nb_caixa` está disponível para atender um cliente;
- `aumenta_clientes_atendidos`, recebe como argumento o número de uma caixa, `nb_caixa`, e actualiza em uma unidade o número de clientes atendidos por essa caixa;
- `muda_info_produtos`, recebe dois argumentos, o número de uma caixa, `nb_caixa` e um inteiro, `nb_prods`, e actualiza com o valor `nb_prods` o número de produtos processados pela caixa `nb_caixa`;
- `muda_info_t_espera`, recebe como argumentos o número de uma caixa (`nb_caixa`) e uma unidade de tempo (`t`) e actualiza com o valor `t` o tempo acumulado de espera dos clientes atendidos pela caixa `nb_caixa`.

A classe `caixa` é definida do seguinte modo⁷:

```
from filas import *
```

⁷Esta classe necessita de importar a classe `fila` e a função `random`.

```
from random import *

class caixa:

    def __init__(self, nb_cx, numero):
        self.fila_cl = fila()
        self.pronta_em = 0
        self.cts_atend = 0
        self.produtos_processados = 0
        self.total_espera = 0
        self.n_c = numero
        # número de produtos tratados por unidade de tempo
        self.prods_ut = int(random() * 5) + 1

    def muda_info_caixa(self, valor):
        self.pronta_em = valor

    def aumenta_clientes_atendidos(self):
        # muda o valor
        self.cts_atend = self.cts_atend + 1

    def muda_info_produtos(self, produtos):
        self.produtos_processados =
            self.produtos_processados + produtos

    def muda_info_t_espera(self, t):
        self.total_espera = self.total_espera + t

    def fila_caixa(self):
        return self.fila_cl

    def num_caixa(self):
        return self.n_c
```

```

def info_caixa(self):
    return self.pronta_em

def clientes_atendidos(self):
    return self.cts_atend

def info_produtos(self):
    return self.produtos_processados

def info_t_espera(self):
    return self.total_espera

def produtos_processados_ciclo(self):
    return self.prods_ut

def __repr__(self):
    rep = 'Caixa ' + str(self.n_c) + ' (' + \
          str(self.pronta_em) + \
          '): ' + self.fila_cl.__repr__()
    return rep

```

O programa de simulação do supermercado é constituído por dois ciclos:

1. No primeiro, um ciclo `for` que é executado tantas vezes quanto o número de ciclos a simular, são tratados os clientes que se encontram na filas das caixas (função `trata_clientes`) e decide-se aleatoriamente se deve ser gerado um novo cliente, com um número de compras que também é decidido de forma aleatória, colocando-o, de novo de forma aleatória, numa das filas de caixa.
2. O segundo ciclo, um ciclo `while` que é executado enquanto existirem clientes nas filas, aparece depois de todos os ciclos especificados na simulação terem sido executados. Este ciclo serve para processar os clientes que entretanto foram colocados nas filas das caixas mas que não foram atendidos. Podemos considerar que este segundo ciclo surge depois das portas do supermercado já terem sido fechadas (não entram mais clientes) e é executado enquanto existem clientes por atender.

```
from random import *
from filas import *
from caixa import *
from cliente import *

def simula_supermercado(afluencia, apetencia, n_caixas, ciclos):

    caixas = []
    for i in range(n_caixas):
        caixas = caixas + [caixa(n_caixas, i)]

    for i in range(1, ciclos+1):
        print('== CICLO == ', i)

        # processa os clientes nas caixas
        trata_clientes(i, caixas)

        # decide a criação de novo cliente e número de compras
        aleatorio = random()
        limiar = afluencia/100
        if aleatorio < limiar: # um novo cliente é criado
            num_compras = int(random() * 3 * apetencia) + 1
            print('--> Criado cliente com', num_compras, 'artigos')
            c = cliente(num_compras, i)

            # insere o cliente na fila de uma caixa
            nb_cx = int(random() * n_caixas) # selecciona a caixa
            cx = caixas[nb_cx]
            fila = cx.fila_caixa()
            fila.coloca(c)

        mostra_caixas(caixas)

    # processa as filas depois do fecho de entradas
    i = ciclos + 1
    print('Entradas fechadas')
```

```

while existem_caixas_com_fila(caixas):
    print(' == CICLO == ', i)
    # processa os clientes nas caixas
    trata_clientes(i, caixas)
    mostra_caixas(caixas)
    i = i + 1

processa_resultados(caixas)

```

A função `trata_clientes` corresponde ao atendimento dos clientes que estão no início das filas das caixas num determinado instante de tempo. Esta função é definida do seguinte modo:

```

def trata_clientes(tempo, caixas):

    for i in range(len(caixas)):
        cx = caixas[i]
        if not cx.fila_caixa().fila_vazia():
            # se a fila associada à caixa não é vazia
            # verifica se o atendimento ao cliente terminou
            cliente = cx.fila_caixa().inicio()
            nb_artigos = cliente.artigos()
            t_atendimento = tempo - cx.info_caixa()
            artigos_prcds = \
                nb_artigos/cx.produtos_processados_ciclo()
            if artigos_prcds < t_atendimento:
                # o cliente sai da fila da caixa
                t_entrada_fila = cliente.tempo_entrada()
                t_espera = tempo - t_entrada_fila
                print('--> Processado cliente com', \
                    nb_artigos, 'artigos na caixa', \
                    cx.num_caixa(), \
                    'tempo de espera', t_espera)
                cx.muda_info_caixa(tempo+1)
                cx.aumenta_clientes_atendidos()
                cx.muda_info_produtos(nb_artigos)

```

```

        cx.muda_info_t_espera(t_espera)
        f = cx.fila_caixa()
        f.retira() # o cliente sai da fila
    else:
        # para as caixas vazias actualiza o tempo potencial
        # para atendimento
        cx.muda_info_caixa(tempo)

```

A função `processa_resultados` mostra as estatísticas da execução do programa.

```

def processa_resultados(caixas):

    for i in range(len(caixas)):
        cx = caixas[i]
        c_t = cx.clientes_atendidos()
        if c_t != 0:
            print('Caixa', cx.num_caixa(), '(atendimento ' + \
                  str(cx.produtos_processados_ciclo()) + \
                  ' produtos por ciclo):')
            print(str(c_t) + ' clientes atendidos, ' + \
                  'média produtos/cliente ' + \
                  str(cx.info_produtos()/c_t) + \
                  ',\ntempo médio de espera ' + \
                  str(cx.info_t_espera()/c_t))
        else:
            print('Caixa ' + str(cx.num_caixa()) + \
                  ': não atendeu clientes')

```

A função `mostra_caixas` mostra em forma de texto a informação associada às caixas:

```

def mostra_caixas(cx):
    for cx in cx:
        print(cx)

```

A função `existem_caixas_com_fila` tem o valor verdadeiro apenas se existir alguma caixa cuja fila de clientes não seja vazia:

```
def existem_caixas_com_fila(caixas):
    for i in range(len(caixas)):
        if not caixas[i].fila_caixa().fila_vazia():
            return True
    return False
```

A seguinte interacção mostra parte dos resultados produzidos durante uma simulação:

```
>>> simula_supermercado(100, 15, 5, 10)
== CICLO == 1
--> Criado cliente com 1 artigos
Caixa 0 (1): < <
Caixa 1 (1): < [1:1] <
Caixa 2 (1): < <
Caixa 3 (1): < <
Caixa 4 (1): < <

== CICLO == 2
--> Processado cliente com 1 artigos na caixa 2 tempo de espera 1
--> Criado cliente com 35 artigos
Caixa 0 (2): < <
Caixa 1 (2): < [35:2] <
Caixa 2 (2): < <
Caixa 3 (2): < <
Caixa 4 (2): < <

...
== CICLO == 9
--> Criado cliente com 24 artigos
Caixa 0 (8): < [3:8] <
Caixa 1 (2): < [35:2] [4:6] [16:7] [24:9] <
Caixa 2 (9): < <
Caixa 3 (3): < [27:3] <
Caixa 4 (4): < [45:4] [9:5] <

== CICLO == 10
--> Processado cliente com 3 artigos na caixa 1 tempo de espera 2
```

```
--> Criado cliente com 20 artigos
Caixa 0 (10): < <
Caixa 1 (2): < [35:2] [4:6] [16:7] [24:9] <
Caixa 2 (10): < <
Caixa 3 (3): < [27:3] <
Caixa 4 (4): < [45:4] [9:5] [20:10] <

Entradas fechadas
== CICLO == 11
Caixa 0 (11): < <
Caixa 1 (2): < [35:2] [4:6] [16:7] [24:9] <
Caixa 2 (11): < <
Caixa 3 (3): < [27:3] <
Caixa 4 (4): < [45:4] [9:5] [20:10] <
...
== CICLO == 44
Caixa 0 (44): < <
Caixa 1 (32): < [24:9] <
Caixa 2 (44): < <
Caixa 3 (44): < <
Caixa 4 (44): < <

== CICLO == 45
--> Processado cliente com 24 artigos na caixa 2 tempo de espera 36
Caixa 0 (45): < <
Caixa 1 (45): < <
Caixa 2 (45): < <
Caixa 3 (45): < <
Caixa 4 (45): < <

Caixa 0 (atendimento 2 produtos por ciclo):
1 clientes atendidos, media produtos/cliente 3.0,
tempo médio de espera 2.0
Caixa 1 (atendimento 2 produtos por ciclo):
5 clientes atendidos, media produtos/cliente 16.0,
tempo médio de espera 19.4
Caixa 2: não atendeu clientes
```

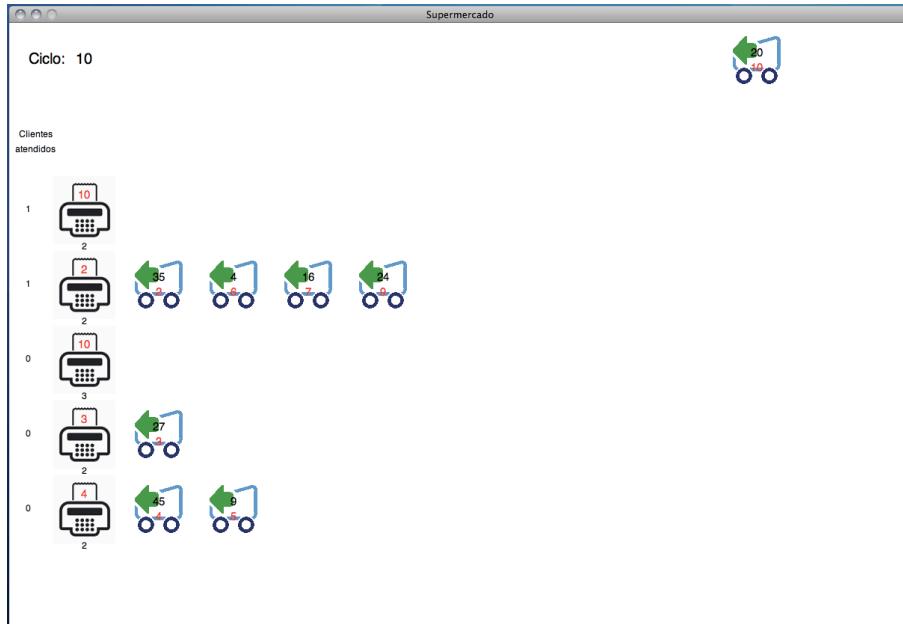


Figura 13.4: Representação gráfica do supermercado no ciclo 10.

Caixa 3 (atendimento 2 produtos por ciclo):

1 clientes atendidos, media produtos/cliente 27.0,
tempo médio de espera 14.0

Caixa 4 (atendimento 2 produtos por ciclo):

3 clientes atendidos, media produtos/cliente 24.666666666666668,
tempo médio de espera 27.666666666666668

13.6 Representação gráfica

Em muitos programas de simulação é desejável apresentar de uma forma gráfica a evolução do processo simulado em vez de mostrar de forma textual o que está a acontecer, tal como o fizemos no exemplo anterior. Por exemplo, seria mais interessante que pudéssemos seguir a evolução da situação das filas das caixas através de uma janela como se mostra na Figura 13.4.

Nesta secção apresentamos as modificações que é necessário introduzir no programa de simulação do supermercado de modo que este possa mostrar a evolução

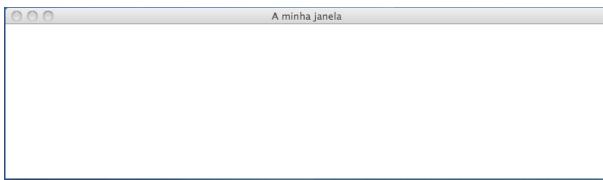


Figura 13.5: Janela criada por `GraphWin('A minha janela', 800, 200)`.

da simulação de uma forma gráfica. Para isso, utilizamos o módulo gráfico desenvolvido por John Zelle⁸.

Para a utilização de uma janela gráfica, é necessário começar por criar um objecto que corresponde à janela a ser utilizada. A criação de uma janela é feita pela função `GraphWin`, cuja sintaxe é definida pelas seguintes expressões em notação BNF:

```
GraphWin({<título>, <largura>, <altura>})
<título> ::= <cadeia de caracteres>
<largura> ::= <expressão>
<altura> ::= <expressão>
```

Nesta função, `<título>` corresponde ao título que é colocado na janela e `<largura>` e `<altura>` (expressões cujo valor é um inteiro positivo) especificam as dimensões da janela. Os parâmetros da função `GraphWin` são opcionais. Se não forem especificados, o Python assume os seguintes valores: 'Graphics Window', 200, 200. Por exemplo, a instrução `j = GraphWin('A minha janela', 800, 200)` cria a janela apresentada na Figura 13.5, associando-a à variável `j`.

A partir do momento em que é criada uma janela correspondente a um objecto gráfico, é possível desenhar entidades nessa janela. Para isso convém saber que o conteúdo de uma janela (a zona onde podemos desenhar entidades) é constituído por um conjunto de pequenos pontos chamados *pixeis*⁹. Na janela, os *pixeis* são referenciados através de um sistema de coordenadas cartesianas, cuja origem se situa no canto superior esquerdo da janela, aumentando a primeira coordenada da esquerda para a direita até ao valor máximo de `<largura>` - 1, e aumentando a

⁸Este módulo pode ser obtido em <http://mcsp.wartburg.edu/zelle/python/graphics.py> e a sua documentação está disponível em <http://mcsp.wartburg.edu/zelle/python/graphics/index.html>.

⁹Do inglês, "pixel" (picture element).

segunda coordenada de cima para baixo até ao valor máximo de $\langle\text{altura}\rangle - 1$. O *pixel* localizado no ponto de coordenadas x e y é referenciado através da função `Point(x , y)`.

Associado a um objecto do tipo `GraphWin`¹⁰ existe um conjunto de métodos para manipular a janela, entre os quais:

- `Circle($\langle\text{ponto}\rangle$, $\langle\text{raio}\rangle$)`, cria um objecto gráfico correspondente a um círculo, centrado no *pixel* correspondente ao $\langle\text{ponto}\rangle$ e com um raio (em número de *pixels*) dado pela expressão $\langle\text{raio}\rangle$. A criação deste objecto gráfico não origina o seu desenho na janela.
- `Rectangle($\langle\text{ponto}_1\rangle$, $\langle\text{ponto}_2\rangle$)`, cria um objecto gráfico correspondente a um rectângulo, com vértices opostos localizados nos *pixels* correspondentes aos pontos $\langle\text{ponto}_1\rangle$ e $\langle\text{ponto}_2\rangle$. A criação deste objecto gráfico não origina o seu desenho na janela.
- `Text($\langle\text{ponto}\rangle$, $\langle\text{texto}\rangle$)`, cria um objecto gráfico correspondente a texto, associado à cadeia de caracteres $\langle\text{texto}\rangle$, o qual está centrado no *pixel* correspondente ao ponto $\langle\text{ponto}\rangle$. A criação deste objecto gráfico não origina o seu desenho na janela.
- `setTextColor($\langle\text{cor}\rangle$)`, muda a cor do objecto gráfico correspondente a texto para a cor correspondente à cadeia de caracteres $\langle\text{cor}\rangle$. Esta alteração ao objecto gráfico não origina o seu desenho na janela.
- `Image($\langle\text{ponto}\rangle$, $\langle\text{ficheiro}\rangle$)`, cria um objecto gráfico correspondente à imagem contida no ficheiro indicado pela cadeia de caracteres $\langle\text{ficheiro}\rangle$ (o qual deve estar em formato `gif`), centrada no *pixel* correspondente ao ponto $\langle\text{ponto}\rangle$. A criação deste objecto gráfico não origina o seu desenho na janela.
- `getMouse()`. Espera que o utilizador carregue com o rato na janela gráfica.
- `update()`. Força a execução das operações gráficas que possam estar pendentes.
- `close()`. Fecha a janela.

¹⁰Note-se que a função `GraphWin` é o construtor da classe `GraphWin`.

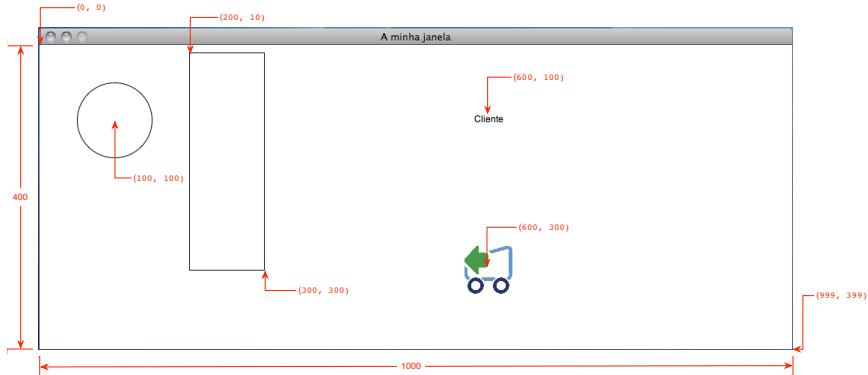


Figura 13.6: Objectos gráficos criados durante a interacção.

Cada objecto gráfico criado está, por sua vez, associado a métodos que controlam a sua manipulação na janela gráfica:

- `draw(janela)`, desenha o objecto gráfico correspondente na janela gráfica `janela`.
- `undraw()`, apaga o objecto gráfico correspondente da janela gráfica onde este está desenhado.

Como exemplo, consideremos a seguinte interacção, na qual o ficheiro com o nome `clientepq.gif` corresponde à imagem apresentada no lado esquerdo da Figura 13.7:

```
>>> from graphics import *
>>> janela = GraphWin('A minha janela', 1000, 400)
>>> obj1 = Circle(Point(100, 100), 50)
>>> obj2 = Rectangle(Point(200, 10), Point(300, 300))
>>> obj1.draw(janela)
>>> obj2.draw(janela)
>>> icon = Image(Point(600, 300), 'clientepq.gif')
>>> icon.draw(janela)
>>> texto = Text(Point(600, 100), 'Cliente')
>>> texto.draw(janela)
```

Esta interacção origina os objectos gráficos apresentados na Figura 13.6, na qual

também se mostram as coordenadas associadas a cada objecto.

Vamos agora discutir as alterações que é necessário introduzir no programa apresentado na Secção 13.5 de modo a que este apresente uma interface gráfica. Estas envolvem uma mudança aos tipos de informação utilizados, de modo a que estes também contenham informação relativa aos objectos gráficos que os representam, e envolvem alterações ao próprio programa.

Comecemos por considerar as alterações necessárias ao tipo `cliente`. Este tipo está definido no ficheiro com o nome `cliente0G` (cliente com objectos gráficos). Recorde-se que um cliente é um objecto cujo estado interno contém duas variáveis, o número de compras e o instante de entrada na fila. Para utilizar a interface gráfica adicionamos mais uma variável ao seu estado interno, `self.rep_graf`, correspondente à representação gráfica do cliente. Esta representação gráfica é constituída por três objectos gráficos, uma representação do carrinho de compras (correspondente à imagem apresentada no lado esquerdo da Figura 13.7¹¹), uma imagem da representação do número de compras e uma imagem da representação do instante de entrada na fila. Assim, a variável `self.rep_graf` está associada a um tuplo de três elementos, contendo a representação de cada um destes objectos gráficos. Como a posição na janela em que aparece a representação de um cliente varia ao longo da execução do programa, decidimos não criar a representação gráfica no momento da criação do cliente, sendo esta apenas criada quando se sabe qual a posição na janela gráfica em que o cliente deve ser mostrado. Existe um modificador associado à classe cliente, `cria_icons_cliente`, que cria os ícones associados a um cliente, altera o estado interno, e mostra esses ícones numa determinada posição da janela.

```
from graphics import *

class cliente:

    def __init__(self, items, entrada, win):
        self.items = items
        self.entrada = entrada

    def cria_icons_cliente(self, items, t_entrada, x, y, win):
```

¹¹Esta imagem foi obtida de <http://findicons.com/search/check-out/7>.

```
objs_graf = self.mostra_cliente(items, t_entrada, x, y, win)
self.rep_graf = (objs_graf[0], objs_graf[1], objs_graf[2])
return objs_graf

def apaga_cliente(self, icons, win):
    # remove da janela os icons correspondentes ao cliente
    icons[0].undraw()
    icons[1].undraw()
    icons[2].undraw()

def artigos(self):
    return self.items

def tempo_entrada(self):
    return self.entrada

def rep_grafica(self):
    return self.rep_graf

def __repr__(self):
    return '[' + str(self.items) + \
           ':' + str(self.entrada) + ']'

def mostra_cliente(self, n_compras, t_entrada, x, y, win):
    icon = Image(Point(x, y), 'clientepq.gif')
    icon.draw(win)
    rep_n_comp = Text(Point(x, y-10), n_compras)
    rep_n_comp.setSize(15)
    rep_n_comp.draw(win)
    rep_t_ent = Text(Point(x, y+10), t_entrada)
    rep_t_ent.setSize(15)
```

```

rep_t_ent.setTextColor('red')
rep_t_ent.draw(win)
return (icon, rep_n_comp, rep_t_ent)

```

A classe `caixa` para o programa com interface gráfica existe no ficheiro com o nome `caixasOG` (caixas com objectos gráficos). Para além da informação que este tipo já continha (descrita na página 360), este tipo tem as seguintes variáveis de estado adicionais:

- Fila com os objectos gráficos existentes na fila da caixa. Estes objectos gráficos correspondem à representação de clientes que já foi apresentada;
- A posição, na janela gráfica, do primeiro cliente que se encontra na fila da caixa;
- A distância, na fila gráfica, entre os objectos correspondentes aos clientes;
- A posição na janela gráfica onde é mostrado o número de clientes atendidos pela caixa;
- O objecto gráfico correspondente ao número de clientes atendidos.

```

from filas import *
from random import *
from posicao import *
from graphics import *

class caixa:

    def __init__(self, numero, win):
        cxs = []
        for i in range(numero):
            pronta_em = 0
            clientes_atendidos = 0
            produtos_processados = 0
            total_espera = 0
            # número de produtos tratados por unidade de tempo
            prods_ut = int(random() * 5) + 1

```

```

cxs = cxs + \
    [[fila(), \
        pronta_em, \
        clientes_atendidos, \
        produtos_processados, \
        total_espera, \
        prods_ut]]
self.nb_caixas = numero
self.filas = cxs

# lista que para cada caixa contém os objectos gráficos dos
# clientes na fila da caixa, a posição gráfica do
# primeiro cliente na fila, o objecto gráfico correspondente
# ao numero de clientes atendidos e a sua posição e a
# representação do instante em que a caixa está pronta
# para atender um novo cliente
objgrfs = []
for i in range(numero):
    pos_primeiro_cliente_x = 200
    pos_primeiro_cliente_y = 750 - 100 * (numero - i)
    d_c = 100
    pos_nb_clientes_x = 25
    pos_nb_clientes_y = pos_primeiro_cliente_y
    c_a_inic = Text(Point(pos_nb_clientes_x, \
                          pos_nb_clientes_y), \
                  0)
    c_a_inic.draw(win)
    rep_graf_pronta = Text(\
        Point(100, \
              730 - 100 * (self.nb_caixas - i)), \
        pronta_em)
    rep_graf_pronta.setSize(15)
    rep_graf_pronta.setTextColor('red')
    rep_graf_pronta.draw(win)
    objgrfs = objgrfs + \
        [[fila(), \
            (pos_primeiro_cliente_x, \

```

```

        pos_primeiro_cliente_y), \
        d_c,
        (pos_nb_clientes_x, pos_nb_clientes_y), \
        c_a_inic, \
        rep_graf_pronta]]
# mostra a eficiência da caixa
ppc = Text(Point(100, \
                  800 - 100 * (self.nb_caixas - i)), \
            self.produtos_processados_ciclo(i))
ppc.draw(win)
self.rep_graf_filas = objgrfs

def muda_info_caixa(self, nb_caixa, valor, win):
    self.filas[nb_caixa][1] = valor
    self.rep_graf_filas[nb_caixa][5].undraw()
    rep_graf_pronta = Text(\
        Point(100, \
              730-100*(self.nb_caixas-nb_caixa)), \
        valor)
    rep_graf_pronta.setSize(15)
    rep_graf_pronta.setTextColor('red')
    rep_graf_pronta.draw(win)
    self.rep_graf_filas[nb_caixa][5] = rep_graf_pronta

def aumenta_clientes_atendidos(self, nb_caixa, win):
    # muda o valor
    self.filas[nb_caixa][2] = self.filas[nb_caixa][2] + 1
    # apaga na interface gráfica valor antigo
    icon_valor = self.rep_graf_filas[nb_caixa][4]
    icon_valor.undraw()
    # cria novo objecto e desenha-o
    (x, y) = self.rep_graf_filas[nb_caixa][3]
    icon_valor = Text(Point(x, y), \
                      self.filas[nb_caixa][2])
    self.rep_graf_filas[nb_caixa][4] = icon_valor
    icon_valor.draw(win)

```

```
win.update()

def muda_info_produtos(self, nb_caixa, produtos):
    self.filas[nb_caixa][3] = \
        self.filas[nb_caixa][3] + produtos

def muda_info_t_espera(self, nb_caixa, t):
    self.filas[nb_caixa][4] = \
        self.filas[nb_caixa][4] + t

def apaga_clientes(self, nb_cx, win):
    fila = self.filas[nb_cx][0]
    fila_graf = self.rep_graf_filas[nb_cx][0]
    while not fila_graf.fila_vazia():
        c = fila.inicio()
        rc = fila_graf.inicio()
        c.apaga_cliente(rc, win)
        fila_graf.retira()

def rep_graf_cliente_na_fila(self, n_compras, nb_cx, win):
    n_caixas = self.nb_caixas
    (x_0, y_0) = self.rep_nb_clientes[nb_cx][0]
    # calcula a posição na fila gráfica
    x = x_0 + \
        100 * (self.pos_graf_fila(nb_cx).comprimento() + 1)
    y = y_0 - \
        100 * (n_caixas - nb_cx)
    #
    c = cria_rep_gr_cliente(num_compras, x, y, win)
    fila = scaixas.fila_icons(nb_cx, win)
    fila.coloca(c)
    return (icon_cliente, icon_compras)
```

```
def actualiza_interface(self, nb_caixa, win):
    # cria a fila gráfica representando os clientes na caixa

    # obtém a lista com os clientes na fila
    fila = self.filas[nb_caixa][0]
    clientes = fila.fila_para_lista()

    for i in range(len(clientes)):
        cl = clientes[i]
        fila_graf = self.rep_graf_filas[nb_caixa][0]
        (x_i, y_i) = self.rep_graf_filas[nb_caixa][1]
        inc = self.rep_graf_filas[nb_caixa][2]
        (carrinho, compras, entrada) = \
            cl.cria_icons_cliente(cl.artigos(), \
                                  cl.tempo_entrada(), \
                                  x_i + i * inc, \
                                  y_i, \
                                  win)
        win.update()
        fila_graf.coloca((carrinho, compras, entrada))

def num_caixas(self):
    return self.nb_caixas

def fila_caixa(self, nb_caixa):
    return self.filas[nb_caixa][0]

def info_caixa(self, nb_caixa):
    return self.filas[nb_caixa][1]

def clientes_atendidos(self, nb_caixa):
    return self.filas[nb_caixa][2]
```

```
def info_produtos(self, nb_caixa):
    return self.filas[nb_caixa] [3]

def info_t_espera(self, nb_caixa):
    return self.filas[nb_caixa] [4]

def produtos_processados_ciclo(self, nb_caixa):
    return self.filas[nb_caixa] [5]

def fila_graf_caixa(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [0]

def pos_fila_graf_caixa(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [1]

def incremento_icons_caixas(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [2]

def pos_inicial_icons_clientes(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [3]

def pos_ikon_nb_clientes(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [4]

def icon_instante_pronta(self, nb_caixa):
    return self.rep_graf_filas[nb_caixa] [5]
```

```

def pos_novo_icon_fila(self, nb_caixa, fila):
    (x_i, y_i) = self.rep_graf_filas[nb_caixa][1]
    incremento = self.rep_graf_filas[nb_caixa][2]
    x = x_i + fila.comprimento() * incremento
    y = y_i
    return (x, y)

def existem_caixas_com_fila(self):
    res = False
    for i in range(self.nb_caixas):
        fila_da_caixa = self.fila_caixa(i)
        if not fila_da_caixa.fila_vazia():
            res = True
            break
    return res

def __repr__(self):
    rep = ''
    for i in range(self.nb_caixas):
        fila = self.filas[i][0]
        rep = rep + 'Caixa ' + str(i) + ' (' + \
            str(self.filas[i][1]) + \
            '): ' + fila.__repr__() + '\n'
    return rep

```

Para além do ícone correspondente ao carrinho de compras (apresentado na parte esquerda da Figura 13.7), o programa usa um ícone para representar as caixas, o qual é apresentado na parte direita da Figura 13.7¹². Este ícone existe no ficheiro cujo nome é `caixapq.gif`.

O seguinte código define a classe ponto.

```
class ponto:
```

```
    def __init__(self, x, y):
```

¹²Obtido de http://www.123rf.com/clipart-vector/cash_counter.html.



Figura 13.7: Ícones usados pelo programa.

```
self.x = x
self.y = y
```

```
def pos_x(self):
    return self.x
```

```
def pos_y(self):
    return self.y
```

```
def __repr__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

O programa de simulação é alterado do seguinte modo:

```
from random import *
from filas import *
from caixaOG import *
from clienteOG import *
from graphics import *

def simula_supermercado(afluencia, apetencia, n_caixas, ciclos):

    # define a janela para visualizaçāoo dos resultados
    win = GraphWin('Supermercado', 1200, 800)
    mostra_janela_inicial(win, n_caixas, 0)
    nbciclo = Text(Point(100, 50), 0)
    nbciclo.setSize(20)
```

```
nbciclo.draw(win)
# desenha o cabeçalho dos clientes atendidos
ca = Text(Point(35, 750 - 100 * (n_caixas + 1)), \
          'Clientes')
ca.draw(win)
ca2 = Text(Point(35, 750 - 100 * (n_caixas + 1) + 20), \
           'atendidos')
ca2.draw(win)
# cria as filas vazias correspondentes às caixas
# e os respectivos objectos gráficos
scaixas = caixas(n_caixas, win)
win.update()
win.getMouse()

for i in range(1, ciclos+1):
    print(' == CICLO == ', i)
    nbciclo = actualiza_ciclo(nbciclo, i, win)

    # decide a criação de novo cliente e número de compras
    aleatorio = random()
    limiar = afluencia/100

    # Processa os clientes nas caixas
    trata_clientes(i, scaixas, win)

    if aleatorio < limiar: # Um novo cliente é criado
        num_compras = int(random() * 3 * apetencia) + 1
        print('--> Criado cliente com', num_compras, 'artigos')
        c = cliente(num_compras, i, win)
        # mostra temporariamente o novo cliente na janela
        icons = c.cria_icons_cliente(num_compras, \
                                      i, \
                                      1000, \
                                      50, \
                                      win)
        win.getMouse()
        # apaga o icon do novo cliente
```

```

c.apaga_cliente(Icons, win)

# insere o cliente na fila de uma caixa
nb_cx = int(random() * n_caixas) # selecciona a caixa
fila = scaixas.fila_caixa(nb_cx)
fila.coloca(c)
fila_graf = scaixas.fila_graf_caixa(nb_cx)
(x, y) = scaixas.pos_novo_icon_fila(nb_cx, fila_graf)
icons = c.cria_icons_cliente(num_compras, \
                               i, \
                               x, \
                               y, \
                               win)
fila_graf.coloca(icons)
print(scaixas)

# processa as filas depois do fecho de entradas
i = ciclos + 1
cxsfechadas = Text(Point(400, 50), 'Portas Fechadas')
cxsfechadas.setSize(20)
cxsfechadas.draw(win)
print('Entradas fechadas')
while scaixas.existem_caixas_com_fila():
    print(' == CICLO == ', i)
    nbciclo = actualiza_ciclo(nbciclo, i, win)
    # Processa os clientes nas caixas
    trata_clientes(i, scaixas, win)
    print(scaixas)
    i = i + 1

processa_resultados(scaixas, n_caixas)

```

A função `trata_clientes` é alterada com a introdução de chamadas a funções que modificam a interface gráfica.

```
def trata_clientes(tempo, caixas, win):
```

```

for i in range(caixas.num_caixas()):
    cx = caixas.fila_caixa(i)
    if not cx.fila_vazia():
        # se a fila associada à caixa i não é vazia
        # verifica se o atendimento ao cliente terminou
        cliente = cx.inicio()
        nb_artigos = cliente.artigos()
        t_atendimento = tempo - caixas.info_caixa(i)
        artigos_prcds = \
            nb_artigos/caixas.produtos_processados_ciclo(i)
        if artigos_prcds < t_atendimento:
            # o cliente sai da fila da caixa
            t_entrada_fila = cliente.tempo_entrada()
            t_espera = tempo - t_entrada_fila
            print('--> Processado cliente com', \
                  nb_artigos, 'artigos na caixa', i + 1, \
                  'tempo de espera', t_espera)
            caixas.muda_info_caixa(i, tempo, win)
            caixas.aumenta_clientes_atendidos(i, win)
            caixas.muda_info_produtos(i, nb_artigos)
            caixas.muda_info_t_espera(i, t_espera)
            # actualiza a informação gráfica
            fila_graf = caixas.fila_graf_caixa(i)
            caixas.apaga_clientes(i, win)
            cx.retira() # o cliente sai da fila
            caixas.actualiza_interface(i, win)
    else:
        # para as caixas vazias actualiza o tempo potencial
        # para atendimento
        caixas.muda_info_caixa(i, tempo, win)

```

A função `processa_resultados` não é alterada, sendo igual à apresentada na página 366.

No programa com interface gráfica surgem as funções `mostra_janela_inicial`, que mostra na janela gráfica os ícones das caixas e o número de clientes atendidos, e `actualiza_ciclo`, que actualiza na interface gráfica o número do ciclo

do simulador.

```
def mostra_janela_inicial(win, n_caixas, ciclo):
    infociclos = Text(Point(50, 50), 'Ciclo:')
    infociclos.setSize(20)
    infociclos.draw(win)
    for i in range(n_caixas):
        # desenha icon correspondente à caixa i
        icon_caixa = \
            Image(Point(100, 750 - 100 * (n_caixas - i)), \
                  'caixapq.gif')
        icon_caixa.draw(win)
    win.update()

def actualiza_ciclo(nbciclo, i, win):
    nbciclo.undraw()
    nbciclo = Text(Point(100, 50), i)
    nbciclo.setSize(20)
    nbciclo.draw(win)
    win.update()
    win.getMouse()
    return nbciclo
```

13.7 Notas finais

Apresentámos duas estruturas de informação cujos elementos apresentam uma ordem sequencial, as pilhas e as filas, bem como exemplos de aplicações que as utilizam.

Introduzimos o conceito de interface gráfica, utilizando um módulo gráfico muito simples. Existem vários módulos gráficos disponíveis para o Python, incluindo o wxPython (disponível em <http://www.wxpython.org/>) e o Tkinter (disponível a partir de <http://wiki.python.org/moin/TkInter>), os quais são mais potentes do que o módulo que apresentámos. No entanto, para a finalidade da nossa apresentação, o módulo gráfico que usámos apresenta a simplicidade e a funcionalidade necessárias.

13.8 Exercícios

1. Defina o tipo fila como entidade imutável, recorrendo a funções.
2. Uma *fila de prioridades* é uma estrutura de informação composta por um certo número de filas, cada uma das quais associada a uma determinada prioridade.

Suponha que desejava criar uma fila de prioridades com duas prioridades, urgente e normal. Nesta fila de prioridades, os novos elementos são adicionados à fila, indicando a sua prioridade, e são colocados no fim da fila respectiva. Os elementos são removidos da fila através da remoção do elemento mais antigo da fila urgente. Se a fila urgente não tiver elementos, a operação de remoção remove o elemento mais antigo da fila normal. Existe uma operação para aumentar a prioridade, a qual remove o elemento mais antigo da fila normal e coloca-o como último elemento da fila urgente.

 - (a) Especifique as operações básicas para o tipo fila de prioridades (com prioridades urgente e normal).
 - (b) Escolha uma representação interna para o tipo fila de prioridades (com prioridades urgente e normal).
 - (c) Com base na representação escolhida, escreva as operações básicas para o tipo fila de prioridades (com prioridades urgente e normal).
3. O programa de simulação do supermercado apresentado na Secção 13.5 escolhe aleatoriamente a caixa em que vai ser inserido o cliente. Isto leva a situações em que clientes são inseridos em caixas em que existe uma fila com clientes por atender num instante em que existem filas de caixa vazias. Modifique este programa de modo que um cliente é inserido na fila de caixa com menor número de clientes.
4. Em supermercados é vulgar a existência de caixas dedicadas ao atendimento de clientes com menos de 10 artigos. Modifique o programa da alínea anterior, de modo a que este conte com a existência deste tipo de caixas. Modifique também o seu algoritmo de inserção de clientes em caixas tendo em atenção este aspecto.

Capítulo 14

Árvores

*So she went on, wondering more and more at every step,
as everything turned into a tree the moment she came up
to it.*

Lewis Carroll, *Through the Looking Glass*

A árvore é um tipo estruturado de informação que é muito utilizado em programação. No dia-a-dia, utilizamos árvores para representar, por exemplo, a estrutura hierárquica de organizações e árvores genealógicas. Em informática, as árvores podem ser utilizadas para representar, entre muitas outras, estruturas de expressões (como foi feito nas figuras 1.4 a 1.7) e as funções invocadas durante a execução de um processo (como foi feito nas figuras 7.1 e 7.5).

Uma *árvore*¹ é um tipo que apresenta uma relação hierárquica entre os seus constituintes. A terminologia utilizada para designar os constituintes de uma árvore mistura termos provenientes das árvores que aparecem na natureza e termos provenientes de árvores genealógicas. Uma árvore pode ser *vazia* ou ser constituída por um elemento, a *raiz* da árvore, a qual domina, hierarquicamente, outras árvores. Uma árvore que apenas domina árvores vazias chama-se uma *folha*. As árvores dominadas chamam-se *filhas* da árvore dominadora, e esta árvore chama-se *mãe* das árvores dominadas. A ligação entre uma árvore e a árvore dominada é chamada um *ramo* da árvore.

Em programação, as árvores são normalmente escritas com a raiz no topo e com

¹Em inglês “tree”.

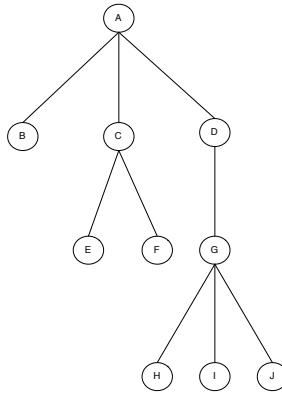


Figura 14.1: Exemplo de uma árvore.

as folhas na base. Na Figura 14.1, mostramos uma árvore cuja raiz é rotulada com A. Esta árvore domina três árvores: (1) uma árvore cuja raiz é rotulada com B e apenas domina árvores vazias; (2) uma árvore cuja raiz é rotulada com C; e (3) uma árvore cuja raiz é rotulada com D. A é a mãe de B, C e D; G é filha de D; as folhas da árvore são rotuladas com B, E, F, H, I e J.

Existe um caso particular de árvores, as *árvores binárias*, em que cada raiz domina exactamente duas árvores binárias. Uma *árvore binária* ou é vazia ou é constituída por uma raiz que domina duas árvores binárias, a árvore esquerda e a árvore direita. A árvore apresentada na Figura 14.1 não é binária, porque algumas das suas árvores (nomeadamente, as árvores cujas raízes são rotuladas com A e com G) dominam mais do que duas árvores. Na Figura 14.2, apresentamos uma árvore binária. É importante notar que a árvore cuja raiz é C domina à sua esquerda a árvore vazia e à sua direita a árvore cuja raiz é F.

Nesta secção, apenas consideramos árvores binárias. Isto não representa uma limitação séria, porque qualquer árvore pode ser transformada numa árvore binária². Daqui em diante, utilizaremos a palavra “árvore”, como sinónimo de árvore binária.

²Ver [Knuth, 1973a], Secção 2.3.2.

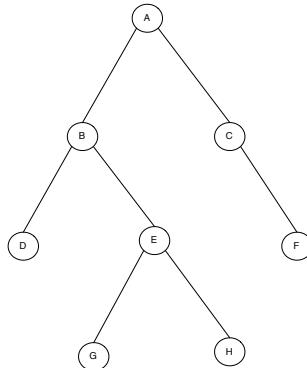


Figura 14.2: Exemplo de uma árvore binária.

14.1 Operações básicas para árvores

Apenas iremos definir árvores como entidades imutáveis. O primeiro passo para a criação do tipo árvore consiste em definir as suas operações básicas:

1. Construtores:

Os construtores para o tipo árvore incluem uma operação que gera árvores a partir do nada, à qual chamaremos *nova_arv*.

O outro construtor é uma operação que recebe como argumentos uma raiz e duas árvores, e que cria uma árvore com essa raiz cujas árvores esquerda e direita são as árvores recebidas. Esta operação será chamada *cria_arv*.

2. Selectores:

Deveremos ter um selector para escolher a raiz da árvore e selectores para a árvore esquerda e para a árvore direita. Estes selectores são chamados, respectivamente, *raiz*, *arv_esq* e *arv_dir*.

3. Reconhecedores:

A operação *árvore* tem como argumento um elemento de qualquer tipo e decide se este corresponde ou não a uma árvore.

A operação *arv_vazia* tem como argumento uma árvore e decide se esta corresponde ou não à árvore vazia (a árvore gerada por *nova_arv*).

4. Testes:

A operação *arv_iguais* tem como argumentos duas árvores e decide se estas são iguais.

Em resumo, o tipo árvore tem as seguintes operações básicas, as quais se referem ao tipo *elemento* que corresponde ao tipo dos elementos da raiz:

1. *Construtores*:

- *nova_arv* : $\{\}$ \mapsto árvore
 $nova_arv()$ tem como valor uma árvore vazia.
- *cria_arv* : *elemento* \times árvore \times árvore \mapsto árvore
 $cria_arv(raiz, a_{esq}, a_{dir})$ tem como valor a árvore com raiz *raiz*, com árvore esquerda *a_{esq}* e com árvore direita *a_{dir}*.

2. *Selectores*:

- *raiz* : árvore \mapsto *elemento*
 $raiz(\text{árv})$ recebe uma árvore, *árv*, e tem como valor a sua raiz. Se a árvore for vazia, o valor desta operação é indefinido.
- *arv_esq* : árvore \mapsto árvore
 $arv_esq(\text{árv})$ recebe uma árvore, *árv*, e tem como valor a sua árvore esquerda. Se a árvore for vazia, o valor desta operação é indefinido.
- *arv_dir* : árvore \mapsto árvore
 $arv_dir(\text{árv})$ recebe uma árvore, *árv*, e tem como valor a sua árvore direita. Se a árvore for vazia, o valor desta operação é indefinido.

3. *Reconhecedores*:

- *arv* : universal \mapsto lógico
 $arv(arg)$ tem o valor *verdadeiro* se *arg* é uma árvore e tem o valor *falso*, em caso contrário.
- *arv_vazia* : árvore \mapsto lógico
 $arv_vazia(\text{árv})$ tem o valor *verdadeiro* se *árv* é uma árvore vazia e tem o valor *falso*, em caso contrário.

4. *Testes*:

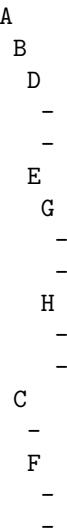


Figura 14.3: Representação externa da árvore da Figura 14.2.

- $arv_iguais : árvore \times árvore \mapsto lógico$

$arv_iguais(árv_1, árv_2)$ tem o valor *verdadeiro* se $árv_1$ e $árv_2$ são árvores iguais e tem o valor *falso*, em caso contrário.

De modo a utilizarmos o tipo árvore temos ainda de definir os transformadores de entrada e de saída, os quais transformam entre a representação que utilizamos para árvores, a representação externa, e a representação utilizada no nosso programa³. Seria interessante utilizar como *representação externa* para árvores uma representação gráfica como a que apresentamos nas figuras 14.1 e 14.2. Para isso, poderemos utilizar as funções gráficas apresentadas na Secção 13.6. Por uma questão de simplicidade, optamos por uma representação externa para árvores, tal como a que apresentamos na Figura 14.3, a qual corresponde à árvore da Figura 14.2. A nossa representação externa utiliza as seguintes convenções: (1) a árvore vazia é representada por “–”; (2) a árvore esquerda e a árvore direita de uma árvore são escritas em linhas diferentes (começando à mesma distância da margem esquerda), e dois espaços mais para a direita do que a sua raiz, que aparece na linha acima. A operação *escreve_arv* recebe uma árvore e mostra-a de acordo com esta convenção.

³Novamente, não iremos definir o transformador de entrada.

14.2 Axiomatização

Entre as operações básicas devemos impor as seguintes condições (axiomatização), em que a , a_1 e a_2 são árvores e r é um elemento da raiz de uma árvore⁴:

$$\begin{aligned}
 arv(nova_arv()) &= verdadeiro \\
 arv(cria_arv(r, a_1, a_2)) &= verdadeiro \\
 arv_vazia(nova_arv()) &= verdadeiro \\
 arv_vazia(cria_arv(r, a_1, a_2)) &= falso \\
 raiz(cria_arv(r, a_1, a_2)) &= r \\
 arv_esq(cria_arv(r, a_1, a_2)) &= a_1 \\
 arv_dir(cria_arv(r, a_1, a_2)) &= a_2 \\
 cria_arv(raiz(a), arv_esq(a), arv_dir(a)) &= a \\
 arv_iguais(a, cria_arv(raiz(a), arv_esq(a), arv_dir(a))) &= verdadeiro
 \end{aligned}$$

14.3 Representação de árvores

A escolha de uma representação interna para árvores vai ser influenciada pelo tipo de programa que queremos utilizar para o tipo árvore. Podemos pensar em realizar o tipo árvore usando funções ou podemos realizar o tipo árvore como objectos.

14.3.1 Representação para o uso recorrendo a funções

Criar o tipo árvore recorrendo a funções, precisamos que as árvores estejam contidas numa única estrutura de informação, a qual será passada às várias funções que manipulam árvores. Uma das possíveis representações internas para árvores recorre a listas e poderá ser definida do seguinte modo (Figura 14.4):

1. Uma árvore vazia é representada lista vazia $[]$.

⁴Por uma questão de simplicidade, nas duas últimas equações omitimos a verificação de árvore não vazia.

$$\mathfrak{R}[\text{Árvore vazia}] = []$$

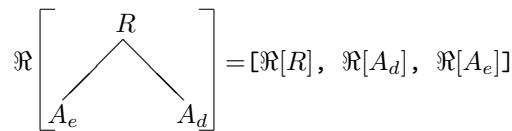


Figura 14.4: Representação de árvores utilizando listas.

2. Uma árvore não vazia é representada por uma lista cujo primeiro elemento contém a raiz da árvore, cujo segundo elemento contém a representação da árvore esquerda e cujo terceiro elemento contém a representação da árvore direita.

Por exemplo, a árvore da Figura 14.2, será representada pela lista [A, [B, [D, [], []], [E, [G, [], []], [H, [], []]]], [C, [], [F, [], []]]], ou, de uma forma mais intuitiva, baseando-nos no princípio que utilizámos para a representação externa:

```
[A,
[B,
[D, [], []],
[E,
[G, [], []],
[H, [], []]]],
[C,
[], 
[F, [], []]]]
```

14.3.2 Representação para o uso recorrendo a objectos

Sabemos que um objecto é uma entidade com estado interno. Representado as árvores como objectos, o estado interno desses objectos deverá conter os diferentes constituintes de uma árvore. Neste caso, podemos utilizar três componentes

$$\begin{aligned} \Re[\text{Árvore vazia}] &= \begin{array}{l} \mathbf{self.r : None} \\ \mathbf{self.e : } \perp \\ \mathbf{self.d : } \perp \end{array} \end{aligned}$$

$$\Re \left[\begin{array}{c} R \\ / \quad \backslash \\ A_e \quad A_d \end{array} \right] = \begin{array}{l} \mathbf{self.r : } \Re[R] \\ \mathbf{self.e : } \Re[A_e] \\ \mathbf{self.d : } \Re[A_d] \end{array}$$

Figura 14.5: Representação de árvores recorrendo a objectos.

distintos para representar os três constituintes de uma árvore, a raiz, a árvore esquerda e a árvore direita. Embora estas três entidades sejam distintas, elas existem dentro das instâncias da classe árvore. Assim, uma árvore será representada por três variáveis, `self.r`, `self.e` e `self.d`, contendo, respectivamente, a raiz, a árvore esquerda e a árvore direita.

Para representar uma árvore vazia, vamos recorrer a um tipo de informação elementar existente em Python, o tipo `None`. O tipo `None` tem apenas uma constante, `None`, a qual representa “nada”. Já tínhamos apresentado este tipo, embora não explicitamente, quando na página 58 dissemos que a função `print` não devolve qualquer valor. A representação externa desta constante não é mostrada pelo Python quando uma expressão tem o valor `None` como o mostra a seguinte interacção:

```
>>> None
>>>
```

Assim, a representação que escolhemos para árvores é apresentada na Figura 14.5 (recorda-se que \perp representa indefinido).

14.4 Realização das operações básicas

Apresentamos a realização das operações básicas para as duas alternativas apresentadas na Secção 14.3.

14.4.1 Árvores recorrendo a funções

```
def nova_arv():
    return []

def cria_arv(r, a_e, a_d):
    if arvore(a_e) and arvore(a_d):
        return [r, a_e, a_d]
    else:
        raise ValueError ('cria_arv: o segundo e terceiro \
argumentos devem ser árvores')

def raiz(a):
    if a == []:
        raise ValueError ('raiz: a árvore é vazia')
    else:
        return a[0]

def arb_esq(a):
    if a == []:
        raise ValueError ('arb_esq: a árvore é vazia')
    else:
        return a[1]

def arb_dir(a):
    if a == []:
        raise ValueError ('arb_dir: a árvore é vazia')
    else:
        return a[2]

def arvore(x):
    if isinstance(x, list):
        if x == []:
            ...
```

```
        return True
    elif len(x) == 3 and \
        arvore(x[1]) and \
        arvore(x[2]):
        return True
    else:
        return False
else:
    return False

def arv_vazia(a):
    return a == []

def arv_iguais(a1, a2):
    if arv_vazia(a1):
        return arv_vazia(a2)
    elif raiz(a1) == raiz(a2):
        return arv_iguais(arv_esq(a1), arv_esq(a2)) and \
               arv_iguais(arv_dir(a1), arv_dir(a2))
    else:
        return False

def escreve_arv(a):

    def escreve_aux(a, indent):
        if arv_vazia(a):
            print(' ' * indent, '-')
        else:
            print(' ' * indent, raiz(a))
            escreve_aux(arv_esq(a), indent + 2)
            escreve_aux(arv_dir(a), indent + 2)

    escreve_aux(a, 0)
```

14.4.2 A classe árvore

Recordemos que definimos dois construtores para árvores, a operação *nova_arv* o que gera árvores a partir do nada e a operação *cria_arv* que recebe como argumentos uma raiz e duas árvores, e que cria uma árvore com essa raiz cujas árvores esquerda e direita são as árvores recebidas. Existem pois dois modos distintos de construir árvores, modos esses que devem ser contemplados pelo método `__init__`: se não lhe forem fornecidos argumentos, então `__init__` deve construir uma árvore vazia; se lhe forem fornecidos uma raiz e duas árvores, então `__init__` constrói uma árvore não vazia. Precisamos pois de definir uma função, `__init__`, que recebe um número variável de argumentos.

A função embutida `print`, que temos utilizado nos nossos programas, aceita qualquer número de argumentos. A execução de `print()` origina uma linha em branco, e a execução de `print` com qualquer número de argumentos origina a escrita dos valores dos seus argumentos seguidos de um salto de linha. Até agora, as funções que escrevemos têm um número fixo de argumentos, o que contrasta com algumas das funções embutidas do Python, por exemplo, a função `print`.

Antes de apresentar a classe `arvore`, vamos considerar um novo aspecto na definição de funções correspondente à utilização de um número arbitrário de elementos. O Python permite a definição de funções que aceitam um número arbitrário de argumentos. Para podermos utilizar este aspecto, teremos que rever a definição de função apresentada na página 76. Uma forma mais completa da definição de funções em Python é dada pelas seguintes expressões em notação BNF:

```
<definição de função> ::= def <nome> (<parâmetros formais>): CR
                           TAB <corpo>
```

```
<parâmetros formais> ::= <nada>{*<nome>} | <nomes>{, *<nome>}
```

A diferença desta definição em relação à definição apresentada na página 76, corresponde à possibilidade dos parâmetros formais terminarem com um nome que é antecedido por `*`. Por exemplo, esta definição autoriza-nos a criar uma função em Python cuja primeira linha corresponde a “`def ex_fn(a, b, *c):`”.

Ao encontrar uma chamada a uma função cujo último parâmetro formal seja antecedido por um asterisco, o Python associa os $n - 1$ primeiros parâmetros formais com os $n - 1$ primeiros parâmetros concretos e associa o último parâmetro

formal (o nome que é antecedido por um asterisco) com o tuplo constituído pelos restantes parâmetros concretos. A partir daqui, a execução da função segue os mesmos passos que anteriormente.

Consideremos a seguinte definição de função:

```
def ex_fn(a, b, *c):
    print('a =', a)
    print('b =', b)
    print('c =', c)
```

Esta função permite gerar a seguinte interacção, a qual revela este novo aspecto da definição de funções:

```
>>> ex_fn(3, 4)
a = 3
b = 4
c = ()
>>> ex_fn(5, 7, 9, 1, 2, 5)
a = 5
b = 7
c = (9, 1, 2, 5)
>>> ex_fn(2)
TypeError: ex_fn() takes at least 2 arguments (1 given)
```

A função `ex_fn` aceita, no mínimo, dois argumentos. Os dois primeiros parâmetros concretos são associados com os parâmetros formais `a` e `b` e tuplo contendo os restantes parâmetros concretos é associado ao parâmetro formal `c`.

Deste modo, o método `__init__` para a classe `arvore` deve permitir a utilização de zero ou três argumentos.

```
class arvore:

    def __init__(self, *args):
        if args == ():
            self.r = None
        else:
```

```
if len(args) == 3:
    if isinstance(args[1], arvore) and \
       isinstance(args[2], arvore):
        self.r = args[0]
        self.e = args[1]
        self.d = args[2]
    else:
        raise ValueError ('arvore: o segundo e \
                           terceiro argumentos devem ser arvores')
    else:
        raise ValueError ('arvore: aceita zero ou \
                           tres argumentos')

def raiz(self):
    if self.r == None:
        raise ValueError ('raiz: a arvore é vazia')
    else:
        return self.r

def arv_esq(self):
    if self.r == None:
        raise ValueError ('arv_esq: a arvore é vazia')
    else:
        return self.e

def arv_dir(self):
    if self.r == None:
        raise ValueError ('arv_dir: a arvore é vazia')
    else:
        return self.d

def arv_vazia(self):
    return self.r == None
```

```

def __repr__(self):

    def __repr__aux(a, indent):
        if a.r == None:
            return ' ' * indent + '-' + '\n'
        else:
            return ' ' * indent + \
                   '[' + str(a.r) + '\n' + \
                   __repr__aux(a.e, indent + 2) + \
                   __repr__aux(a.d, indent + 2) + \
                   ' ' * (indent + 2) + ']' + '\n'

    return __repr__aux(self, 0)

```

14.5 Ordenação por árvore

Nesta secção, apresentamos uma aplicação que utiliza árvores, a ordenação por árvore. A *ordenação por árvore* é um algoritmo de ordenação muito eficiente, executado em dois passos sequenciais: primeiro coloca os elementos a serem ordenados numa árvore binária (chamada *árvore binária de procura*), depois percorre esta árvore seguindo um método determinado, “visitando” as raízes das árvores que a constituem. Isto é feito de modo a que a primeira raiz visitada contenha o menor elemento, a segunda raiz visitada, o elemento que é imediatamente maior, e assim sucessivamente.

A árvore binária é construída do seguinte modo. Começando com uma árvore vazia, inserimos nesta árvore os elementos a serem ordenados, um de cada vez.

1. Um elemento é inserido numa árvore vazia através da criação de uma árvore cuja raiz contém o elemento a inserir e em que as árvores esquerda e direita são vazias.
2. Um elemento é inserido numa árvore não vazia comparando o elemento com a raiz da árvore. Se o elemento a ser inserido for maior do que o elemento que se encontra na raiz da árvore, o elemento é inserido, utilizando o mesmo método, na árvore direita da árvore inicial, caso contrário, é inserido, pelo mesmo método, na árvore esquerda.

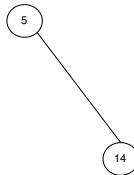


Figura 14.6: Árvore depois da inserção de 5 e de 14.

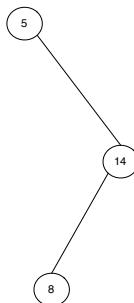


Figura 14.7: Árvore depois da inserção de 8.

Como exemplo desta função, suponhamos que desejávamos criar uma árvore binária de procura com os elementos 5, 14, 8, 2 e 20.

Em primeiro lugar, criamos uma árvore vazia, onde inserimos o elemento 5, criando uma árvore cuja raiz contém 5 e em que as árvores esquerda e direita são vazias. Seguidamente, inserimos o elemento 14 nesta árvore. Uma vez que $14 > 5$ (5 é a raiz da árvore), 14 é inserido na árvore da direita, originando a árvore apresentada na Figura 14.6. O elemento seguinte a ser inserido é 8. Uma vez que $8 > 5$, este elemento será introduzido na árvore da direita. Para inserir este elemento na árvore da direita, este é comparado com a raiz, e, uma vez que $8 < 14$, este é inserido na árvore da esquerda, dando origem à árvore apresentada na Figura 14.7. Em seguida, insere-se o número 2. Uma vez que $2 < 5$, este elemento será inserido na árvore da esquerda, dando origem à árvore apresentada na Figura 14.8. Finalmente, inserimos o elemento 20, dando origem à árvore apresentada na Figura 14.9.

Uma vez construída a árvore, teremos de a percorrer, visitando as raízes das suas árvores. As raízes devem ser visitadas por ordem crescente dos elementos existentes na árvore.

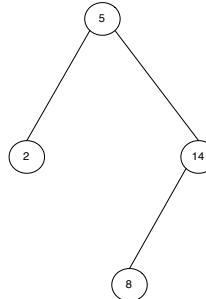


Figura 14.8: Árvore depois da inserção de 2.

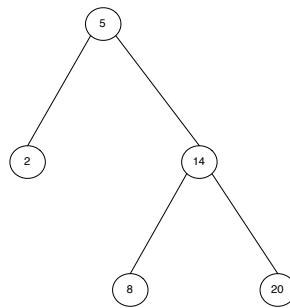


Figura 14.9: Árvore depois da inserção de 20.

O algoritmo para percorrer a árvore é o seguinte:

1. Percorrer uma árvore vazia não causa nenhuma ação.
2. Para percorrer uma árvore não vazia, primeiro percorremos a sua árvore da esquerda, depois visitamos a raiz, depois percorremos a sua árvore da direita.

Por exemplo, considerando a árvore da Figura 14.9, uma vez que esta árvore não é vazia, teremos primeiro de percorrer a sua árvore da esquerda, depois visitar a raiz e depois percorrer a árvore da direita. Para percorrer a árvore da esquerda (cuja raiz contém 2), teremos de percorrer a árvore da esquerda (que é vazia), visitar a raiz, que contém o valor 2, e depois percorrer a árvore da direita, que também é vazia. Acabamos assim de percorrer a árvore cuja raiz contém 2, visitamos agora a raiz da árvore original, que contém 5, e percorremos a árvore da direita. O processo repete-se para a árvore da direita. Deixamos como

exercício a verificação de que os elementos são visitados por ordem crescente do seu valor.

A função `ordena_arvore` recebe uma lista contendo valores a ordenar e devolve a lista correspondente à ordenação dos elementos recebidos. Esta função utiliza a definição de árvores como objectos.

```
def ordena_arvore(lst):
    return percorre(lista_para_arvore(lst))

def lista_para_arvore(lst):

    def insere_arv(lst, arv):
        def insere_elemento(el, arv):
            if arv.arv_vazia():
                return arvore(el, arvore(), arvore())
            elif el > arv.raiz():
                return arvore(arv.raiz(), \
                             arv.arv_esq(), \
                             insere_elemento(el, arv.arv_dir()))
            else:
                return arvore(arv.raiz(), \
                             insere_elemento(el, arv.arv_esq()), \
                             arv.arv_dir())
        if lst == []:
            return arv
        else:
            return insere_arv(lst[1:], \
                              insere_elemento(lst[0], arv))

    return insere_arv(lst, arvore())

def percorre(arv):
```

```
if arv.arv_vazia():
    return []
else:
    return percorre(arv.arv_esq()) + \
        [arv.raiz()] + \
        percorre(arv.arv_dir())
```

Com a qual podemos gerar a interacção:

```
>>> ordena_arvore([5, 14, 8, 2, 20])
[2, 5, 8, 14, 20]
```

14.6 Notas finais

Apresentámos o tipo árvore que é muito utilizado em programação. Informação adicional sobre árvores e algoritmos que as manipulam podem ser consultados em [Knuth, 1973b] e em [Cormen et al., 2009].

14.7 Exercícios

1. Defina a representação externa para árvores utilizando uma interface gráfica.

Capítulo 15

Ponteiros

Here's a path that leads straight to it – at least, no, it doesn't do that – ... but I suppose it will at last. But how curiously it twists!

Lewis Carroll, *Through the Looking Glass*

O Python apresenta estruturas de informação que são dinâmicas, no sentido em que o seu número de elementos pode aumentar ou diminuir durante a execução do programa. São exemplos destas estruturas as listas e os dicionários. Grande parte das linguagens de programação não apresenta esta característica, oferecendo apenas estruturas estáticas, ou seja, estruturas de informação em relação às quais o número de elementos é determinado no instante da sua criação, não se podendo remover nem adicionar elementos a estas estruturas. As operações disponíveis para estas estruturas limitam-se ao acesso aos seus elementos e à alteração dos valores dos seus elementos. As linguagens de programação que apenas apresentam estruturas de informação estáticas, oferecem um tipo de informação chamado ponteiro, o qual é usado para a criação de estruturas dinâmicas.

Embora o tipo ponteiro não exista explicitamente em Python, este é usado implicitamente na manipulação das estruturas de informação dinâmicas. Neste capítulo, discutimos as características do tipo ponteiro, simulando a sua realização em Python. Os ponteiros não têm grande interesse por si só (na realidade nem são sequer um tipo, na verdadeira acepção da palavra), mas tornam-se

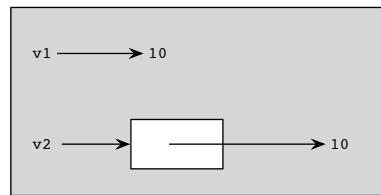


Figura 15.1: Nomes associados a variáveis e a ponteiros.

muito importantes quando são considerados como elos de ligação entre os vários componentes de uma estrutura de informação.

15.1 A noção de ponteiro

Um *ponteiro* é qualquer coisa que aponta. Em programação, uma entidade do tipo ponteiro “aponta” para uma outra entidade computacional. Ao passo que com as variáveis que temos utilizado até agora, estamos interessados no valor da variável, com uma variável correspondente a um ponteiro não estamos interessados no valor da variável mas sim no valor para onde ela aponta.

Ao trabalhar com ponteiros é útil recorrer a uma representação gráfica. Para isso, representaremos uma variável do tipo ponteiro como um rectângulo do qual sai uma seta que aponta para o valor da variável. Na figura 15.1, mostramos a representação de uma variável com o nome *v1* e cujo valor é 10 e de uma variável com o nome *v2*, correspondente a um ponteiro, que aponta para o valor 10. Repare-se que *v2* corresponde a uma variável que aponta para uma outra variável, que contém o valor que nos interessa considerar. Esta segunda variável não tem um nome directamente associado, sendo designada por variável anónima. Uma *variável anónima* (também conhecida por *variável referenciada*¹) é uma variável cujo valor não é obtido directamente através do seu nome, mas sim através de uma entidade correspondente a um ponteiro.

Na Figura 15.1, é também evidente a semelhança entre a notação que introduzimos para ponteiros e a notação que temos vindo a utilizar para, num ambiente, associar o nome de uma variável ao seu valor. Na realidade, a associação do nome de uma variável ao seu valor é feito implicitamente recorrendo a pontei-

¹Do inglês, “referenced variable”.

ros, o que explica o funcionamento da passagem de parâmetros por referência apresentado na Secção 5.2.2.

A importância dos ponteiros revela-se quando estes são utilizados como elos de ligação entre os vários elementos de uma estrutura de informação. Para apresentar a utilidade dos ponteiros, iremos definir um tipo a que chamamos nó. Um *nó* agrupa duas entidades, um valor e um ponteiro. Existem selectores, *val* e *prox*, que acedem a cada uma destas entidades e existem modificadores, *muda_val* e *muda_prox* que permitem modificar cada uma destas entidades. Utilizamos a constante `None` para indicar que um ponteiro não aponta para nenhuma entidade.

O tipo *nó* é realizado através da classe `nó` do seguinte modo:

```
class nó:

    def __init__(self, val):
        self.v = val
        self.p = None

    def val(self):
        return self.v

    def prox(self):
        return self.p

    def muda_val(self, val):
        self.v = val

    def muda_prox(self, prox):
        self.p = prox

    def __repr__(self):
        return '[' + str(self.v) + ' | ' + self.p.__repr__() + ']'
```

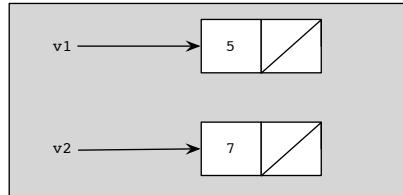


Figura 15.2: Valores iniciais de v1 e v2.

Consideremos agora a seguinte interacção (representamos externamente um nó, com os seus dois componentes entre parêntesis rectos, separados por uma barra vertical; o primeiro componente contém o valor associado ao nó e o segundo componente contém a entidade para onde o ponteiro do nó aponta):

```
>>> v1 = no(5)
>>> v1
[5 | None]
>>> v2 = no(7)
>>> v2
[7 | None]
>>> v2.muda_prox(v1)
>>> v2
[7 | [5 | None]]
```

Na Figura 15.2 mostramos os valores iniciais das variáveis `v1` e `v2` (originados pelas instruções `v1 = no(5)` e `v2 = no(7)`), usando a convenção que um rectângulo com uma diagonal a cheio corresponde à representação de um ponteiro cujo valor é `None`. A instrução `v2.muda_prox(v1)` altera o valor do ponteiro associado a `v2` de modo a que este aponte para `v1`, dado origem ao ambiente apresentado na Figura 15.3. Repare-se que ao executar esta instrução, a estrutura associada à variável `v2` “cresceu”, com a adição do nó correspondente a `v1`.

15.2 Listas ligadas

Com base no tipo `nó` da secção anterior, vamos discutir o modo de criar listas em linguagens de programação que não apresentam estruturas dinâmicas.

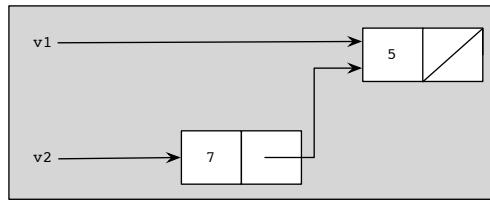


Figura 15.3: Valores de `v1` e `v2` após a execução de `v2.muda_prox(v1)`.

Vulgarmente, nestas linguagens o tipo lista é designado por *lista ligada* para enfatizar que os seus elementos estão ligados entre si através de ponteiros.

Iremos considerar as seguintes operações básicas para o tipo lista² (estas operações referem o tipo *elemento*, que corresponde ao tipo dos elementos da lista):

1. *Construtores:*

- *nova_lista* : $\{\}$ \mapsto *lista*
 $nova_lista()$ tem como valor uma lista sem elementos (a lista vazia).
- *lista* : *elemento*^{*n*} \mapsto *lista*
 $lista(e_1, \dots, e_n)$ recebe *n* elementos e tem como valor a lista com esses elementos, pela ordem em que foram escritos.

2. *Selectores:*

- *elem_pos* : *lista* \times \mathbb{N}_0 \mapsto *elemento*
 $elem_pos(lst, n)$ tem como valor o elemento que se encontra na *n*-ésima posição da lista *lst* (o valor 0 corresponde ao primeiro elemento da lista). Se a lista tiver menos do que *n* elementos, o valor desta operação é indefinido.
- *comprimento* : *lista* \mapsto \mathbb{N}_0
 $comprimento(lst)$ tem como valor o número de elementos da lista *lst*.

3. *Modificadores:*

- *insere_pos* : *lista* \times *elemento* \times \mathbb{N}_0 \mapsto *lista*

²Estas operações básicas foram influenciadas pelas listas existentes em Python. É possível definir outros conjuntos de operações básicas para o tipo lista (ver [Martins e Cravo, 2007]).

insere_pos(lst, elm, n) altera de forma permanente a lista *lst* para a lista que resulta de inserir o elemento *elm* na *n*-ésima posição de *lst*. Se *lst* tiver menos do que *n* elementos, o valor desta operação é indefinido.

— *remove_pos : lista × N₀ ↪ lista*

remove_pos(lst, n) altera de forma permanente a lista *lst* para a lista que resulta de remover o *n*-ésimo elemento da lista *lst*. Se *lst* tiver menos do que *n* elementos, o valor desta operação é indefinido.

— *muda_elem_pos : lista × N₀ × elem ↪ lista*

muda_elem_pos(lst, n, elm) altera de forma permanente o elemento na *n*-ésima posição da lista *lst* para o valor *elm*. Se *lst* tiver menos do que *n* elementos, o valor desta operação é indefinido.

4. Reconhecedores:

— *lista : universal ↪ lógico*

lista(arg) tem o valor *verdadeiro*, se *arg* é uma lista, e tem o valor *falso*, em caso contrário.

— *lista_vazia : lista ↪ lógico*

lista_vazia(lst) tem o valor *verdadeiro*, se *lst* é a lista vazia, e tem o valor *falso*, em caso contrário.

5. Testes:

— *em : lista × elemento ↪ lógico*

em(lst, elm) tem o valor *verdadeiro*, se *elm* pertence à lista *lst*, e tem o valor *falso*, em caso contrário.

— *listas_iguais : lista × lista ↪ lógico*

listas_iguais(lst₁, lst₂) tem o valor *verdadeiro*, se *lst₁* e *lst₂* têm os mesmos elementos e pela mesma ordem, e tem o valor *falso*, em caso contrário.

Vamos definir as listas ligadas como objectos. A classe correspondente às listas ligadas tem duas variáveis de estado: (1) **self.lst**, a qual contém a representação da lista, sendo esta representação **None**, no caso da lista ser vazia, ou um conjunto de *nós* em que cada nó contém um elemento da lista e um ponteiro para o próximo elemento da lista; (2) **self.comp**, a qual contém um inteiro que representa o número de elementos da lista. A decisão de utilizar a variável de

estado `self.comp` está associada a uma questão de eficiência na manipulação de listas.

A representação externa de listas ligadas é semelhante à das listas do Python, representando-se os elementos dentro de parêntesis rectos, separados por vírgulas.

O método `__init__` desta classe aceita um número arbitrário de elementos. No caso do número de argumentos deste método ser superior a zero, cria a estrutura da lista resultante utilizando um método semelhante ao apresentado na página 410; se o número de argumentos for zero, cria a lista vazia:

```
def __init__(self, *elmts):

    def cria_lista(elmts):
        res = None
        for i in range(len(elmts)-1, -1, -1):
            novo_el = no(elmts[i])
            novo_el.muda_prox(res)
            res = novo_el
        return res

        self.comp = len(elmts)
        if self.comp == 0:
            self.lst = None
        else:
            self.lst = cria_lista(elmts)
```

O método `elem_pos`, recebe uma posição da lista, `pos`, e segue a cadeia de ponteiros na representação interna da lista, “passando por cima” de `pos - 1` elementos, devolvendo o valor associado ao `pos`-ésimo elemento da lista. Neste método, é evidente a vantagem da utilização da variável de estado `self.comp`.

```
def elem_pos(self, pos):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
```

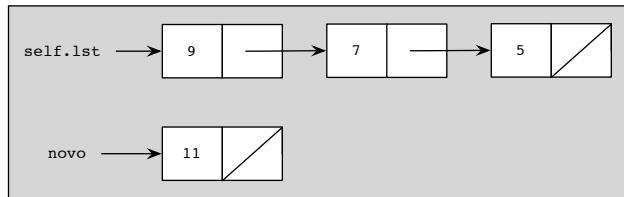


Figura 15.4: O novo elemento da lista é criado.

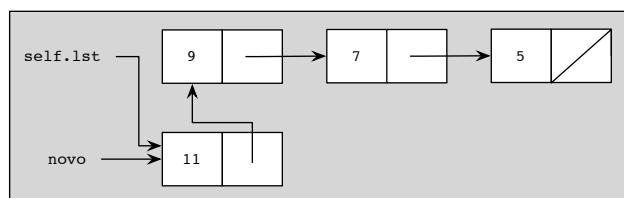


Figura 15.5: O novo elemento passa a ser o primeiro elemento da lista.

```

no_el = self.lst
for i in range(pos):
    no_el = no_el.prox()
return no_el.val()

```

O método `insere_pos` precisa de alguma explicação adicional. Suponhamos que desejávamos inserir o elemento 11 na primeira posição da lista [9, 7, 5]. O primeiro passo nesta operação corresponde à criação do nó [11 | `None`], designado por `novo` na Figura 15.4. Como este nó passará a ser o primeiro elemento da lista resultante, teremos que alterar o ponteiro existente em `novo.prox()` para apontar para o (antigo) primeiro elemento da lista, o qual é dado pelo valor da variável `self.lst`. Seguidamente, o valor da variável `self.lst` deverá passar a ter o valor de `novo` (Figura 15.5).

Suponhamos agora que desejávamos inserir o elemento 11 na terceira posição (`pos = 2`) da lista [9, 7, 5]. Para além de criarmos o nó [11 | `None`], designado por `novo`, como fizémos anteriormente, vamos também inicializar uma variável, a que chamamos `actual`, cujo valor corresponde ao primeiro elemento da lista (`actual = self.lst`), como se mostra na Figura 15.6. A variável `actual` vai ser usada para determinar o elemento da lista após o qual será inserido o novo elemento. Esta posição de inserção é feita através de um ciclo `for`.

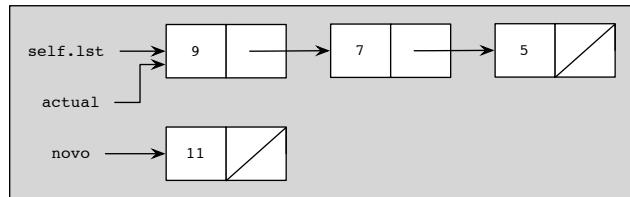
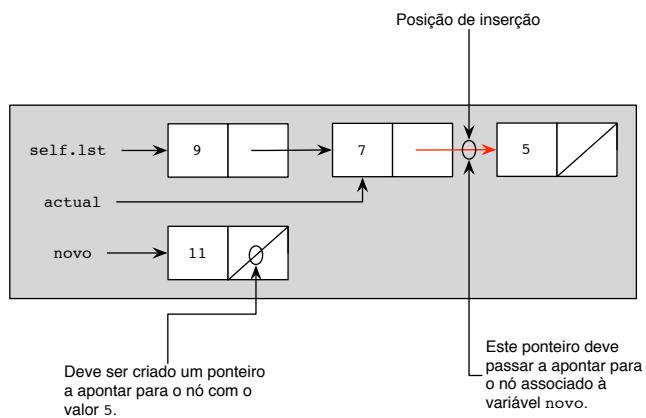
Figura 15.6: Criação do novo nó e inicialização de `actual`.

Figura 15.7: Identificação da posição de inserção.

Uma vez determinado o local de inserção do novo elemento, teremos que fazer a actualização de dois ponteiros, como se mostra na Figura 15.7: o ponteiro associado ao nó `novo` deverá apontar para o elemento depois do ponto de inserção e o ponteiro associado ao elemento antes da inserção deverá apontar para o nó `novo`. Na Figura 15.8 apresentamos a lista resultante da inserção do elemento. Seguindo os ponteiros a partir da variável de estado `self.lst`, obtemos a lista [9, 7, 11, 5].

```
def insere_pos(self, el, pos):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
```

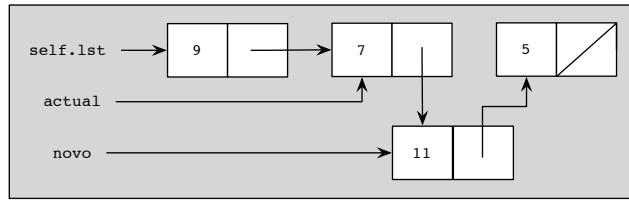


Figura 15.8: Resultado da inserção do elemento.

```

novo = no(el)
if pos == 0:
    novo.muda_prox(self.lst)
    self.lst = novo
else:
    actual = self.lst
    proximo = actual.prox()
    for i in range(pos-1):
        # procura da posição de inserção
        actual = actual.prox()
        proximo = actual.prox()
    # o elemento é inserido
    actual.muda_prox(novo)
    novo.muda_prox(proximo)
    self.comp = self.comp + 1

```

Consideremos agora o método `remove_pos`. Suponhamos que desejávamos remover o segundo elemento (`pos = 1`) da lista `[9, 7, 5]`. Para isso, e à semelhança do que fizemos no método `insere_pos`, usamos uma variável, a que chamamos `actual`, para determinar a posição do elemento da lista a remover. A posição de remoção é determinada através de um ciclo `for`. Na Figura 15.9 apresentamos a situação após a identificação do elemento a remover.

Teremos agora que alterar apenas um ponteiro: o ponteiro que aponta para o elemento a remover, deve ser alterado de modo a que este aponte para o elemento referenciado no ponteiro do elemento a remover, dando origem à situação apresentada na Figura 15.10.

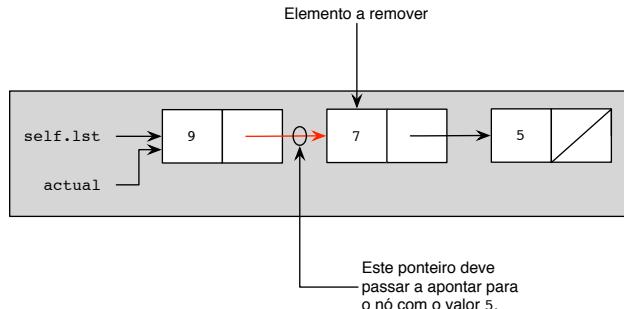


Figura 15.9: O elemento a remover foi identificado.

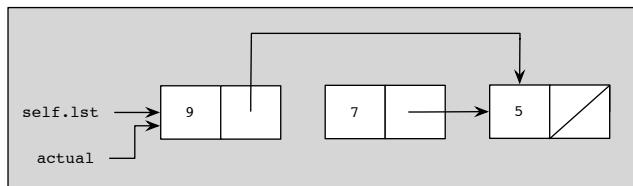


Figura 15.10: O novo elemento passa a ser o primeiro elemento da lista.

Seguindo os ponteiros a partir da variável de estado `self.lst`, obtemos a lista `[9, 5]`. Notemos que o nó associado ao valor 7 ainda aponta para o nó `[5 | None]`, mas já não faz parte da lista. O nó associado ao valor 7 existe na memória do computador mas não pode ser acedido pelo nosso programa uma vez que não tem um nome associado nem nenhum ponteiro aponta para ele. Em termos de programação, este nó é designado por *lixo*. Na Secção 15.3 apresentamos técnicas para lidar com estas situações.

```
def remove_pos(self, pos):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
        if pos == 0:
            self.lst = self.lst.prox()
        else:
            actual = self.lst
            for i in range(pos - 1):
```

```

        actual = actual.prox()
        actual.muda_prox(actual.prox().prox())
    self.comp = self.comp - 1

```

O método `muda_elem_pos` segue, usando um algoritmo semelhante ao do método `elem_pos`, a cadeia de ponteiros na representação interna da lista, “passando por cima” de `pos - 1` elementos e alterando o valor associado ao seguinte nó da lista.

A classe `lista` é definida pelo seguinte modo:

```

class lista:

    def __init__(self, *elmts):

        def cria_lista(elmts):
            res = None
            for i in range(len(elmts)-1, -1, -1):
                novo_el = no(elmts[i])
                novo_el.muda_prox(res)
                res = novo_el
            return res

            self.comp = len(elmts)
            if self.comp == 0:
                self.lst = None
            else:
                self.lst = cria_lista(elmts)

    def elem_pos(self, pos):
        if pos > self.comp:
            raise ValueError \
                ('A lista não tem elementos suficientes')
        else:
            no_el = self.lst
            for i in range(pos):
                no_el = no_el.prox()

```

```
    return no_el.val()

def comprimento(self):
    return self.comp

def insere_pos(self, el, pos):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
        novo = no(el)
        if pos == 0:
            novo.muda_prox(self.lst)
            self.lst = novo
        else:
            actual = self.lst
            proximo = actual.prox()
            for i in range(pos-1):
                # procura da posição de inserção
                actual = actual.prox()
                proximo = actual.prox()
            # o elemento é inserido
            actual.muda_prox(novo)
            novo.muda_prox(proximo)
        self.comp = self.comp + 1

def remove_pos(self, pos):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
        if pos == 0:
            self.lst = self.lst.prox()
        else:
            actual = self.lst
```

```
        for i in range(pos - 1):
            actual = actual.prox()
            actual.muda_prox(actual.prox().prox())
        self.comp = self.comp - 1

def muda_elem_pos(self, pos, val):
    if pos > self.comp:
        raise ValueError \
            ('A lista não tem elementos suficientes')
    else:
        no_el = self.lst
        for i in range(pos):
            no_el = no_el.prox()
        no_el.muda_val(val)

def lista_vazia(self):
    return lst.comp == 0

def em(self, el):
    el_lst = self.lst
    for i in range(self.comp):
        if el_lst.val() == el:
            return True
        el_lst = el_lst.prox()
    return False

def __eq__(self, outra):
    if self.comp == outra.comprimento():
        l1 = self.lst
        for i in range(self.comp):
            if l1.val() != outra.elem_pos(i):
                return False
            l1 = l1.prox()
        return True
    else:
        return False
```

```
def __repr__(self):
    res = '['
    el = self.lst
    for i in range(self.comp - 1):
        res = res + str(el.val()) + ', '
        el = el.prox()
    res = res + str(el.val()) + ']'
    return res
```

Com a classe `lista` podemos gerar a seguinte interacção:

```
>>> l1 = lista(1, 2, 3, 4)
>>> l1
[1, 2, 3, 4]
>>> l1.elem_pos(2)
3
>>> l1.remove_pos(2)
>>> l1
[1, 2, 4]
>>> l1.muda_elem_pos(0, 5)
>>> l1
[5, 2, 4]
>>> l1.em(6)
False
```

15.3 A gestão de memória

Numa linguagem de programação em que exista o tipo ponteiro, as variáveis associadas a ponteiros podem ser criadas e destruídas dinamicamente em qualquer ponto da execução de um programa. Nesta secção, apresentamos uma discussão sobre técnicas utilizadas para a destruição de variáveis associadas a ponteiros.

Qualquer linguagem de programação que permita a utilização de ponteiros mantém uma zona da memória do computador destinada a fornecer o armazenamento necessário para as entidades que são criadas de modo dinâmico. Esta zona de memória é chamada o *amontoado*³. Quando é criada uma nova variável

³Do inglês, “heap”.

associada a ponteiros, o computador vá buscar ao amontoado a quantidade de memória necessária para armazenar uma variável anónima. Por “ir buscar” entenda-se a associação de uma parte do amontoado com a variável dinâmica, a qual deixa de ser considerada como pertencente ao amontoado. No caso do tipo `no`, quando criamos uma nova instância (correspondente à invocação do método `__init__`), esta criação simula a operação que vai buscar uma porção de memória ao amontoado. Em muitas linguagens de programação, por exemplo, o C, esta operação é realizada através da invocação de uma função, por exemplo, a função `malloc` em C.

Quando uma variável dinâmica deixa de ser necessária (como se ilustra na Figura 15.10), a porção de memória com ela associada, deve ser devolvida ao amontoado, caso contrário, este pode esgotar-se. Em programação, existem dois métodos básicos para devolver memória ao amontoado, a gestão manual e a recolha de lixo.

15.3.1 A gestão manual do amontoado

A *gestão manual*⁴ do amontoado, que é utilizada em linguagens de programação como o C, transfere para o programador a responsabilidade de devolver ao amontoado a memória que já não é necessária. O C possui uma função, chamada `free`, que recebe como argumento uma variável do tipo ponteiro e cuja execução tem por efeito devolver ao amontoado a quantidade de memória ocupada pela variável anónima para onde a variável que é seu argumento aponta. Depois da execução da instrução `free(P)`, em que `P` é uma variável do tipo ponteiro, a variável `P` é indefinida e a memória ocupada pela variável anónima para onde `P` apontava foi devolvida ao amontoado.

Este método, que pode parecer ser o método mais racional para recuperar memória para o amontoado, pode dar origem a dois problemas, a geração de lixo e as referências soltas. Uma variável anónima para onde não aponta nenhuma variável do tipo ponteiro é chamada *lixo*, pois corresponde a uma porção de memória que não tem utilidade, pois não pode ser acedida, e não pode ser devolvida ao amontoado. Este aspecto já foi apresentado na página 417, em que mostrámos uma variável anónima que corresponde a lixo. A geração de lixo tem o problema de produzir áreas de memória que não podem ser devolvidas ao

⁴Do inglês, “manual updates”.

amontoado. As referências soltas representam um problema muito mais sério. Uma *referência solta*⁵ (ou um *ponteiro solto*) é gerada quando uma variável anónima é destruída (e, consequentemente, devolvida ao amontoado) antes de todos os ponteiros para essa variável serem destruídos. Neste caso, todos os ponteiros ainda existentes para esta variável tornam-se referências soltas: apontam para algo que já não existe. Na realidade, estes ponteiros apontam para uma zona de memória que se encontra no amontoado; se esta zona de memória for atribuída a uma outra variável anónima estes ponteiros continuam a apontar para esta variável, que pode agora ter um significado completamente diferente para o programa.

15.3.2 A recolha de lixo

A técnica de *recolha de lixo*⁶, permite a criação de lixo durante a execução de um programa. Quando o espaço disponível no amontoado desce abaixo de um certo limiar, a execução do programa é temporariamente suspensa e outro programa, o programa da recolha do lixo, é automaticamente executado. Este programa identifica o lixo existente e volta a colocá-lo no amontoado.

O princípio subjacente à recolha do lixo é o de que qualquer variável anónima que possa ser acedida a partir de um ponteiro existente no programa não é lixo, e todo o outro espaço associado a variáveis anónimas corresponde a lixo e deve ser devolvido ao amontoado. Uma vez que a recolha do lixo só raramente se faz, o seu algoritmo pode ser computacionalmente pesado. A recolha do lixo é efectuada em duas fases sequenciais:

1. Na fase de *marcação*⁷, toda a zona de memória a que é possível aceder a partir do nome das variáveis existentes nos ambientes é marcada como não sendo lixo.
2. Na fase de *varrimento*⁸, toda a memória que foi inicialmente atribuída ao amontoado é percorrida, e todas as suas zonas que não estão marcadas como não sendo lixo são devolvidas ao amontoado.

O Python utiliza a técnica da recolha de lixo.

⁵Do inglês, “dangling reference”.

⁶Do inglês, “garbage collection”.

⁷Do inglês, “mark phase”.

⁸Do inglês, “sweep phase”.

15.4 Notas finais

Neste capítulo, introduzimos a noção de ponteiro. Ao utilizar um ponteiro não estamos interessados no seu valor, mas sim no valor para que ele aponta. Os ponteiros podem ser criados e destruídos durante a execução de um programa. Discutimos também o conceito de estruturas dinâmicas, estruturas cujo tamanho varia durante a execução de um programa. A variação do tamanho das estruturas dinâmicas é obtida através da manipulação de ponteiros que fazem a ligação entre os seus componentes.

Discutimos superficialmente duas técnicas para a gestão de memória, a gestão manual e a recolha do lixo. As técnicas de gestão de memória são muito importantes em programação. Os conceitos introduzidos neste capítulo podem ser consultados com maior profundidade em [Cohen, 1981], um artigo de perspectiva sobre o processo da recolha do lixo, que apresenta não só as técnicas clássicas utilizadas na recolha do lixo, mas também abordagens mais recentes. Apresenta também uma bibliografia exaustiva sobre a recolha do lixo. O livro [Knuth, 1973a] discute, em grande pormenor, os ponteiros, as estruturas obtidas com recurso a ponteiros e as técnicas de gestão de memória.

15.5 Exercícios

1. Considere a classe no apresentada na página 409:

- (a) Escreva instruções em Python para gerar a estrutura apresentada na Figura 15.11

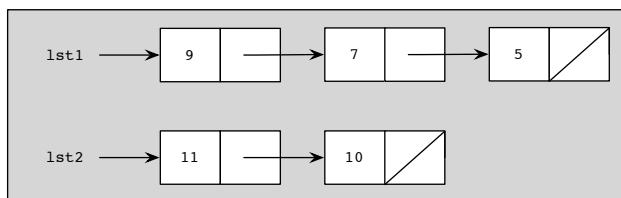


Figura 15.11: Estrutura de nós.

- (b) Partindo da estrutura apresentada na Figura 15.11, escreva instruções em Python para gerar a estrutura apresentada na Figura 15.12. As

susas instruções deram origem à criação de lixo? Justifique a sua resposta.

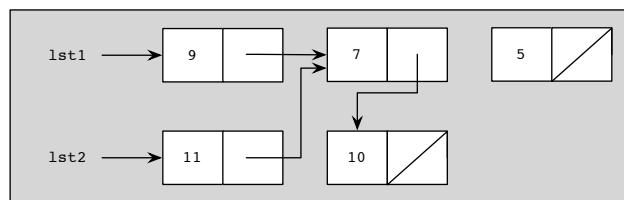


Figura 15.12: Alteração à estrutura da Figura 15.11.

2. Considerando a classe `no` apresentada na página 409, escreva em Python as instruções necessárias para obter as estruturas representadas nas figuras 15.13 e 15.14. Note que estas estruturas contêm um ponteiro que aponta para a própria estrutura. Por esta razão, são chamadas *estruturas circulares*.

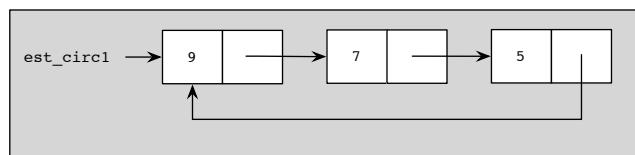


Figura 15.13: Estrutura circular (caso 1).

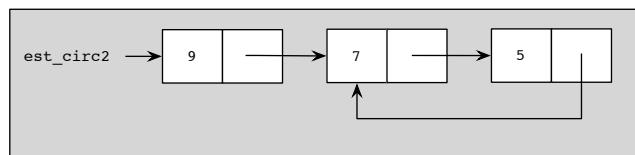


Figura 15.14: Estrutura circular (caso 2).

3. Escreva em Python uma função chamada `estrutura_circular`, que recebe uma estrutura de nós e devolve *verdadeiro*, se se tratar de uma estrutura circular, e *falso*, em caso contrário.
4. Escreva em Python uma função chamada `desfaz_estrutura_circular`, que recebe uma estrutura circular e a transforma na correspondente es-

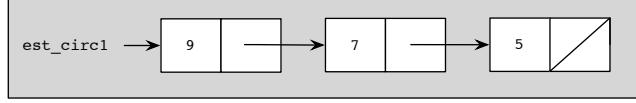


Figura 15.15: Estrutura circular modificada (caso 1).

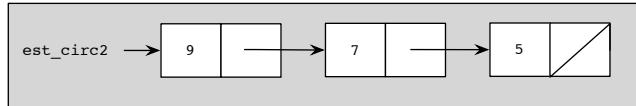


Figura 15.16: Estrutura circular modificada (caso 2).

trutura não circular, transformando o ponteiro para a própria estrutura numa entidade que não aponta para nada. Por exemplo, aplicando esta função às estruturas `est_circ1` e `est_circ2` das figuras 15.13 e 15.14, obtinhamos, respectivamente, as estruturas apresentadas nas figuras 15.15 e 15.16.

5. O tipo de informação *lista circular* corresponde a uma lista na qual, excepto no caso de ser uma lista vazia, ao último elemento se segue o primeiro. A Figura 15.17 (a) mostra esquematicamente uma lista circular em que o primeiro elemento é 4, o segundo, 3, o terceiro, 5, o quarto, 2 e o quinto (e último) é 1. Uma lista circular tem um elemento que se designa por primeiro elemento ou elemento do início da lista. Na lista anterior, esse elemento é 4.

Com listas circulares, podemos:

- Inserir um elemento na lista, realizado através da operação `insere_circ`. Com esta operação, o novo elemento passa a ser o primeiro da nova lista, o primeiro elemento da lista original passa a ser o segundo da nova lista, e assim sucessivamente.
- Inspeccionar o primeiro elemento da lista, realizado através da operação `primeiro_circ`, sem alterar a lista.
- Retirar um elemento da lista, realizado através da operação `retira_circ`. Com esta operação, o elemento retirado é sempre o do início da lista, passando o início da nova lista a ser o segundo elemento (se este existir) da lista original.

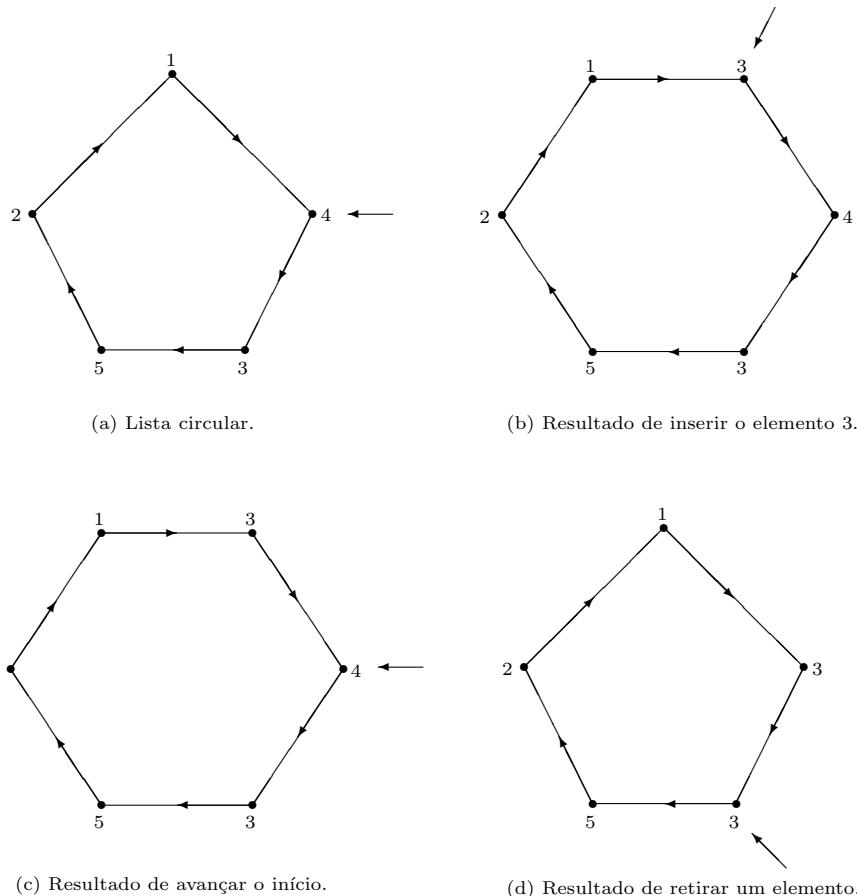


Figura 15.17: Operações sobre listas circulares.

— Avançar o início da lista para o elemento seguinte, realizado através da operação *avanca_circ*. Esta operação não altera os elementos da lista, apenas altera o início da lista, que passa a ser o segundo elemento da lista original, se esta tiver pelo menos dois elementos; se apenas tiver um elemento, nada se altera; se a lista circular for vazia, esta operação tem um valor indefinido.

Na Figura 15.17 estão exemplificadas estas operações.

- Especifique estas operações, e outras que considerar necessárias, e classifique-as. Defina também uma representação externa para listas circulares.

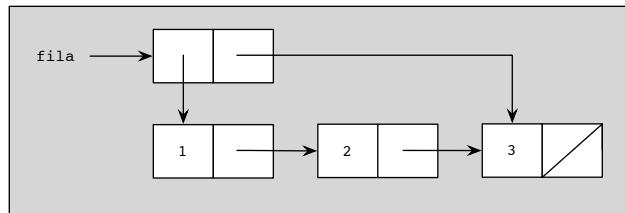


Figura 15.18: Representação da fila < 1 2 3 <.

- (b) Escolha uma representação interna para o tipo lista circular, e escreva em Python as operações básicas e o transformador de saída.
- 6. Considere a classe `lista` apresentada na página 418. Defina um método chamado `junta` que junta uma lista arbitrária no final de uma dada lista.
- 7. Escreva a classe `fila` (baseada nas operações apresentadas na página 355), usando uma representação baseada em ponteiros. Por uma questão de eficiência, deve utilizar uma representação semelhante à usada na Figura 15.18. Note que precisa de definir uma estrutura que agrupa dois ponteiros.
- 8. Escreva a classe `arvore` (baseada nas operações apresentadas na página 399), usando uma representação baseada em ponteiros. Note que vai precisar de alterar o tipo `no` de modo a que este contenha, para além de um valor, dois ponteiros.

Capítulo 16

Epílogo

'Would you tell me, please, which way I ought to go from here?'
'That depends a good deal on where you want to get to,' said the Cat.

Lewis Carroll, *Alice's Adventures in Wonderland*

No início dos anos 80, com o advento do computador pessoal, a utilização dos computadores banalizou-se. Num artigo escrito para o grande público, a revista TIME, em 1984, qualificou os programas (o chamado “software”) como *o feitiçario dentro da máquina*. O artigo era iniciado do seguinte modo: “Hardware ou software?, pode questionar uma criança da era dos computadores. O seu mundo é dividido em duas partes: hardware, os componentes físicos que constituem um computador, e software, os programas que indicam ao computador o que fazer. Embora o hardware seja visível e tangível, a criança sabe que o software é a alma da máquina. Sem software, um computador é pouco mais do que um aglomerado de plástico e silicone sem qualquer utilidade. Um computador sem software é como um automóvel sem gasolina, uma máquina fotográfica sem filme, um aparelho estereofónico sem discos.” [Taylor, 1984], página 42.

Nas quase três décadas que decorreram desde a publicação deste artigo, verificaram-se avanços profundos tanto no “hardware” como no “software”. O “hardware” tornou-se mais rápido, mais barato, mais pequeno, a sua capacidade de armazenamento de informação aumentou ordens de magnitude, implantou-se em quase todos os aspectos relacionados com a vida moderna, telemóveis, agendas electrónicas, máquinas fotográficas, automóveis, aparelhos electrodomésticos,

entre milhares de outros. O “*software*” foi sujeito a novos desafios, abriram-se novas áreas de aplicação, aumentou-se a complexidade das aplicações, surgiu a Internet. No entanto, a ideia transmitida no artigo da TIME mantém-se, sem o “*software*”, os computadores (“*hardware*”) não têm utilidade.

Como foi bem caracterizado pela revista TIME, o “*software*” é o *feiticeiro dentro da máquina*. Um dos objectivos da Informática corresponde ao estudo e desenvolvimento das entidades abstractas que residem nos computadores, os processos computacionais. Um *processo computacional* é um ente imaterial que evolui ao longo do tempo, executando accções que levam à solução de um problema. A evolução de um processo computacional é ditada por uma sequência de instruções a que se chama *programa*, e a actividade de desenvolvimento de programas é chamada *programação*.

Neste livro apresentámos conceitos básicos de programação. Neste último capítulo fornecemos uma perspectiva dos principais conceitos introduzidos e apontamos direcções para o seu estudo mais aprofundado. Juntamente com cada tópico, fornecemos a indicação de disciplinas da Licenciatura em Engenharia Informática e de Computadores do Instituto Superior Técnico que abordam o tópico e referências bibliográficas nas quais o tópico é discutido com profundidade.

16.1 Programas

Um dos conceitos basilares da programação corresponde à noção de *programa*. Um programa é uma entidade estática que define a evolução de um processo computacional.

Um programa relaciona-se com uma multiplicidade de outros conceitos: um programa corresponde a um *algoritmo*, um programa é expresso numa dada *linguagem*, um programa engloba um conjunto de *abstracções*, e a execução de um programa origina um *processo computacional*.

Na Figura 16.1 apresentamos as relações entre os conceitos associados com programa, indicando, dentro de um rectângulo, os conceitos abordados neste livro.

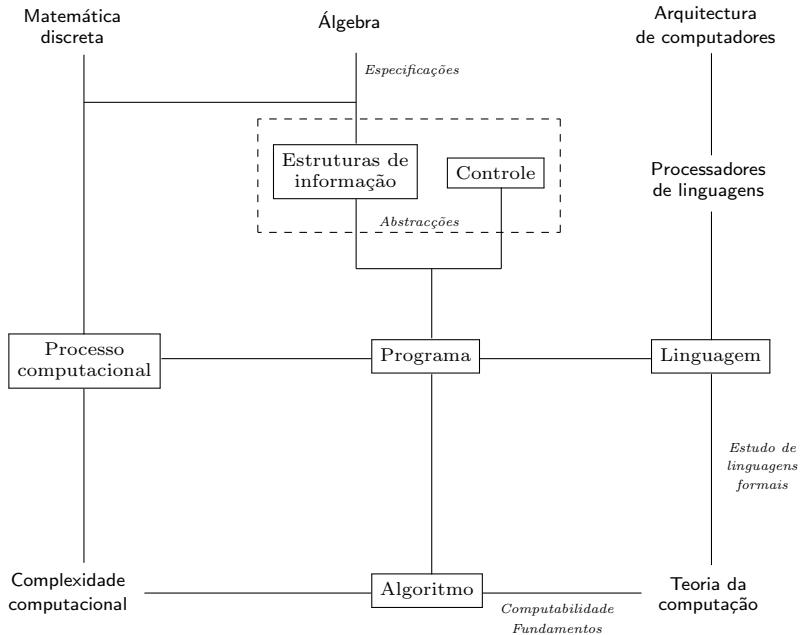


Figura 16.1: “Mapa” dos principais conceitos associados a programas.

16.1.1 Algoritmos

Um programa é caracterizado matematicamente como sendo um *algoritmo* – uma sequência de passos que podem ser executados mecanicamente (isto é, sem exigir recurso à criatividade ou à imaginação), de modo a atingir um determinado objectivo.

Um algoritmo está sempre associado a um dado *objectivo*, ou seja, à solução de um dado problema. A execução das instruções do algoritmo garante que o seu objectivo é atingido. A sequência de passos de um algoritmo deve ser executada por um agente, o qual pode ser humano, mecânico, electrónico, ou qualquer outra coisa. Cada algoritmo está associado a um *agente* (ou, mais correctamente, a uma classe de agentes) que deve executar as suas instruções. Aquilo que representa um algoritmo para um agente pode não o representar para outro agente.

Embora um algoritmo não seja mais do que uma descrição da sequência de passos a seguir para atingir um objectivo, nem todas as sequências de passos para atingir um objectivo podem ser consideradas um algoritmo, pois um algoritmo

deve possuir três características: (1) um algoritmo é *rigoroso*; (2) um algoritmo é *eficaz*; (3) um algoritmo *deve terminar*.

O estudo de algoritmos é uma das principais vertentes ligadas à programação. Este estudo pode ser encarado sob várias perspectivas:

— *O estudo dos fundamentos dos algoritmos*

Esta perspectiva aborda o conceito de *computabilidade* (ou seja, qual o significado da execução mecânica de instruções) e as *limitações teóricas* a que estão sujeitos os algoritmos.

Este estudo, fundamentalmente de natureza matemática, começou a ser abordado nos anos 30, antes do advento dos computadores digitais, com os trabalhos de Church, Kleene, Post e Turing, e continua a ser de grande relevância para a programação, pois permite-nos determinar até que ponto podemos garantir que existe um algoritmo para resolver um dado problema. O domínio científico que se debruça sobre este tema é conhecido por *Teoria da Computação* [Sernadas, 1993], [Sipser, 2012].

— *O estudo da complexidade de algoritmos*

No Capítulo 7 apresentámos alguns padrões típicos da evolução de processos computacionais, estudando a ordem de grandeza do número de operações associadas ao processo e o “espaço” exigido pela evolução global do processo. Vimos que, para resolver um mesmo problema, é possível desenvolver algoritmos diferentes, originando processos computacionais que apresentam grandes diferenças na taxa a que consomem recursos.

A disciplina de *Análise e Síntese de Algoritmos* aborda o estudo da *complexidade computacional* associada a diferentes algoritmos, a definição de várias medidas para avaliar a ordem de crescimento de processos (para além da notação O apresentada na Secção 5.7 existem muitas outras, por exemplo, a notação Θ , a notação o e a notação Ω) e a classificação de classes de “dificuldade” de problemas (por exemplo, problemas de dificuldade P e problemas de dificuldade NP) [Cormen et al., 2009], [Sipser, 2012].

A análise de algoritmos necessita de um conjunto de conceitos matemáticos tais como análise combinatória, teoria das probabilidades, manipulação algébrica, as quais são tratadas num ramo da matemática conhecido por *Matemática Discreta* [Graham et al., 1989], [Johnsonbaugh, 2009].

16.1.2 Linguagens

De modo a expressar algoritmos é necessário utilizar linguagens formais (linguagens que não apresentam ambiguidade).

Qualquer linguagem apresenta dois aspectos distintos, a forma da linguagem e o significado associado à forma. Estes aspectos são denominados, respectivamente, sintaxe e semântica da linguagem. A *sintaxe* de uma linguagem é o conjunto de regras que definem quais as relações válidas entre os componentes da linguagem, tais como nomes, expressões, formas. A sintaxe nada diz em relação ao significado das frases da linguagem. A *semântica* de uma linguagem define qual o significado de cada frase da linguagem. A semântica nada diz quanto ao processo de geração das frases da linguagem.

Uma linguagem que é compreendida por um computador é chamada uma *linguagem de programação*. Ao estudar uma linguagem de programação, é fundamental uma perfeita compreensão da sua sintaxe e da sua semântica: a sintaxe vai determinar a forma das instruções a fornecer ao computador, e a semântica vai determinar o que o computador faz ao executar cada uma das instruções.

É pois natural que o estudo de linguagens seja um aspecto importante associado à Programação. Este estudo pode ser abordado através de várias perspectivas:

— *O estudo de linguagens formais*

Aborda o estudo dos diferentes tipos de gramáticas, das linguagens que estas originam, e da relação de uma linguagem com o conceito de *autómato*, um modelo abstracto de uma “máquina” que é comandada por uma linguagem. Esta área que surge em meados dos anos 50 com o trabalho de Noam Chomsky [Chomsky, 1956] tem tido uma influência profunda no desenvolvimento das linguagens de programação modernas. O estudo de linguagens formais é abordado no domínio científico da *Teoria da Computação* [Ginsburg, 1966], [Sernadas, 1993], [Moret, 1997].

— *O estudo de processadores de linguagens*

Aborda os métodos para o desenvolvimento de camadas de abstracção que “entendam” determinadas linguagens de programação.

Antes de iniciar a discussão deste tópico, convém mencionar alguns dos tipos de linguagens de programação. As linguagens de programação utilizadas no

desenvolvimento da esmagadora maioria das aplicações são conhecidas por linguagens de alto nível.

Uma *linguagem de alto nível* apresenta semelhanças com as linguagens que os humanos usam para resolver problemas, e é independente do computador onde é executada. O Python é um exemplo de uma linguagem de alto nível. As linguagens de alto nível diferem drasticamente das linguagens que na realidade comandam directamente o funcionamento dos computadores.

Sendo os computadores digitais, toda a informação neles representada recorre apenas a dois estados discretos (e daí a designação de digital). A *linguagem máquina* é a linguagem utilizada para comandar directamente as acções do computador. As instruções em linguagem máquina são constituídas por uma sequência de símbolos correspondendo às entidades discretas manipuladas pelo computador (normalmente representados por 0 e por 1) e actuam directamente sobre os componentes do computador. A linguagem máquina é difícil de usar e de compreender por humanos e varia de computador para computador (é a linguagem nativa de cada computador).

Para que os computadores possam “entender” os programas escritos numa linguagem de alto nível, existem programas que “traduzem” as linguagens de alto nível em linguagem máquina (recordese, que no início do Capítulo 1, dissemos que um computador é uma “caixa electrónica” que tem a capacidade de compreender e de executar as instruções que correspondem a programas). Esta “caixa electrónica” corresponde a uma camada de abstracção que se chama *máquina virtual* – corresponde à realização de uma máquina conceptual através de um programa. Existem duas abordagens para fazer esta tradução, os interpretadores e os compiladores:

— Interpretadores

No caso do Python, a tradução é feita através de um programa, chamado *interpretador* do Python, que lê formas, avalia-as e fornece o seu resultado.

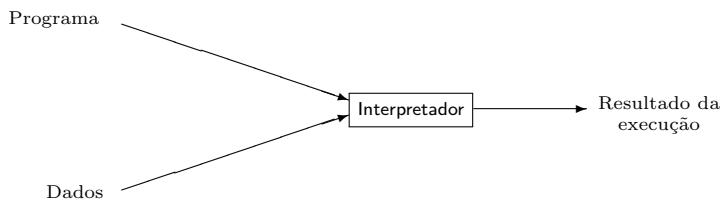


Figura 16.2: Conceito de interpretação.

Com a utilização de um interpretador, a máquina virtual recebe uma forma (instrução) de cada vez, analisa-a e executa a sequência de instruções que, em linguagem máquina, corresponde a essa forma. Só depois da forma avaliada, é que a máquina virtual vai receber a próxima forma (Figura 16.2). Esta abordagem dá origem ao *ciclo lê-avalia-escreve* discutido no início do Capítulo 2.

— *Compiladores*

No caso de outras linguagens de alto nível, por exemplo o C, a tradução é feita através de um programa, chamado *compilador*, que recebe um programa escrito na linguagem (a que se dá o nome de *programa fonte*) e que produz um programa equivalente (equivalente no sentido em que com os mesmos dados produz os mesmos resultados) escrito em linguagem máquina (a que se dá o nome de *programa objecto*).

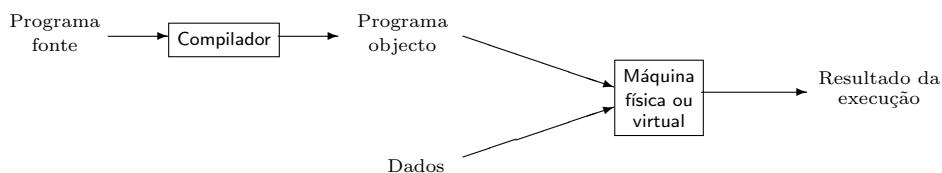


Figura 16.3: Conceito de compilação.

Com a utilização de um compilador, fornecem-se todas as instruções do programa fonte e, antes de o começar a executar, o compilador analisa cada uma das suas instruções e substitui-a pelo conjunto de instruções que a realiza, criando-se assim um *novo programa*, o programa objecto, equivalente ao primeiro, mas constituído por instruções para outra “máquina”. Em compilação, a execução de um programa passa por duas fases distintas: em primeiro lugar cria-se o programa objecto, depois executa-se o programa objecto (Figura 16.3).

Uma das vertentes associadas à programação corresponde ao desenvolvimento de *interpretadores* e de *compiladores* [Aho et al., 1986], ou, de um modo mais genérico, de *processadores de linguagens*, os quais correspondem à operacionalização de linguagens de programação. No desenvolvimento de um processador para uma linguagem, vários aspectos têm de ser considerados, tais como a análise léxica das frases da linguagem (o reconhecimento individual dos componentes), a análise sintáctica das frases, a análise semântica,

a criação de estruturas resultantes da análise, a sua tradução, a optimização do código gerado, os modelos de avaliação, a gestão de memória, etc.

O estudo de processadores de linguagens, para além de necessitar de conceitos associados ao estudo de linguagens formais, está intimamente ligado com o estudo da *Arquitectura de Computadores* [Mano e Kime, 1997], [Arroz et al., 2007], ou seja, o estudo dos vários componentes que constituem um computador, as suas interligações físicas e o modo como estes componentes comunicam entre si.

— *O estudo comparativo de linguagens de programação*

Para dominar a programação, é essencial um conhecimento pormenorizado de várias linguagens de programação. O estudo comparativo de linguagens de programação [Fischer e Grodzinsky, 1993], [Wilson e Clark, 1988], [Scott, 2000], [Turbak et al., 2008] visa a análise do modo como diferentes linguagens abordam os conceitos subjacentes à programação.

Existem conceitos que estão presentes em todas as linguagens de programação, tais como tipos de informação, estruturas de controle, abstracção, nomeação, entre muitos outros. A análise da utilização destes conceitos em diferentes linguagens permite a sua compreensão a um nível de abstracção mais elevado, favorecendo a sua utilização em linguagens específicas. Este estudo não só aumenta a compreensão sobre linguagens de programação, como também facilita a aprendizagem de novas linguagens.

16.1.3 Construção de abstracções

Uma técnica fundamental em programação, a *abordagem do topo para a base*, consiste em identificar os principais subproblemas que constituem o problema a resolver e em determinar qual a relação entre eles. Este processo é repetido para cada um dos subproblemas, até se atingirem problemas cuja solução é simples.

A abordagem do topo para a base permite estruturar um problema em subproblemas, de modo a perceber os passos que têm de ser seguidos para atingir a sua solução. Cada um dos subproblemas em que um problema é dividido corresponde a uma *abstracção* – existem pormenores que são ignorados, nomeadamente os passos para os resolver.

Qualquer programa corresponde a um algoritmo para resolver um modelo, ou

uma abstracção, de um dado problema. Sob este ponto de vista, a actividade de programação corresponde a uma *construção de abstracções*. Estas abstracções estão estruturadas em camadas. Em cada camada de abstracção interessa separar o que é essencial ao problema dos conceitos ou pormenores acessórios. A abstracção está omnipresente em programação.

As primeiras abstracções utilizadas em programação corresponderam à abstracção de máquinas e à abstracção procedural.

— *Abstracção de máquinas*

A abstracção de máquinas consiste na criação de máquinas virtuais (como descrito na Secção 16.1.2) que aceitam uma dada linguagem de programação, permitindo abstrair a linguagem subjacente ao computador utilizado e os pormenores exigidos por esta linguagem.

Assim, uma linguagem de programação corresponde à abstracção de uma máquina virtual cuja “linguagem máquina” é a linguagem de programação – *o computador é assim a “caixa electrónica” que tem a capacidade de compreender e de executar as instruções que correspondem a programas*.

— *Abstracção procedural*

A abstracção procedural consiste em abstrair do modo como as funções realizam as suas tarefas, concentrando-se apenas na tarefa que as funções realizam. Ou seja, corresponde à separação do “como” de “o que”. A abstracção procedural é realizada através da nomeação de funções, abstraindo-se do modo como estas realizam a sua tarefa.

Posteriormente, reconheceu-se que num programa são necessárias outras duas facetas da abstracção, a abstracção de controle e a abstracção de dados. Esta verificação levou à famosa expressão introduzida por [Wirth, 1976]: “*Algoritmos + Estruturas de Informação = Programas*”.

— *Abstracção de controle*

A abstracção de controle corresponde à conceptualização de mecanismos que especificam o modo como as instruções de um programa são executadas. Em linguagem máquina, o controle é realizado fundamentalmente através da sequenciação implícita, seguindo a sequência física das instruções de um

programa, complementada por dois mecanismos de excepção, o salto incondicional e o salto condicional. Estas estruturas de controle aparecem enraizadas nas primeiras linguagens de programação.

O movimento em favor da *programação estruturada* (liderado nos anos 60 por Edsger W. Dijkstra [Dijkstra, 1976] e [Dahl et al., 1972]) levou à introdução de mecanismos abstractos de controle, tais como o *if-then-else*, o *case*, o *repeat-until*, o *while-do* e o *for*, que permitem que o programador se concentre em especificar o controle da execução em termos de actividades simples, modulares e bem compreendidas.

Mais recentemente, o *paralelismo* permite conceptualizar a utilização de várias máquinas em simultâneo, abstraindo do facto que que, eventualmente, a máquina subjacente apenas tem um processador.

— Abstracção de dados

A abstracção de dados surge nos anos 70 com o trabalho de [Liskof e Zilles, 1974], [Liskof e Guttag, 1986] e corresponde à separação entre o estudo das propriedades dos dados e os pormenores da realização dos dados numa linguagem de programação. A abstracção de dados é traduzida pela separação das partes do programa que lidam com o modo como os dados são *utilizados* das partes que lidam com o modo como os dados são *representados*.

A abstracção de dados, corporizada na teoria dos tipos abstractos de informação, tem grandes ligações com a *matemática discreta* e as *álgebras*. Juntamente com o desenvolvimento da abstracção de dados, têm surgido novas abstracções de controle, associadas a tipos específicos de dados, por exemplo, os iteradores e os geradores.

Antes de concluir esta secção, convém referir que a distinção entre programas e dados corresponde também a uma abstracção. De facto, considerando a Figura 16.3, ao utilizarmos um compilador, o *programa fonte* para o programador é *um programa*; no entanto, o mesmo programa, para o compilador, corresponde a *dados*.

16.2 Programação

De um modo abstracto, o desenvolvimento de um programa pode ser visto como uma sequência de fases através das quais as descrições de um sistema se tornam

progressivamente mais pormenorizadas. Começando com a *análise do problema*, que dá ênfase ao que tem de ser feito, a descrição é refinada progressivamente para a descrição de como o problema é resolvido de um modo mecânico. Para isso, na fase do *desenvolvimento da solução* descreve-se rigorosamente como o problema vai ser resolvido, sem se entrar, no entanto, nos pormenores inerentes a uma linguagem de programação. Na fase da *programação da solução* o algoritmo desenvolvido é escrito recorrendo a uma linguagem de programação. Em resumo, durante a sequência de fases seguida no desenvolvimento de um programa, a caracterização de o que tem de ser feito transforma-se progressivamente numa especificação de como vai ser feito.

As fases do desenvolvimento de programas que levam dos requisitos iniciais ao desenvolvimento de código executável são guiadas por metodologias adequadas e sistemáticas, estudadas na disciplina de *Engenharia de Software*, e descritas, por exemplo, em [Alagar e Periyasamy, 1998], [Jalote, 1997], [Sommerville, 1996], [Pfleeger e Atlee, 2010].

Para além das metodologias de desenvolvimento de programas, a programação estuda a arquitectura dos programas, utiliza técnicas desenvolvidas em vários domínios da Informática e pode ser abordada seguindo diversos paradigmas. A programação pode abordar o desenvolvimento de aplicações concretas ou o desenvolvimento de programas para gerir outros programas, como é o caso dos sistemas operativos.

Na Figura 16.4 apresentamos as relações entre os conceitos associados com programação, indicando, dentro de um rectângulo, os conceitos tratados neste livro.

16.2.1 Arquitectura de programas

Um programa complexo é constituído por vários componentes que comunicam entre si. Estes componentes podem ser módulos do próprio programa ou podem ser outros programas existentes na organização para a qual o programa é desenvolvido.

A *Arquitectura de Programas* (do inglês “Software Architecture”) aborda os vários métodos para estruturar componentes e os mecanismos possíveis para a sua comunicação [Bass et al., 1998], [Hofmeister et al., 2000].

A arquitectura de programas é uma abstracção que ignora os algoritmos e a

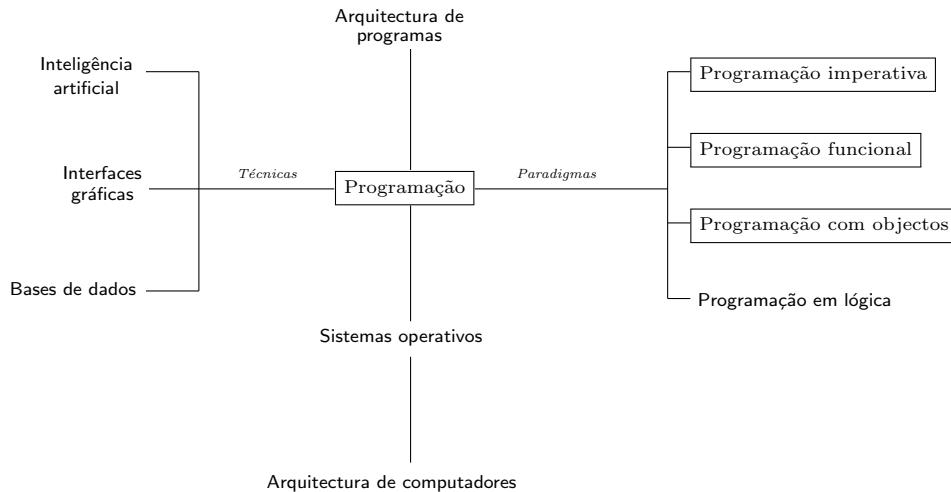


Figura 16.4: “Mapa” dos principais conceitos associados a programação.

representação dos dados e que se concentra no tipo de tarefas que vão ser realizadas pelos programas, na partilha e na reutilização de componentes (componentes esses que podem ser escritos em diferentes linguagens de programação), na interligação entre os componentes, nos modos de comunicação e na localização “geográfica” dos componentes (estes podem ser sistemas centralizados ou sistemas distribuídos [Marques e Guedes, 1998]).

16.2.2 Paradigmas de programação

Um *paradigma de programação* é um modelo para abordar o modo de raciocinar durante a fase de programação e, consequentemente, o modo como os programas são escritos. Existem vários paradigmas de programação, sendo os mais importantes a programação imperativa, a programação funcional, a programação com objectos e a programação em lógica.

— Programação imperativa

O paradigma da programação imperativa corresponde à abordagem mais comum à programação. A programação imperativa segue o modelo de computação designado por *modelo de von Neumann*¹ ou *modelo do programa*

¹Em honra do matemático John von Neumann (1903–1957) que teve grande influência no desenvolvimento do conceito de computação baseada num programa armazenado.

armazenado – o qual está subjacente à esmagadora maioria dos computadores –, segundo o qual, tanto o programa como os dados estão armazenados na memória do computador e, durante a execução do programa, as células (ou posições) da memória são repetidamente acedidas, interpretadas e actualizadas.

A programação imperativa reflecte as acções executadas por um computador ao nível da linguagem máquina. A este nível, as principais acções executadas por um computador correspondem a ir buscar os conteúdos de certas posições de memória (que contêm as instruções – indicação de operadores e eventualmente de operandos), à aplicação da operação explicitada na instrução a valores obtidos da memória e em armazenar o resultado numa determinada posição de memória. As linguagens imperativas abstraem a noção de conteúdo de uma posição de memória, substituindo-a pelo valor de uma variável, mas os passos seguidos são basicamente os mesmos.

A base da programação imperativa reside na operação de atribuição através da qual uma variável recebe o resultado de uma operação – como vimos no Capítulo 5, isto corresponde à utilização de efeitos. Paralelamente, a programação imperativa recorre exaustivamente a métodos para controlar e especificar a execução de sequências de instruções. A tarefa de programação é considerada como a escrita de uma sequência de ordens dadas ao computador (e daí a designação de imperativa).

São exemplos de linguagens baseadas na programação imperativa o C, o Pascal, o FORTRAN e o BASIC.

— *Programação funcional*

A programação funcional [Cousineau e Mauny, 1998], [Hudac, 1989], [MacLennan, 1990] tem as suas raízes no modelo desenvolvido pelo matemático Alonzo Church (1903–1995) sobre a teoria das funções recursivas e sobre o cálculo lambda. A programação funcional é baseada no conceito de função matemática, uma transformação de elementos do domínio em elementos do contradomínio. O paradigma funcional considera os programas como funções matemáticas – caixas pretas que recebem um certo número de valores de entrada, os parâmetros concretos, e produzem um único valor, o resultado. No paradigma funcional, o único modo de especificar as acções de uma função é através da invocação de outras funções.

Uma das propriedades das linguagens funcionais corresponde à não existência

de variáveis e à não existência da operação de atribuição. Os parâmetros de funções, embora sejam nomeados e utilizados de modo semelhante a variáveis, nunca são alterados. Sob esta perspectiva, os parâmetros de funções podem ser considerados como designações de constantes.

Um outro aspecto associado à programação funcional é o facto de os parâmetros de funções e o valor de funções poderem ser outras funções. Este aspecto faz com que frequentemente se diga que a programação funcional trata as funções como cidadãos de primeira classe.

O principal mecanismo de controle da execução em linguagens funcionais corresponde à recursão combinada com a utilização de expressões condicionais (de que é exemplo a expressão `cond` em Scheme), as quais avaliam condições (expressões cujo valor é *verdadeiro* ou *falso*) e, mediante o resultado da avaliação, avaliam selectivamente outras funções.

São exemplos de linguagens exclusivamente baseadas no paradigma da programação funcional o Miranda, o Haskell e o FP. O Scheme, o LISP e o ML são linguagens que, embora baseadas na programação funcional, apresentam algumas características imperativas.

— *Programação com objectos*

O paradigma de programação com objectos [Meyer, 1997], [Schach, 2005] surge com o desenvolvimento das ideias introduzidas, em meados dos anos 60, pela linguagem Simula, juntamente com o reforço dos conceitos ligados à abstracção de dados.

A programação com objectos, embora baseada no modelo de von Newmann, difere da programação imperativa, pois em vez de considerar um processador e uma memória monolíticos, considera um modelo de computação mais estruturado e distribuído, organizando os processos computacionais em torno de objectos, cada um dos quais pode ser considerado um “computador” independente com a sua própria memória (o seu estado) e as suas funções executáveis (os métodos) que manipulam a sua memória.

Em programação com objectos, a programação é centrada na construção de abstracções chamadas objectos, os quais comunicam entre si através de mensagens e estão organizados em estruturas hierárquicas de classes. Associado à organização de classes, existe um mecanismo de herança, o qual permite a criação de novas abstracções através da especialização e da combinação das classes existentes.

São exemplos de linguagens baseadas no paradigma da programação com objectos, o Smalltalk, o Eiffel, o C++, o CLOS (Common LISP Object System), o Java e o Python.

— *Programação em lógica*

O paradigma da programação em lógica [Hogger, 1990] tem as suas raízes no desenvolvimento da lógica matemática e nos trabalhos de Herbrand [Herbrand, 1930] e de Robinson [Robinson, 1965] sobre a demonstração automática de teoremas. De um modo simples (e ideal) podemos definir a tarefa da programação em lógica do seguinte modo: O programador descreve as propriedades lógicas que caracterizam o problema a resolver. Esta descrição é utilizada pelo sistema para encontrar uma solução do problema a resolver (ou para *inferir* a solução para o problema).

A abordagem da programação em lógica é obtida representando as especificações que caracterizam o problema num componente individualizado, que se chama *base de conhecimento*. A base de conhecimento é independente do programa que a manipula, programa esse que é chamado a *máquina de inferência*. Em programação em lógica, as descrições de um problema são fornecidas num formalismo baseado em lógica de primeira ordem.

Suponhamos, a título de exemplo, que desejávamos especificar o efeito de procurar um elemento, x , numa lista de elementos, L . Podemos definir o predicado *em?* como verdadeiro, sempre que o elemento x pertence à lista L . Este predicado pode ser especificado do seguinte modo:

Para qualquer elemento x e qualquer lista L :

$em?(x, L)$ se e só se
 $L =$ lista que apenas contém x
ou
 $L =$ junção de L_1 e L_2 e ($em?(x, L_1)$ ou $em?(x, L_2)$)

esta especificação descreve uma função de procura de um modo declarativo: o elemento pertence à lista se a lista é constituída exactamente por esse elemento; em caso contrário consideramos a lista como sendo constituída por duas sublistas cuja junção é a lista original, estando o elemento na lista se este estiver numa das sublistas.

A programação em lógica baseia-se no facto de que existem duas abordagens distintas para representar o conhecimento: através de declarações (a repre-

sentação declarativa) e através de funções (a representação procedural). A *representação declarativa* parte do princípio de que saber é “saber o quê” e, consequentemente, o conhecimento é representado através de um conjunto de declarações ou afirmações descrevendo um domínio particular. Sobre estas declarações actua uma função de ordem geral que permite inferir nova informação. A *representação procedural* parte do princípio de que saber é “saber como” e, consequentemente, o conhecimento é representado como um conjunto de funções, cada uma das quais representa um fragmento de conhecimento.

Do ponto vista formal não existe distinção entre estas duas abordagens. Sabemos que os programas podem ser olhados como dados, e sob este ponto de vista tudo é declarativo. Por outro lado, podemos olhar para as declarações como instruções de um programa, e sob este ponto de vista tudo são funções.

O paradigma da programação em lógica é corporizado na linguagem PROLOG. Esta linguagem foi desenvolvida em 1972 associada à resolução de problemas em Inteligência Artificial. Actualmente, o PROLOG apresenta um grande leque de aplicações que transcendem a Inteligência Artificial. O PROLOG tem sido utilizado para a definição de tradutores de linguagens, interpretadores e compiladores, bases de dados dedutivas, interfaces em língua natural, etc.

A programação em lógica apenas satisfaz parcialmente a abordagem declarativa. Para tornar a execução do programa eficiente, é introduzido um certo número de compromissos que diluem o aspecto da representação declarativa pura. Isto significa que o programador tem de se preocupar com mais aspectos do que apenas a especificação daquilo que o programa é suposto fazer. Neste sentido, existem aspectos não declarativos que podem ser vistos como indicações dadas pelo programador à máquina de inferência. De um modo geral, estes aspectos reduzem a clareza das descrições do programa, pois misturam a descrição do problema com preocupações de implementação.

Usando a programação em lógica, um programa é constituído por uma série de declarações em lógica, juntamente com um conjunto de indicações procedimentais que controlam a utilização das declarações. Por esta razão um programa é considerado como uma combinação de lógica e de controlo, traduzidas na conhecida expressão: “*Programa = Lógica + Controlo*” [Kowalski, 1979].

16.2.3 Técnicas usadas em programação

Existem vários domínios científicos em Informática que contribuem para a programação com um conjunto de técnicas que estendem e que facilitam a utilização dos computadores. Nesta secção apresentamos resumidamente três desses domínios científicos, a Inteligência Artificial, as interfaces gráficas e o domínio associado ao desenvolvimento de bases de dados.

— *Inteligência Artificial*

A Inteligência Artificial [Russell e Norvig, 2010] é o ramo da Informática que estuda os métodos para os computadores executarem tarefas que, quando executadas por humanos, são ditas requerer inteligência.

Embora a Informática tenha contribuído para o desenvolvimento de programas que resolvem problemas muito variados, estes problemas têm uma característica em comum, a de a sua solução poder ser descrita por modelos matemáticos e, portanto, poder ser expressa por um algoritmo. A solução da maioria dos problemas que encaramos na nossa vida quotidiana não pode ser descrita por um algoritmo – existe falta de especificação completa do problema, existem ambiguidades e, por vezes, mesmo contradições. A nossa capacidade para encarar e resolver com sucesso problemas deste tipo dá-nos o direito à classificação de inteligentes. Para resolver estes problemas, nem sempre seguimos regras exactas, frequentemente utilizamos regras empíricas, acumuladas ao longo de anos de experiência que, embora não nos garantam a solução dos problemas, a maior parte das vezes resultam. Estas regras empíricas são chamadas *heurísticas*².

Existe ainda outro tipo de problemas para os quais, embora a solução possa ser descrita por algoritmos, estes não são úteis devido à sua complexidade computacional. Por mais potentes que sejam os computadores, os problemas deste tipo não podem ser resolvidos algoritmicamente, devido a limitações de espaço e de tempo. Uma vez mais, são as heurísticas que permitem a resolução destes problemas. Embora não garantam a melhor solução, e por vezes nem sequer solução alguma, as heurísticas permitem, a maior parte das vezes, resolvê-los num período de tempo razoável.

Uma das características das técnicas usadas em Inteligência Artificial é a orientação para problemas que podem ser expressos em termos simbólicos. O

²A palavra *heurística* vem da palavra grega *heuriskein* que significa descobrir.

processamento simbólico engloba um conjunto de metodologias e de técnicas que foram criadas para resolver problemas não estruturados, que lidam com informação não rigorosa e incompleta, e que usam regras empíricas ganhas com a experiência. O processamento simbólico consiste na utilização de informação ou de conhecimento representado através de símbolos. Os símbolos são usados para representar objectos, situações, acontecimentos do mundo real e as propriedades a elas associadas. Os símbolos podem ser interligados, usando estruturas como grafos ou redes, para representar relações como hierarquias e dependências.

Uma outra característica da Inteligência Artificial é o recurso a heurísticas. Uma *heurística* é qualquer função para a resolução de um problema que não é um algoritmo ou que não foi demonstrado ser um algoritmo. As razões mais comuns para que uma sequência de instruções seja uma heurística e não um algoritmo são: (1) a execução das instruções não termina para algumas classes de problemas; (2) não se garante que produza uma situação em que o objectivo tenha sido atingido, devido a problemas com a própria heurística; ou (3) não se garante que produza uma situação em que o objectivo tenha sido atingido, por o problema que pretende resolver não ser bem definido.

A Inteligência Artificial estuda também técnicas para incorporar formas de raciocínio em máquinas. O raciocínio é feito através de um processo de inferência. Recorrendo a técnicas de raciocínio, a Inteligência Artificial permite um tipo diferente de programação: em vez de especificar o comportamento a atingir pelo sistema em função de sequências e repetições de acções, o programador fornece um conjunto de regras lógicas, semelhantes às usadas em actividades quotidianas. A responsabilidade de ligar essas regras de um modo lógico, com a finalidade de obter o comportamento desejado, é da exclusiva responsabilidade do componente do sistema encarregado do raciocínio. A independência das regras em relação ao componente do programa que sabe como usar essas regras é uma das características que distingue muitos dos sistemas de Inteligência Artificial. Esta separação é mantida para que o mesmo programa possa ser usado, mesmo quando um novo domínio requeira um novo conjunto de regras. Esta independência garante que, quando introduzimos uma nova regra, não tenhamos de pensar em todas as situações em que ela vai ser usada e como vai ser usada.

Com a Inteligência Artificial reconheceu-se a necessidade de utilizar conhecimento para se obter comportamento inteligente. Por outras palavras, não

basta possuir capacidade de raciocínio, é necessária uma grande quantidade de conhecimento para permitir que o raciocínio possa resolver problemas. A Inteligência Artificial desenvolve técnicas para representar, manipular e explicitar o conhecimento.

— *Interfaces Gráficas*

As técnicas de interfaces gráficas [Foley et al., 1997] têm sido tornadas possíveis devido ao desenvolvimento de novas tecnologias associadas à construção de unidades de visualização de informação (os monitores de computador), ao aumento da velocidade de processamento dos computadores e ao desenvolvimento de um conjunto de algoritmos que permite a manipulação e a visualização das entidades envolvidas.

O domínio das interfaces gráficas preocupa-se com a construção de abstracções de objectos e com a sua apresentação sob a forma de figuras. As interfaces gráficas estão hoje vulgarizadas através dos sistemas operativos do Macintosh e do Windows, nos quais a interacção entre o computador e o utilizador recorre à manipulação interactiva de ícones e de janelas. Nos últimos anos, o tratamento digital de imagens e a crescente utilização de técnicas de animação tem vulgarizado as aplicações das interfaces gráficas.

As interfaces gráficas, de um modo genérico, lidam com a síntese, em forma de figuras, de objectos reais ou imaginários. Recentemente, a utilização conjunta de documentos contendo texto, gráficos e imagens (conhecida por *multimédia*) está em franco desenvolvimento.

As técnicas de interfaces gráficas podem ser classificadas segundo vários aspectos:

- As *dimensões* utilizadas na representação do objecto (objectos bidimensionais ou objectos tridimensionais) e o *tipo de imagem* a ser produzida (a preto e branco, em tonalidades de cinzento ou com cor).
- O *tipo de interacção* com a imagem, o qual define qual o grau de controle que o utilizador tem sobre a imagem ou sobre o objecto.
- O *papel da imagem* em relação à sua utilização, ou seja, se a visualização da imagem corresponde ao objectivo final ou se corresponde a um mero meio para atingir outro objectivo.
- As *relações entre os objectos e as suas imagens*, as quais apresentam uma gama extensa de possibilidades, desde a utilização de uma única imagem

estática, à utilização de várias imagens que evoluem e interagem ao longo do tempo.

A tendência actual em interfaces gráficas concentra-se também na modelação dos objectos, não apenas na criação das suas imagens.

— *Bases de Dados*

As tecnologias de bases de dados são fundamentais para os *sistemas de informação* existentes em organizações. De um modo simples, uma *base de dados* corresponde a um repositório de informação, juntamente com um conjunto de programas, o *sistema de gestão da base de dados*, que acede, guarda e gere a informação contida na base de dados [Ramakrishnan e Gehrke, 2000]. Sob esta perspectiva, uma base de dados parece ser semelhante a um tipo abstracto de informação.

Uma base de dados pode ser considerada a três níveis de abstracção, o nível físico, o nível lógico e o nível da interface.

— *O nível físico*

Preocupa-se com a representação interna e com a organização da informação contida na base de dados. Este nível corresponde ao nível da representação de um tipo abstracto de informação. A este nível surgem diferentes preocupações associadas à programação: a *gestão de memória* aborda a utilização eficiente de enormes quantidades de informação, informação essa que é demasiado volumosa para ser totalmente carregada na memória do computador; a *gestão da concorrência* aborda a partilha da informação entre vários utilizadores em simultâneo, garantindo a coerência dos dados na base de dados; a *recuperação* desenvolve técnicas para garantir a salvaguarda da informação no caso de ocorrência de erros (quer por parte do programa que manipula a base de dados, quer por parte do utilizador), falhas no funcionamento do computador (por exemplo, originadas por quebras de corrente), ou utilização maliciosa.

— *O nível lógico*

Desenvolve modelos de dados que correspondem à organização conceptual da informação na base de dados. Destes modelos, o mais vulgarizado é o *modelo relacional* através do qual uma base de dados é considerada como um conjunto de tabelas, com eventuais relações entre si.

O nível lógico apresenta uma linguagem para a definição dos aspectos estruturais da informação (a linguagem de *definição de dados*) e uma

linguagem para aceder à informação contida na base de dados e para efectuar a actualização desta informação (a linguagem de *manipulação de dados*). Com estas linguagens desenvolvem-se programas específicos para a manipulação de bases de dados.

Certos aspectos tratados ao nível físico, como por exemplo, a gestão da concorrência, traduzem-se ao nível lógico em outros conceitos, por exemplo, o conceito de *transacção*.

— *O nível da interface*

Aborda a organização e apresentação da informação contida na base de dados. Este nível permite criar camadas de abstracção sobre a informação ao nível lógico, de modo a que diferentes utilizadores possam ter percepções diferentes do nível de granularidade da informação contida na base de dados.

As bases de dados são uma das tecnologias em que muito trabalho está a ser realizado, tendo surgido nos últimos anos *bases de dados de objectos* que combinam a programação com objectos com a utilização de bases de dados e as *bases de dados dedutivas* que combinam as capacidades de raciocínio desenvolvidas em Inteligência Artificial com o armazenamento de grandes quantidades de informação.

16.2.4 Sistemas operativos

O sistema operativo de um computador [Marques e Guedes, 1994], [Silberschatz et al., 2001], [Tanenbaum, 2001] é um programa que controla a execução de todos os programas dentro do computador – funciona como um supervisor de todas as operações efectuadas pelo computador.

Um sistema operativo corresponde a uma máquina virtual que esconde o funcionamento físico dos vários componentes e unidades ligadas a um computador – processador, memória, unidades de disco, impressoras, rede, etc.

Ao nível do sistema operativo, o utilizador fornece indicações tais como, “execute este programa”, “guarde esta informação”, “imprima esta informação”; cada uma destas instruções vai originar a execução de um programa, ao nível do sistema operativo, que efectua a acção necessária.

O sistema operativo tem várias funções específicas, entre as quais:

- Criar, manter e permitir o acesso ao sistema de ficheiros. Um *ficheiro* pode ser definido como um conjunto de informação – programas ou dados. O sistema operativo permite abstrair o modo como os ficheiros são armazenados internamente, fornecendo a possibilidade de nomear ficheiros e organizá-los em estruturas hierárquicas.
- Movimentar ficheiros entre as várias unidades de um computador, por exemplo, memória, disco e impressoras.
- Garantir a segurança de ficheiros através de um sistema de autenticação e de permissões.
- Permitir o funcionamento do computador em regime de *multitarefa* (do inglês “*multi-tasking*”). Este funcionamento permite que um computador realize várias tarefas em “simultâneo”, por exemplo, aceder à Internet, imprimir um ficheiro e executar um programa. Este funcionamento tira partido da grande velocidade de execução dos computadores e consiste em dividir a “atenção” do processador sequencialmente e repetitivamente por várias tarefas, dando a ideia ao utilizador de que estas estão a ser realizadas em simultâneo.
- Permitir a utilização simultânea do mesmo computador por vários utilizadores, a chamada *multiutilização*.

O desenvolvimento de sistemas operativos, que tem naturalmente ligações profundas com o estudo da *Arquitectura de Computadores*, aborda as seguintes grandes áreas:

- *Gestão de processos*

Um sistema operativo lida com vários processos computacionais, simultaneamente activos. A gestão dos vários processos envolve a possibilidade de interrupção temporária da execução de um processo e posteriormente a activação desse processo a partir do ponto de interrupção. Para isso, o sistema operativo, para cada processo, tem de manter e gerir um conjunto de variáveis de estado (semelhantes às descritas na Secção 7.2).

Ao lidar com vários processos surgem vários aspectos que têm de ser considerados, o *escalonamento* e *despacho* de processos que aborda a decisão de qual o processo a que deve ser dado o direito de execução e qual o tempo de execução antes da próxima interrupção, a *sincronização* entre processos,

que aborda a comunicação entre os processos e o tratamento de eventuals problemas de *bloqueamento* (em inglês, “*deadlocks*”) entre processos nos quais dois ou mais processos comunicantes entre si podem ficar mutuamente bloqueados à espera que certo acontecimento se verifique.

— *Gestão de memória*

O sistema operativo deve gerir a informação que está na memória do computador e providenciar o tráfego de informação entre a memória e o disco do computador (ou outras unidades de armazenamento).

Para além disso, o sistema operativo lida com o conceito de *memória virtual*, um modo de simular que o computador tem uma memória superior à sua memória física, através do recurso ao espaço existente em disco.

— *Sistema de ficheiros*

O sistema operativo tem de manter e gerir o sistema de ficheiros, uma camada de abstracção construída sobre a informação contida no disco (ou em outras unidades de armazenamento). O sistema de ficheiros deve contemplar a possibilidade da existência de informação distribuída por vários computadores ligados em rede.

Associados à utilização do sistema de ficheiros existem todos os problemas de protecção e segurança da informação.

— *Gestão de comunicações*

O sistema operativo tem a responsabilidade da transferência da informação que é lida ou escrita por um programa, escondendo os pormenores físicos da unidade utilizada para a leitura ou escrita.

16.3 Notas Finais

Este livro corresponde ao primeiro passo no mundo fascinante da programação. Muitos outros passos se devem seguir, aprofundando os conceitos aqui apresentados e aprendendo muitos outros. Paralelamente com o aprofundar dos conceitos de programação, é essencial programar, programar muito.

A utilização de várias linguagens de programação, pertencentes a vários paradigmas, é fundamental para dominar em pleno os conceitos apresentados neste livro.

Durante a aprendizagem de novas linguagens deve ser dada especial atenção ao modo como cada uma delas aborda os conceitos de programação, subindo acima dos pormenores sintácticos associados a cada linguagem.

Nos cerca de 60 anos da vida da Informática, largas centenas de linguagens de programação foram desenvolvidas. Algumas, frutos de modas e caprichos, tiveram uma vida muito curta, outras sobreviveram dezenas de anos. Seja qual for a evolução futura das linguagens de programação, os conceitos subjacentes às linguagens persistem. Daí a sua grande importância neste mundo em constante evolução que é a programação.

Apêndice A

Soluções de exercícios seleccionados

A.1 Exercícios do Capítulo 1

1. $\langle \text{inteiro} \rangle ::= \langle \text{dígitos} \rangle \mid -\langle \text{dígitos} \rangle \mid +\langle \text{dígitos} \rangle$
 $\langle \text{dígitos} \rangle ::= \langle \text{dígito} \rangle \mid \langle \text{dígito} \rangle \langle \text{dígitos} \rangle$
 $\langle \text{dígito} \rangle ::= 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0$
4. (a) Símbolos terminais: A, B, C, D, 1, 2, 3, 4
Símbolos não terminais: $\langle \text{idt} \rangle$, $\langle \text{letras} \rangle$, $\langle \text{numeros} \rangle$, $\langle \text{letra} \rangle$, $\langle \text{num} \rangle$
(b) ABCD Não pertence, dado que não apresenta nenhum símbolo correspondente a $\langle \text{numeros} \rangle$.
1CD Não pertence, dado que o seu primeiro símbolo não corresponde a $\langle \text{letras} \rangle$.
A123CD Não pertence, dado que os seus dois últimos símbolos não correspondem a $\langle \text{numeros} \rangle$.
AAAAAB12 Pertence
- (c) As frases da linguagem começam por um número arbitrário das letras A, B, C ou D (por qualquer ordem), existindo pelo menos uma destas letras, as quais são seguidas por um número arbitrário dos dígitos

1, 2, 3 ou 4 (por qualquer ordem), existindo pelo menos um destes dígitos.

```
5. <frase> ::= c <meio> r
   <meio> ::= <letra>+
   <letra> ::= a | d
```

A.2 Exercícios do Capítulo 2

```
1. print('Escreva um número de segundos')
segundos = eval(input('(um número negativo para terminar)\n? '))
while segundos > 0:
    dias = segundos / (24 * 60 * 60)
    print('O número de dias correspondente é', dias)
    print('Escreva um número de segundos')
    segundos = eval(input('(um número negativo para terminar)\n? '))

2. n1, n2, n3 = eval(input('Escreva três números separados por vírgulas\n? '))
if n1 > n2: # n2 não é o maior
    if n2 > n3: # n3 não é o maior
        maior = n1
    else: # o maior é ou n1 ou n3
        if n1 > n3:
            maior = n1
        else:
            maior = n3
    else: # n1 não é o maior
        if n2 > n3:
            maior = n2
        else:
            maior = n3
print('o maior é', maior)

5. num = eval(input('Escreva um inteiro positivo\n? '))
soma = 0

while num > 0:
```

```

# obtém o algarismo das unidades
digito = num % 10
# remove o algarismo das unidades de num
num = num // 10

if digito % 2 == 0: # o dígito é par
    soma = soma + digito

print('Soma dos dígitos pares:', soma)

6. num = eval(input('Escreva um inteiro positivo\n? '))
novo_num = 0
while num > 0:
    digito = num % 10
    num = num // 10
    novo_num = 10 * novo_num + digito
print('O número invertido é', novo_num)

7. x = eval(input('Qual o valor de x?\n> '))
n = eval(input('Qual o valor de n?\n> '))

soma = 0
termo = 1
pos_termo = 0

while pos_termo <= n:
    soma = soma + termo
    pos_termo = pos_termo + 1
    termo = termo * (x / pos_termo)

print('O valor da soma é', soma)

```

A.3 Exercícios do Capítulo 3

```

1. def cinco(n):
    return n == 5

```

```
3. def primo(n):
    from math import sqrt
    if n < 2:
        return False
    res = True
    i = 2
    while i <= int(sqrt(n)):
        if n % i == 0:
            res = False
            break
        else:
            i = i + 1
    return res

4. def n_esimo_primo(n):
    conta_primos = 0
    num = 1
    while conta_primos < n:
        if primo(num):
            conta_primos = conta_primos + 1
        num = num + 1
    return num - 1 # num foi incrementado após o if

6. def e_elevado_a(x):
    soma = 0

    # Cálculo do primeiro termo da série
    termo = 1
    soma = soma + termo
    n = 1

    while termo > 0.0001:
        termo = termo * x / n
        soma = soma + termo
        n = n + 1

    return soma
```

A.4 Exercícios do Capítulo 4

```

2. def conta_menores(t, n):
    el_menores = 0
    for e in t:
        if e < n:
            el_menores = el_menores + 1
    return el_menores

3. def maior_elemento(t):
    maior = t[0]
    for e in t:
        if e > maior:
            maior = e
    return maior

5. def junta_ordenados(t1, t2):
    res = ()
    i = 0
    j = 0
    while i < len(t1) and j < len(t2):
        if t1[i] < t2[j]:
            res = res + (t1[i], )
            i = i + 1
        else:
            res = res + (t2[j], )
            j = j + 1
    #um dos tuplos foi totalmente processado
    if i == len(t1): # trata dos elementos por juntar em t2
        res = res + t2[j:]
    if j == len(t2): # trata dos elementos por juntar em t1
        res = res + t1[i:]
    return res

6. def soma_elementos_atomicos(t):
    i = 0
    soma = 0

```

```

        while i < len(t):
            if isinstance(t[i], tuple):
                t = t[:i] + t[i] + t[i+1:]
            else:
                soma = soma + t[i]
                i = i + 1
        return soma

7. def seq_racaman(n):

    def racaman(n):
        # recorre à variável livre seq_r
        if n == 0:
            return 0
        elif seq_r[n-1] > n and \
             not seq_r[n-1] - n in seq_r:
            return seq_r[n-1] - n
        else:
            return seq_r[n-1] + n

    seq_r = ()
    for i in range(n):
        seq_r = seq_r + (racaman(i), )
    return seq_r

8. def reconhece(frase):
    i = 0
    while i < len(frase)-1 and \
          frase[i] in ['A', 'B', 'C', 'D']:
        i = i + 1
    if i > 0: # foi encontrada pelo menos uma letra
        inicio_nums = i
        while i <= len(frase)-1 and \
              frase[i] in ['1', '2', '3', '4']:
            i = i + 1
        if i == len(frase):
            return True
        else:

```

```

        return False
else:
    return False

```

A.5 Exercícios do Capítulo 5

1. def pertence(el, lst):
 i = 0
 while i < len(lst):
 if el == lst[i]:
 return True
 i = i + 1
 return False
3. def posicoes_lista(lst, e):
 resultado = []
 for i in range(len(lst)):
 if lst[i] == e:
 resultado = resultado + [i]
 return resultado
4. def parte(lst, e):
 menores = []
 maiores = []
 for el in lst:
 if el < e:
 menores = menores + [el]
 else:
 maiores = maiores + [el]
 return [menores, maiores]
6. Usamos a representação de uma matriz como sendo uma lista na qual cada elemento armazena uma linha da matriz. Deste modo, a matriz

$$\begin{array}{cccc}
 a_{11} & a_{12} & \cdots & a_{1n} \\
 a_{21} & a_{22} & \cdots & a_{2n} \\
 \cdots & \cdots & & \cdots \\
 a_{n1} & a_{n2} & \cdots & a_{nn}
 \end{array}$$

é representada pela lista $[[a_{11}, a_{12}, \dots, a_{1n}], [a_{21}, a_{22}, \dots, a_{2n}], \dots, [a_{n1}, a_{n2}, \dots, a_{nn}]]$

```
def elemento_matriz(mat, linha, col):
    if linha > len(mat)-1:
        print('Índice inválido: linha', linha)
    elif col > len(mat[0])-1:
        print('Índice inválido: coluna', col)
    else:
        return mat[linha][col]
```

9. Recorrendo à ordenação por seleção:

```
def ordena(lst):
    # criação da lista com índices
    ind = []
    for i in range(len(lst)):
        ind = ind + [i]

    for i in range(len(lst)):
        pos_menor = i
        for j in range(i + 1, len(lst)):
            if lst[ind[j]] < lst[ind[pos_menor]]:
                pos_menor = j
        ind[i], ind[pos_menor] = ind[pos_menor], ind[i]
    return ind
```

A.6 Exercícios do Capítulo 6

1. def soma_digitos_pares(n):
 if n == 0:
 return 0
 elif n % 2 == 0:
 return n % 10 + soma_digitos_pares(n // 10)
 else:
 return soma_digitos_pares(n // 10)

```
3. def soma_quadrados_lista(lst):
    return acumula(lambda x, y: x + y, \
                   transforma(quadrado, lst))

6. def faz_potencia(n):
    def f_p(x):
        return x ** n
    return f_p

7. def quick_sort(lst):

    def parte(lst, e):
        menores = []
        maiores = []
        for el in lst:
            if el < e:
                menores = menores + [el]
            elif el > e:
                maiores = maiores + [el]
        return [menores, maiores]

        if lst == []:
            return lst
        else:
            menores, maiores = parte(lst, lst[0])
            return quick_sort(menores) + \
                   [lst[0]] + \
                   quick_sort(maiores)

9. def rastro(nome, fn):

    def rastreada(arg):
        print('Avaliação de', nome, 'com argumento', arg)
        res = fn(arg)
        print('Resultado', res)
        return res

    return rastreada
```

```
10. def calcula_sqrt(n):
    return newton(lambda y : quadrado(y) - n, 1.0)
```

A.7 Exercícios do Capítulo 7

```
2. (a) def A(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return A(m-1, 1)
    else:
        return A(m-1, A(m, n-1))

(b) A(2, 2) =
A(1, A(2, 1)) =
A(1, A(1, A(2, 0))) =
A(1, A(1, A(1, 1))) =
A(1, A(1, A(0, A(1, 0)))) =
A(1, A(1, A(0, A(0, 1)))) =
A(1, A(1, A(0, 2))) =
A(1, A(1, 3)) =
A(1, A(0, A(1, 2))) =
A(1, A(0, A(0, A(1, 1)))) =
A(1, A(0, A(0, A(0, A(1, 0)))))) =
A(1, A(0, A(0, A(0, A(0, 1)))))) =
A(1, A(0, A(0, A(0, 2)))) =
A(1, A(0, A(0, 3))) =
A(1, A(0, 4)) =
A(1, 5) =
A(0, A(1, 4)) =
A(0, A(0, A(1, 3))) =
A(0, A(0, A(0, A(1, 2)))) =
A(0, A(0, A(0, A(0, A(1, 1)))))) =
A(0, A(0, A(0, A(0, A(0, A(0, A(1, 0))))))) =
A(0, A(0, A(0, A(0, A(0, A(0, 1))))))) =
A(0, A(0, A(0, A(0, A(0, A(0, 2)))))) =
A(0, A(0, A(0, A(0, 3)))) =
```

```
A(0, A(0, A(0, 4))) =
A(0, A(0, 5)) =
A(0, 7) =
7
```

- (c) A função gera um processo recursivo em árvore pois existem múltiplas fases de expansão e de contracção geradas pela dupla recursão em $A(m-1, A(m, n-1))$.

4. (a) `def produto(x, y):`

```
    if x == 0:
        return 0
    elif x % 2 == 0:
        return produto(x // 2, y + y)
    else:
        return y + produto(x - 1, y)
```

- (b) A função é recursiva porque se chama a si própria.

- (c) A função gera um processo recursivo linear, originando uma fase de expansão seguida de um fase de contracção, como se pode ver pelo seguinte exemplo:

```
produto(5, 6) =
6 + produto(4, 6) =
6 + produto(2, 12) =
6 + produto(1, 24) =
6 + (24 + produto(0, 24)) =
6 + (24 + 0) =
6 + 24 =
30
```

(d) `def produto(x, y):`

```
    def produto_aux(x, y, ac):
        if x == 0:
            return ac
        elif x % 2 == 0:
            return produto_aux(x // 2, y + y, ac)
        else:
            return produto_aux(x - 1, y, ac + y)
```

```

        return produto_aux(x, y, 0)

5. (a) def g(n):
        if n == 0:
            return 0
        else:
            return n - g(g(n-1))

(b) g(3)
    3-g(g(2))
    3-g(2-g(g(1)))
    3-g(2-g(1-g(g(0))))
    3-g(2-g(1-g(0)))
    3-g(2-g(1-0))
    3-g(2-g(1))
    3-g(2-g(1))
    3-g(2-(1-g(g(0))))
    3-g(2-(1-g(0)))
    3-g(2-(1-0))
    3-g(2-(1))
    3-g(1)
    3-(1-g(g(0)))
    3-(1-g(0))
    3-(1-0)
    3-(1)
    2

(c) A função gera um processo recursivo em árvore pois existem múltiplas
    fases de expansão e de contracção geradas pela dupla recursão em
    g(g(n-1)).

```

A.8 Exercícios do Capítulo 8

```

1. def conta_vogais(fich_e, fich_s):

    def pos(lst, el):
        for i in range(len(lst)):

```

```
if lst[i] == el:  
    return i  
  
f = open(fich_e, 'r', encoding='UTF-16')  
vogais = ['a', 'e', 'i', 'o', 'u']  
res = []  
for v in vogais:  
    res = res + [0]  
  
linha = f.readline()  
while linha != '':  
    for c in linha:  
        if c in vogais:  
            pos_vogal = pos(vogais, c)  
            res[pos_vogal] = res[pos_vogal] + 1  
    linha = f.readline()  
f.close()  
f = open(fich_s, 'w')  
for i in range(len(vogais)):  
    f.write(vogais[i] + ' ' + str(res[i]) + '\n')  
f.close()  
  
2. def junta(fich_1, fich_2, fich_s):  
    f_1 = open(fich_1, 'r', encoding='UTF-16')  
    f_2 = open(fich_2, 'r', encoding='UTF-16')  
    f_s = open(fich_s, 'w')  
  
    v_1 = f_1.readline()  
    v_2 = f_2.readline()  
    while v_1 != '' and v_2 != '':  
        if eval(v_1) < eval(v_2):  
            f_s.write(v_1)  
            v_1 = f_1.readline()  
        else:  
            f_s.write(v_2)  
            v_2 = f_2.readline()  
    if v_1 == '':
```

```

while v_2 != '':
    f_s.write(v_2)
    v_2 = f_2.readline()
if v_2 == '':
    while v_1 != '':
        f_s.write(v_1)
        v1 = f_1.readline()
f_1.close()
f_2.close()
f_s.close()

```

A.9 Exercícios do Capítulo 9

1. (a) [('Ana', '4150-036')]
 (b) 'Ana'
 (c) ('Fred', '33136')
 (d) 'F'
 (e) '3'
2. def conta_vogais(fich):
 f = open(fich, 'r', encoding='UTF-16')
 res = {'a':0, 'e':0, 'i':0, 'o':0, 'u':0}
 linha = f.readline()
 while linha != '':
 for c in linha:
 if c in ['a', 'e', 'i', 'o', 'u']:
 res[c] = res[c] + 1
 linha = f.readline()
 f.close()
 return res
3. def conta_vogais(fich_e, fich_s):
 f = open(fich_e, 'r', encoding='UTF-16')
 res = {'a':0, 'e':0, 'i':0, 'o':0, 'u':0}
 linha = f.readline()
 while linha != '':

```

for c in linha:
    if c in ['a', 'e', 'i', 'o', 'u']:
        res[c] = res[c] + 1
    linha = f.readline()
f.close()
f = open(fich_s, 'w')
for c in ['a', 'e', 'i', 'o', 'u']:
    f.write(c + ' ' + str(res[c]) + '\n')
f.close()

```

A.10 Exercícios do Capítulo 10

3.

1. (a) i. *Construtor:*

cria_ponto : $\mathbb{R}^2 \mapsto \text{ponto}$

ii. *Selectores:*

abcissa : *ponto* $\mapsto \mathbb{R}$

ordenada : *ponto* $\mapsto \mathbb{R}$

iii. *Reconhecedores:*

ponto : *universal* $\mapsto \text{lógico}$

origem : *ponto* $\mapsto \text{lógico}$

iv. *Teste:*

mesmo_ponto : $\mathbb{R}^2 \mapsto \text{lógico}$

(b) class ponto:

```

def __init__(self, x, y):
    if isinstance(x, (int, float)) and \
       isinstance(y, (int, float)):
        self.x = x
        self.y = y
    else:
        raise ValueError ('os argumentos não são números')

def abcissa(self):
    return self.x

```

```

def ordenada(self):
    return self.y

def origem(self):
    return abs(self.x) < 0.001 and \
           abs(self.y) < 0.001

def __eq__(self, outro):
    return abs(self.x - outro.abcissa()) < 0.001 and \
           abs(self.y - outro.ordenada()) < 0.001

def __repr__(self):
    return '(' + str(self.x) + ',' + \
           str(self.y) + ')'

(c) def distancia(p1, p2):
    return sqrt(quadrado(p1.abcissa() - p2.abcissa()) + \
                quadrado(p1.ordenada() - p2.ordenada()))

(d) def quadrante(p):
    if p.abcissa() >= 0:
        if p.ordenada() >= 0:
            return 1
        else:
            return 4
    else:
        if p.ordenada() >= 0:
            return 2
        else:
            return 3

5. (a) i. Construtor:
cria_carta : {espadas, copas, ouros, paus} × {A, 2, 3, 4, 5, 6, 7,
8, 9, 10, J, Q, K} ↦ carta

ii. Selectores:
naipe : carta ↦ {espadas, copas, ouros, paus}
valor : carta ↦ {A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K}

iii. Reconhecedor:

```

carta : universal \mapsto lógico

iv. *Teste:*

cartas_iguais : carta² \mapsto lógico

(b) `class carta:`

```
def __init__(self, naipe, valor):
    if naipe in ['espadas', 'copas', 'ouros', 'paus']:
        if valor in ['A', '2', '3', '4', '5', '6', '7',
                     '8', '9', '10', 'J', 'Q', 'K']:
            self.n = naipe
            self.v = valor
        else:
            raise ValueError ('valor não especificado')
    else:
        raise ValueError ('naipe não especificado')

def naipe(self):
    return self.n

def valor(self):
    return self.v
```

(c) `def todas_cartas():`

```
res = []
for n in ['espadas', 'copas', 'ouros', 'paus']:
    for v in ['A', '2', '3', '4', '5', '6', '7',
              '8', '9', '10', 'J', 'Q', 'K']:
        res = res + [carta(n, v)]
return res
```

(d) `def baralha(b):`

```
n_cartas = len(b)
for i in range(n_cartas):
    al = int(random()*n_cartas) + 1
    b[i], b[al] = b[al], b[i]
return b
```

A.11 Exercícios do Capítulo 11

```

1. class estacionamento:

    def __init__(self, lotacao):
        self.lotacao = lotacao
        self.ocupados = 0

    def entra(self):
        if self.lotacao == self.ocupados:
            print('Lotação esgotada')
        else:
            self.ocupados = self.ocupados + 1

    def sai(self):
        if self.lotacao == 0:
            print('O parque está vazio')
        else:
            self.ocupados = self.ocupados - 1

    def lugares(self):
        return self.lotacao - self.ocupados

3. class mem_A:

    def __init__(self):
        self.A = {}

    def val(self, m, n):
        if (m, n) in self.A:
            return self.A[(m, n)]
        else:
            if m == 0:
                self.A[(m, n)] = n + 1
            elif n == 0:
                self.A[(m, n)] = self.val(m-1, 1)
            else:

```

```
A_m_n_1 = self.val(m, n-1)
self.A[(m, n)] = self.val(m-1, A_m_n_1)
return self.A[(m, n)]
```


Bibliografia

- H. Abelson, G. Sussman, e J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 2 edition, 1996.
- ACM. Computing curricula 2001. Technical report, Interim Review Task Force, Association for Computing Machinery, 2000. www.acm.org/education/curric_vols/cc2001.pdf.
- ACM. Computer science curriculum 2008: An interim revision of CS 2001. Technical report, Interim Review Task Force, Association for Computing Machinery, 2008. www.computer.org/portal/cms_docs.ieeecs/ieeecs/education/cc2001/Computer%20Science2008.pdf.
- ACM. Computer science curricula 2013 (strawman draft). Technical report, Interim Review Task Force, Association for Computing Machinery and IEEE Computer Society, 2012. <http://ai.stanford.edu/users/sahami/CS2013>.
- A. Aho, R. Sethi, e J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Co., Reading, MA, 1986.
- V. S. Alagar e K. Periyasamy. *Specifications of Software Systems*. Springer-Verlag, Heidelberg, Alemanha, 1998.
- G. Arroz, J. Monteiro, e A. Oliveira. *Arquitectura de Computadores*. IST Press, Lisboa, Portugal, 2007.
- U. Ascher e C. Greif. *A First Course in Numerical Methods*. SIAM, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2011.
- A. Asperti e G. Longo. *Categories Types and Structures: An Introduction to Category Theory for the working computer scientist*. MIT Press, Cambridge, MA, 1991.
- L. Bass, P. Clements, e R. Kazman. *Software Architecture in Practice*. Addison-Wesley Publishing Co., Reading, MA, 1998.

- A. Bellos. *Alex no País dos Números: Viagem pelo Maravilhoso Mundo da Matemática*. Planeta Manuscrito, Lisboa, Portugal, 2012.
- N. L. Biggs. The roots of combinatorics. *Historia Mathematica*, 6:109–136, 1979.
- B. W. Boehm. *Software Engineering Economics*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.
- C. Boyer. *História da Matemática*. Edgard Blucher Lda., São Paulo, S.P., 1974.
- W. Brainerd e L. Landweber. *Theory of Computation*. John Wiley & Sons, New York, N.Y., 1974.
- F. Brooks. *The Mythical Man-Month*. Addison-Wesley Publishing Co., Reading, MA, 1975.
- A. W. Burks, D. W. Warren, e J. B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8 (46):53–57, 1954.
- N. Chomsky. Three models for the description of language. *IRE Trans. Infor. Theory*, 2(3):113–124, 1956.
- N. Chomsky. *Syntactic Structures*. Mouton, The Hague, The Netherlands, 1957.
- N. Chomsky. On certain formal properties of grammars. *Inf. and Control*, 2:137–167, 1959.
- A. Church. *The Calculi of Lambda Conversion*. Annals of Mathematics Studies. Princeton University Press, Princeton, N.J., 1941.
- J. Cohen. Garbage collection of linked data structures. *Computing Surveys*, 3(3): 341–367, 1981.
- T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3 edition, 2009.
- G. Cousineau e M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, Cambridge, U.K., 1998.
- O. Dahl e K. Nygaard. SIMULA: An ALGOL-based simulation language. *Comm. of the ACM*, 9(9):671–678, 1967.
- O. Dahl, E. Dijkstra, e C. Hoare. *Structured Programming*. Academic Press, New York, N.Y., 1972.

- N. Dale e H.M. Walker. *Abstract Data Types: Specifications, Implementations, and Applications*. D. C. Heath and Company, Lexington, MA, 1996.
- M. Davis. *O Computador Universal*. Editorial Bizâncio, Lisboa, Portugal, 2004.
- E. Dijkstra. *A Discipline of Programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.
- E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- J. Edmonds. *How to Think about Algorithms*. Cambridge University Press, New York, N.Y., 2008.
- A. E. Fischer e F. S. Grodzinsky. *The Anatomy of Programming Languages*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1993.
- J. D. Foley, A. van Dam, S. K. Feiner, e J. F. Huges. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, 2 edition, 1997.
- S. Ginsburg. *The Mathematical Theory of Context Free Languages*. McGraw-Hill Book Company, New York, N.Y., 1966.
- R. L. Graham, D. Knuth, e O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Publishing Co., Reading, MA, 1989.
- F. Hennie. *Introduction to Computability*. Addison-Wesley Publishing Co., Reading, MA, 1977.
- J. Herbrand. *Recherches sur la théorie de la démonstration*. PhD thesis, Sorbonne, Université de Paris, Paris, France, 1930.
- H. Hermes. *Enumerability, Decidability, Computability*. Springer-Verlag, New York, N.Y., 2 edition, 1969.
- C. Hoare. Notes on data structuring. In Dahl, Dijkstra, e Hoare, editors, *Structured Programming*. Academic Press, New York, N.Y., 1972.
- C. Hofmeister, R. Nord, e D. Soni. *Applied Software Architecture*. Addison-Wesley Publishing Co., Reading, MA, 2000.
- D.R. Hofstader. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, N.Y., 1979.
- D.R. Hofstader. *Gödel, Escher, Bach: Um Entrelaçamento de Gênios Brilhantes*. Imprensa Oficial SP, São Paulo, S.P., 2011.

- C. G. Hogger. *Essentials of Logic Programming*. Basil Blackwell Inc., Oxford, U.K., 1990.
- J. Hopcroft e J. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley Publishing Co., Reading, MA, 1969.
- P. Hudac. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.
- P. Jalote. *An Integrated Approach to Software Engineering*. Springer-Verlag, New York, N.Y., 2 edition, 1997.
- R. Johnsonbaugh. *Discrete Mathematics*. Prentice Hall, Englewood Cliffs, N.J., 7 edition, 2009.
- S. C. Kleene. *Introduction to Meta-mathematics*. American Elsevier Publishing Co., New York, N.Y., 1975.
- M. Kline. *Mathematical Thought: From Ancient to Modern Times*. Oxford University Press, Oxford, U.K., 1972.
- D. Knuth. *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*. Addison-Wesley Publishing Co., Reading, MA, 1973a.
- D. Knuth. *The Art of Computer Programming, Vol. 3, Sorting and Searching*. Addison-Wesley Publishing Co., Reading, MA, 1973b.
- E. Koffmann, P. Miller, e C. Wardle. Recommended curriculum for cs1, 1984. *Communications of the ACM*, 27(10):998–1001, 1984.
- E. Koffmann, P. Miller, e C. Wardle. Recommended curriculum for cs2, 1984. *Communications of the ACM*, 28(8):815–818, 1985.
- R. A. Kowalski. Algorithm = logic + control. *Comm. of the ACM*, 22(7):424–436, 1979.
- H. Ledgard. *Programming Proverbs*. Hayden Book Co., New York, N.Y., 1975.
- W. Lenhert e M. Ringle. *Strategies for Natural Language Processing*. Lawrence Erlbaum Associates, Hillsdale, N.J., 1982.
- B. Liskof e J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, MA, 1986.
- B. Liskof e S. Zilles. Programming with abstract data types. *SIGPLAN Symp. on Very High Level Languages – SIGPLAN Notices*, 9(4):50–59, 1974.

- M. Machtey e P. Young. *An Introduction to the General Theory of Algorithms*. North Holland, New York, N.Y., 1978.
- B. J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley Publishing Co., Reading, MA, 1990.
- Z. Manna e R. Waldinger. *Logical Basis for Computer Programming, Volume 1: Deductive Reasoning*. Addison-Wesley Publishing Co., Reading, MA, 1985.
- M. M. Mano e C. Kime. *Logic and Computer Design Fundamentals*. Prentice-Hall International, Englewood Cliffs, N.J., 1997.
- J. A. Marques e P. Guedes. *Fundamentos de Sistemas Operativos*. Editorial Presença, Lisboa, Portugal, 3 edition, 1994.
- J. A. Marques e P. Guedes. *Tecnologia dos Sistemas Distribuídos*. FCA – Editora de Informática, Lisboa, Portugal, 1998.
- J. P. Martins e M. R. Cravo. *Programação em Scheme: Introdução à Programação Utilizando Múltiplos Paradigmas*. IST Press, Lisboa, Portugal, 2 edition, 2007.
- J. J. McConnell. *Analysis of Algorithms: An Active Learning Approach*. Jones and Bartlett Publishers, Sudbury, MA, 2 edition, 2008.
- B. Meyer. *Object Oriented Software Construction*. IEEE Computer Society Press, NEW YORK, N.Y., 2 edition, 1997.
- M. L. Modesto. *Cozinha Tradicional Portuguesa*. Editorial Verbo, Lisboa, Portugal, 1982.
- J. W. Moore. *Software Engineering Standards: A User's Road Map*. IEEE Computer Society, Los Alamitos, CA, 1998.
- B. Moret. *The Theory of Computation*. Addison-Wesley Publishing Co., Reading, MA, 1997.
- S. L. Pfleeger e J. M. Atlee. *Software Engineering*. Prentice-Hall Inc., Englewood Cliffs, N.J., 4 edition, 2010.
- R. Ramakrishnan e J. Gehrke. *Database Management Systems*. Mc-Graw-Hill Book Company, New York, N.Y., 2 edition, 2000.
- B. Raphael. *The Thinking Computer: Mind Inside Matter*. W.H. Freeman, S. Francisco, CA, 1976.
- J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

- S. Russell e P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, N.J., 3 edition, 2010.
- S. Schach. *Object-Oriented and Classical Software Engineering*. Mc-Graw-Hill Book Company, New York, N.Y., 6 edition, 2005.
- G. M. Schneider, S. Weingart, e D. Perlman. *An Introduction to Programming and Problem Solving with Pascal*. John Wiley & Sons, New York, N.Y., 1978.
- M. Schönfinkel. On the building blocks of mathematical logic. In Heijenoort, editor, *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1977.
- M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.
- C. Sernadas. *Introdução à Teoria da Computação*. Editorial Presença, Lisboa, Portugal, 1993.
- D. Shell. A highspeed sorting procedure. *Comm. of the ACM*, 2(7):30–32, 1959.
- A. Silberschatz, P. B. Galvin, e G. Gagne. *Operating System Concepts*. John Wiley & Sons, New York, N.Y., 6 edition, 2001.
- H. Simmons. *An Introduction to Category Theory*. Cambridge University Press, Cambridge, U.K., 2011.
- S. Sing. *The Code Book: The Secret History of Codes and Code-Breaking*. Fourth Estate, London, U.K., 1999.
- M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co., Boston, MA, 3 edition, 2012.
- I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Co., Reading, MA, 5 edition, 1996.
- A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall Inc., Englewood Cliffs, N.J., 2 edition, 2001.
- A. L. Taylor. The wizard inside the machine. *TIME*, 123(16):42–49, 1984.
- F. Turbak, D. Gifford, e M. A. Sheldon. *Design Concepts in Programming Languages*. MIT Press, Cambridge, MA, 2008.
- A. J. Turner. A summary of the ACM/IEEE-CS joint curriculum task force report: Computing curricula 1991. *Communications of the ACM*, 34(6):69–84, 1991.

L. B. Wilson e R. G. Clark. *Comparative Programming Languages*. Addison-Wesley Publishing Co., Reading, MA, 1988.

N. Wirth. *Systematic Programming: An Introduction*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1973.

N. Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

Índice

- `!=`, 53
- `*`, 19, 41, 43, 44, 119, 130, 142, 309
- `+`, 19, 41, 43, 44, 119, 130, 142, 308
- `-`, 41, 43, 44, 309
- `/`, 43, 44, 309
- `//`, 41
- `::=`, 19
- `<`, 53
- `<=`, 53
- `=`, 46
- `==`, 53, 309
- `>`, 53
- `>=`, 53
- `#`, 68, 326
- `%`, 41
- `|`, 19
- abordagem do topo para a base, 90, 436
- abstracção, 80, 264, 320, 430, 436
 - construção de, 437
 - de controle, 437
 - de dados, 264, 438
 - de máquinas, 437
 - procedimental, 81, 82, 109, 264, 437
- `actualiza_ciclo`, 385
- `actualiza_dist`, 259
- açúcar sintáctico, 54
- `acumula`, 188
- acumulador, 188
- ADT, 270
- al-Khowarizm, Abu Ja'far Mohammed ibu-Musa, 6
- Alexandria, Heron de, 88
- ALGOL, 98
- algoritmo, 6, 430, 445
 - de Dijkstra, 254
 - de Euclides, 87, 179
- `alisa`, 121, 122
- ambiente, 48
 - global, 79, 100
 - local, 78, 80
- amontoado, 421
- análise
 - do problema, 318, 439
 - dos requisitos, 319
- `and`, 45
- anonimato da representação, 281
- Argand, Jean-Robert, 265
- Arquitectura de Computadores, 436, 450
- Arquitectura de Programas, 439
- árvore, 389
 - binária, 390
 - de procura, 402
 - filha, 389
 - folha, 389
 - mãe, 389
 - ordenação por, 402
 - raiz, 389
 - ramo, 389
 - vazia, 389
- `arvore` (classe), 400
 - `__init__`, 400
 - `__repr__`, 401
- `arv_dir`, 401

arv_esq, 401
arv_vazia, 401
raiz, 401
arvore (recorrendo a funções)
 arv_dir, 397
 arv_esq, 397
 arv_iguais, 398
 arv_vazia, 398
 arvore, 397
 cria_arv, 397
 escreve_arv, 398
 nova_arv, 397
 raiz, 397
ASCII, 132
assinatura de um tipo, 273
atribuição, 46
avalia_posfixa, 348

Bachmann, Paul, 167
Backus, John, 28
balanceados, 346
base de conhecimento, 443
base de dados, 448
 de objectos, 449
BASIC, 441
Becker, Joe, 132
biblioteca, 109
boa_aprox, 195
bom_palpite, 91
Bombelli, Rafael, 265
bool, 45
Boole, George, 45
break, 66, 124
Brooks Jr., Frederick P., 330

C, viii, 12, 422, 441
C++, viii, 443
cadeia de caracteres, 35, 37, 127
 de documentação, 127
 vazia, 127
caixa (classe), 361, 375
 --init--, 362, 375
 repr, 363, 381
 actualiza_interface, 378
 apaga_clientes, 378
 aumenta_clientes_atendidos, 362, 377
 clientes_atendidos, 363, 379
 existem_caixas_com_fila, 381
 fila_caixa, 362, 379
 fila_graf_caixa, 380
 icon_instante_pronta, 380
 incremento_icons_caixas, 380
 info_caixa, 363, 379
 info_produtos, 363, 379
 info_t_espera, 363, 380
 muda_info_caixa, 362, 377
 muda_info_produtos, 362, 378
 muda_info_t_espera, 362, 378
 num_caixas, 379
 num_caixa, 362
 pos_fila_graf_caixa, 380
 pos_icon_nb_clientes, 380
 pos_inicial_icons_clientes, 380
 pos_novo_icon_fila, 380
 produtos_processados_ciclo, 380
 produtos_processados, 363
 rep_graf_cliente_na_fila, 378
 calc_termo, 95
 calcula_raiz, 89, 198
cálculo lambda, 185
carácter
 de escape, 56
 de pronto, 34, 62, 77
cdm, 257
Chomsky, Noam, 28
Church, Alonzo, 4, 185
ciclo, 65, 197
 contado, 123
 controle do, 65
 corpo do, 65
 infinito, 65

lê-avalia-escreve, 34
 passagem pelo, 65
 cifra de substituição, 134
class, 287
 classe, 287, 295
 hierarquia de, 302
cliente (classe), 359, 373
 `__init__`, 359, 373
 `__repr__`, 359, 374
 `apaga_cliente`, 374
 `artigos`, 359, 374
 `cria_icons_cliente`, 373
 `mostra_cliente`, 374
 `rep_grafica`, 374
 `tempo_entrada`, 359, 374
Clos, 443
close, 233, 236
codifica, 135
codificador, 136
 coerção, 44
 Collins, Lee, 132
 comando, 35
 comentário, 68, 326
 compilador, 435
compl (classe), 283
 `__add__`, 308, 310
 `__eq__`, 310
 `__init__`, 283
 `__mul__`, 310
 `__repr__`, 311
 `__sub__`, 310
 `__truediv__`, 310
 `compl_iguais`, 278, 284
 `compl_zero`, 284, 310
 `escreve`, 284
 `imag_puro`, 284
 `p_imag`, 284, 310
 `p_real`, 284, 309
 `compl_zero`, 277
 complexidade do algoritmo, 432
complexo, 277
 comportamento
 FIFO, 351
 LIFO, 333
 computabilidade, 432
 computador, 4
 características, 4
 condição, 53
 conhecimento
 declarativo, 93
 procedimental, 93
 conjunto
 de chegada, 75
 de partida, 75
 constante, 35
 construtores, 271
conta (classe), 294
 `__init__`, 294
 `consulta`, 294
 `deposito`, 294
 `levantamento`, 295
conta_gen (classe), 298, 300
 `__init__`, 298
 `consulta`, 298
 `deposito`, 298
 `levantamento`, 298
conta_jovem (classe), 301
 `levantamento`, 301
conta_jovem_com_pin (classe), 303, 304
 `__init__`, 303, 304
 `accede`, 303, 305
 `altera_codigo`, 304
 `consulta`, 305
 `deposito`, 304
 `levantamento`, 304
conta_letras, 249
conta_ordenado (classe), 300
 `__init__`, 300
 `levantamento`, 300
 contracção, 208

- controle da complexidade, 320
cria_compl, 266, 268, 276
crivo, 150, 151
Crivo de Eratóstenes, 149
- dados persistentes, 229
Davis, Mark, 132
de Pisa, Leonardo, 215
debugging, 323
definição, 35, 76, 282
 de classe, 282
 de função, 76
 recursiva, 21, 173, 197
del, 142, 271
depuração, 27, 322
 da base para o topo, 324
depurar, 322
derivada, 193
derivada num ponto, 192
derivada_ponto, 192
descodifica, 135
desenvolvimento
 da solução, 439
 do topo para a base, 320
dicionário, 243
 chave, 243
 valor, 243
 vazio, 244
dict, 243
dificuldade *NP*, 432
dificuldade *P*, 432
Dijkstra, Edsger, 254, 328, 438
dist_inicial, 258
dist_min, 258
divide_compl, 267
documentação
 de concepção, 320
 externa, 326
 interna, 326
 técnica, 325, 326
- de utilização, 325
domínio
 de um nome, 106
estático, 106
- efeito, 58
Eiffel, 443
encapsulação da informação, 281
engenharia
 da programação, 316
informática, 2
entidade
 com estado, 294
computacional, 35
 recursiva, 173
Eratóstenes, 149
- erro
 absoluto, 90
 relativo, 90
semântico, 26, 324
sintáctico, 26, 323
escreve_compl, 268
espaço de nomes, 48
estrutura
 circular, 425
 dinâmica, 407
 estática, 407
 linear, 333
estrutura de blocos, 98
Euclides, 87
eval, 56
evolução local, 205
ex_fn, 400
existem_caixas_com_fila, 366
expansão, 208
expressão, 35, 46
 composta, 35, 37
 designatória, 74, 243
- factorial**, 86, 175, 186, 206, 209, 210
factorial_aux, 209

False, 37, 45
fib, 216, 218, 219
fib_aux, 218
Fibonacci, Leonardo, 215
ficheiro, 229, 450

- abertura de, 230
- close**, 233, 236
- fecho de, 233
- indicador
 - de escrita, 236
 - de leitura, 232
- modo de escrita, 230
- modo de leitura, 230
- open**, 230
- print**, 237
- read**, 233
- readline**, 232
- readlines**, 233
- write**, 236
- writelines**, 236

- FIFO, 351
- fila** (classe), 356
- __init__**, 356
- __repr__**, 357
- coloca**, 356
- comprimento**, 356
- fila_para_listा**, 356
- fila_vazia**, 357
- filas_iguais**, 357
- inicio**, 356
- retira**, 356
- fila, 351
- de prioridades, 387
- file**, 229
filtra, 187
filtro, 187
float, 41
float (como função), 44
for, 123
FORTRAN, 441
- FP, 442
função, 74, 243
- anónima, 185
- aplicação de, 75
- argumento, 74
 - número arbitrário de, 399
- chamada à, 75, 78
- como valor de função, 192
- contradomínio, 74
- corpo da, 76
- de Ackermann, 227, 313
- de avaliação, 56
- definição, 75, 76, 399
 - por abstracção, 74
 - por compreensão, 74
 - por enumeração, 74
 - por extensão, 74, 243
- derivada, 193
- domínio, 74
- factorial, 86
- indefinida, 75
- pública, 102
- recursiva, 177
- valor devolvido por, 79
- função em informática
- vs. função matemática, 86
- gestão
- de comunicações, 451
- da concorrência, 448
- de processos, 450
- de memória, 448, 451
- global**, 107
Gödel, Kurt, 4
grafo, 254
- arco, 254
 - dirigido, 254
 - rotulado, 254
- caminho, 254
- nó, 254

- gramática, 17
ambígua, 22
GraphWin (classe), 370
 Circle, 371
 Image, 371
 Rectangle, 371
 Text, 371
 close, 371
 draw, 372
 getMouse, 371
 setTextColor, 371
 undraw, 372
 update, 371
grau de dificuldade de problema, 168
guião, 59

hanoi, 223
hardware, 3
harware, 429
Haskell, 442
help, 128
herança, 302
heurística, 445

identidade, 184
if, 60
imag_puro, 277
in, 119, 130, 142, 246
inc1, 184
inc2, 184
indicador
 de escrita, 236
 de leitura, 232
índice, 116, 117
informação, 2
informática, 3
insere_arv, 405
insere_elemento, 405
instância, 287
instrução, 35
 break, 66, 124
de atribuição, 46, 196
múltipla, 50
simples, 47
de importação, 109
for, 123
global, 107
help, 128
if, 60
raise, 92
try-except, 340
vazia, 67
while, 65
int, 41
int (como função), 44, 272, 352
Inteligência Artificial, 445
interface gráfica, 447
interpretador, 14, 434
introduz_notas, 253
inv_quadrado, 184
isinstance, 121, 272, 276, 286

Java, viii, 12, 443

Kleene, Stephen C., 4
Kramp, Christian, 86

lambda, 185
Landau, Edmund, 167
Ledgard, Henri, 321
lei de Murphy, 318
len, 119, 130, 142, 246, 271
LIFO, 333
linguagem
 assembly, 13
 de alto nível, 13, 59, 434
 de programação, 4, 6, 12, 433
 estruturada em blocos, 98
 imperativa, 46
 máquina, 12, 59
 processador de, 14
 semântica, 25

- sintaxe, 17
- LISP**, 442
- list**, 141
- list** (como função), 142
- lista** (classe), 418
 - _eq_**, 420
 - _init_**, 413, 418
 - _repr_**, 421
 - comprimento**, 419
 - elem_pos**, 413, 418
 - em**, 420
 - insere_pos**, 414, 419
 - lista_vazia**, 420
 - muda_elem_pos**, 418, 420
 - remove_pos**, 416, 419
- lista, 141
 - associativa, 243
 - circular, 426
 - funcionais sobre, 187
 - ligada, 169, 411
 - vazia, 142
 - listas paralelas, 161
 - lista_para_arvore**, 405
 - lista_tel**, 161
 - lixo, 417, 422
- manutenção de um programa, 330
- máquina de inferência, 443
- máquina virtual, 434
- Matemática Discreta, 432
- matriz, 171
- mdc**, 87, 180
- met_newton**, 195
- met_intervalo**, 190
- metalinguagem, 25
- método
 - aproximado, 89, 94
 - de *Curry*, 201
 - definição de, 282
 - do intervalo, 189
- de Newton, 194
- passagem de parametros, 146
- Miranda, 442
- ML, 442
- modelo, 2
 - da cascata, 317
 - do programa armazenado, 440
 - de von Neumann, 440
- modificadores, 271
- modularidade da solução, 320
- módulo, 109
- mostra_janela_inicial**, 385
- move**, 221, 223
- move_disco**, 223
- multimédia, 447
- multiplica_compl**, 267
- multiutilização, 450
- Naur, Peter, 28
- no** (classe), 409
 - _init_**, 409
 - _repr_**, 409
 - muda_prox**, 409
 - muda_val**, 409
 - prox**, 409
 - val**, 409
- nome, 35, 46
 - composto, 47, 110, 232, 248, 283, 285
 - dominio de, 106
 - global, 105
 - indexado, 47, 117
 - livre, 105
 - local, 104
 - não local, 105
 - reservado, 47
 - simples, 47
- None**, 396
- nos_do_grafo**, 258
- not**, 45
- not in**, 119, 130, 142, 246

- notação
 BNF, 18
 científica, 36
 de Bachmann-Landau, 167
 O , 432
 o , 432
 Omaiúsculo, 167
 Ω , 432
 pós-fixa, 347
 Θ , 432
novo_palpite, 90
numero, 276
número, 35
 aleatório, 360
 complexo, 265
 de Fibonacci, 216
 inteiro, 36
 primo, 113, 149
 pseudo-aleatório, 360
 real, 36
 triangular, 114
- objecto, 282
 definição de, 282, 300
 definição de método, 282
 método, 282
 `__add__`, 308
 `__eq__`, 309
 `__init__`, 283, 284
 `__mul__`, 309
 `__repr__`, 309
 `__sub__`, 309
 `__truediv__`, 309
obtem_info, 253
open, 230
operação
 embutida, 37
 escrita de dados, 57
 leitura de dados, 54
polimórfica, 307
- sobrecarregada, 43, 307
operações
 básicas de um tipo, 271
operador, 37
 de atribuição, 46
 prioridade de, 38
operando, 37
or, 45
ordem de crescimento, 163
ordena, 158–160
ordena_arvore, 405
ordenação
 por árvore, 402
 por borbulhamento, 158
 por selecção, 160
 quick sort, 201
 Shell, 159
- p_imag**, 267, 268, 276
p_real, 267, 268, 276
paradigma de programação, 196, 440
paragrafação, 61
parâmetro
 concreto, 77
 formal, 75, 399
 número arbitrário de, 399
Pascal, 441
passagem
 por referência, 148
 por valor, 146
percorre, 405
piatorio, 200
pilha (classe), 343
 `__init__`, 343
 `__repr__`, 344
 empurra, 344
 pilha_vazia, 344
 tira, 344
 topo, 344
pilha (imutável), 342

empurra, 342
mostra_pilha, 343
nova_pilha, 342
pilha_vazia, 343
pilhas_iguais, 343
pilha, 343
tira, 342
topo, 342
pilha, 333
pixel, 370
polimorfismo, 307
ponteiro, 408
ponto (classe), 381
 __init__, 381
 __repr__, 382
 pos_x, 382
 pos_y, 382
pos, 151
Post, Emil Leon, 4
potencia, 85, 96, 99, 197, 206, 211, 212
potência rápida, 225
potencia_ainda_mais_estranha, 107
potencia_ainda_mais_estranha_2, 108
potencia_aux, 97
potencia_estranha, 105
potencia_estranha_2, 105
potencia_rapida, 225
potencia_tambem_estranha, 105
predicado, 53
 embebido, 53
print, 58, 237, 309
prioridade de operador, 38
processa_resultados, 366, 385
processador de linguagem, 433, 435
processamento interativo, 34
processo
 computacional, 3, 205, 430
 iterativo, 214, 215
 linear, 214
 recursivo, 208, 215
em árvore, 215
linear, 209
procura, 153, 154
procura
 binária, 154
 linear, 153
 sequencial, 153
prog_ordena, 156
programa, 2–4, 11, 58, 59, 317, 430
 fonte, 435, 438
 objecto, 435
programação, 3, 430
 com objectos, 442
 em lógica, 443
 funcional, 196, 441
 imperativa, 196, 440
 paradigma de, 196, 440
PROLOG, 444
pseudónimo, 144
quadrado, 81, 184, 193
raciocínio, 446
raiz, 91, 190, 198
raise, 92
range, 124
read, 233
readline, 232
readlines, 233
recolha de lixo, 423
 marcação, 423
 varrimento, 423
reconhecedores, 272
recursão
 definição, 173
 em árvore, 215, 217
 em funções, 214
 em processos, 215
 parte
 básica, 179
 recursiva, 179

- partes da definição, 179
refinamento por passos, 320
regime de multitarefa, 450
remove_multiplos, 151
representação
 declarativa, 444
 em vírgula flutuante, 41
 externa, 35, 43, 116, 268, 338, 354
 interna, 35, 268
 procedimental, 444
return, 78
round, 44, 272

seletores, 271
semântica, 25, 433
 de uma linguagem, 25
sentinela, 66
sequênciação, 59
sessão em Python, 34
Shell, Donald, 159
símbolo
 continuação em Python, 95
 não terminal, 18
 terminal, 18
simbolos_comum, 130
simbolos_comum_2, 131
simula_supermercado, 364, 382
simulação, 358
sin, 95
sintaxe, 17, 433
 de uma linguagem, 17
sistema
 computacional, 317
 de ficheiros, 451
 de gestão da base de dados, 448
 de informação, 448
 operativo, 449
Smalltalk, 443
software, 429
soma_compl, 267, 307
soma_elementos, 121, 124, 125, 128, 207, 213
soma_inteiros, 182
soma_inv_quadrados_impar, 182
soma_quadrados, 182
somatorio, 184–186
str, 127
str (como função), 130, 309, 352
Stratchey, Christopher, 180
string, 37
subclasse, 298–300
 herança, 299, 300
substitui, 120
subtrai_compl, 267
suf_pequeno, 95
suf_perto, 190

tabela, 141
 de dispersão, 243
teoria das categorias, 288
Teoria da Computação, 432, 433
teste
 de aceitação, 329
 casos de, 328
 de integração, 329
 de módulo, 322
 de sistema, 329
 unitário, 322
testes, 272
tipo
 de informação, 39, 270
 dinâmico, 334, 407
 domínio do, 39
 elementar, 40
 elemento do, 39
 em Python
 bool, 45
 dict, 243
 file, 229
 float, 41

int, 41
list, 141
None, 396
str, 37, 127
tuple, 115
 estruturado, 40
 imutável, 118, 128, 273
 mutável, 141, 243, 273, 293
 tipos abstractos de informação, 270
 construtores, 271, 273
 modificadores, 271
 reconhecedores, 272, 273
 selectores, 271, 273
 testes, 272, 274
 transformadores, 272, 352
 Torre de Hanói, 220
transf_newton, 195
transforma, 187
 transformada de Newton, 194
 transformador, 187
 de entrada, 274
 de saída, 274
 transformadores, 272, 352
trata_clientes, 363, 365, 384
troca, 146
troca_2, 148
True, 37, 45
tuple, 115
tuple (como função), 119
tuplo, 115
 vazio, 116
tuplo_ordenado, 126
 Turing, Alan, 4
type, 286, 306
 Unicode, 132, 235
 valor lógico, 35, 37
 valor sentinela, 66
 van Rossum, Guido, viii
 variável, 46