

## Capítulo 1

# Computadores, algoritmos e programas

*'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.*  
*'I don't much care where -' said Alice.*  
*'Then it doesn't matter which way you go,' said the Cat.*  
*'- so long as I get somewhere,' Alice added as an explanation.*  
*'Oh, you're sure to do that,' said the Cat, 'if you only walk long enough.'*

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 corretamente. Por *arte* entende-se a avaliação correta, 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

## 2 CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

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 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 objeto físico, por exemplo, uma ponte ou um avião, seja num objeto 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, diretamente mensuráveis (por exemplo, a gravidade, os campos elétricos e magnéticos) e com objetos materiais que interagem com essas forças (por exemplo, rodas dentadas, vigas, circuitos), a Engenharia Informática lida com entidades intangíveis que apenas podem ser observadas indiretamente através dos efeitos que produzem.

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

Sob a perspetiva 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 diretamente 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 eletrónicos, 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 eletrónica”, 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 objetivos da Informática corresponde ao estudo e desenvolvimento de entidades abstratas geradas durante a execução de programas – os processos computacionais. Um *processo computacional* é um ente imaterial que evolui ao longo do tempo, executando ações que levam à solução de um problema. Um processo computacional pode afetar objetos existentes no mundo real (por exemplo, guiar a aterragem de um avião, distribuir dinheiro em caixas multibanco, comprar e vender ações na bolsa), pode responder a perguntas (por exemplo, indicar quais as páginas da internet que fazem referência a um dado termo), entre muitos outros aspetos.

A evolução de um processo computacional é ditada por uma sequência de instruções a que se chama *programa*, e a atividade de desenvolver programas é chamada *programação*. A programação é uma atividade intelectual fascinante, que não é difícil, mas que requer muita disciplina. O principal objetivo 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 atividades e técnicas que são comuns às utilizadas em projetos 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, eletrónicos 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 *exatamente* 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 efetuar 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. Deveremos, 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 exatamente as funções que podem ser calculadas através de um computador. Este facto é expresso através da *tese de Church-Turing*<sup>1</sup>.

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 efetuam 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 *eletrónico*. A palavra “eletrónico” refere-se aos componentes da máquina, componentes esses que são responsáveis pela grande velocidade das operações efetuadas por um computador.

Um computador é também *digital*. Um computador efectua operações sobre informação que é codificada recorrendo a duas grandezas discretas (tipicamente

---

<sup>1</sup>Uma 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 and Landweber, 1974], [Hennie, 1977] ou [Kleene, 1975].

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.

## 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 algoritmo<sup>2</sup>. 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, 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 a um objetivo, ou seja, à solução de um dado problema. A execução das instruções do algoritmo garante que o seu objetivo é atingido.

---

<sup>2</sup>A 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 efetuar operações aritméticas com números decimais. Al-Khowarizmi foi ainda o criador do termo “Álgebra” (ver [Boyer, 1974], páginas 166–167).

$  \begin{array}{r}  468 \\  +37 \\  \hline  \end{array}  $	$  \begin{array}{r}  1 \\  468 \\  +37 \\  \hline  5  \end{array}  $	$  \begin{array}{r}  1 \\  468 \\  +37 \\  \hline  05  \end{array}  $	$  \begin{array}{r}  468 \\  +37 \\  \hline  505  \end{array}  $
(a)	(b)	(c)	(d)

Figura 1.1: Aplicação do algoritmo para somar dois números.

### 1.2.1 Exemplos de algoritmos

Um dos primeiros algoritmos que nos é ensinado na instrução primária é o algoritmo para somar dois números arbitrariamente grandes: escrevemos os números um sobre o outro com os algarismos das unidades alinhados e traçamos uma linha horizontal por baixo dos números (Figura 1.1 (a)) e começamos a trabalhar na coluna da direita, somando todos os algarismos dessa coluna, o algarismo das unidades da soma resultante é escrito na mesma coluna por baixo da linha horizontal e, se a soma originar um número superior a 9, o algarismo das dezenas é colocado no topo da coluna imediatamente à esquerda, dizendo-se vulgarmente “e vai um” (Figura 1.1 (b)). Este processo é repetido para cada uma das colunas da direita para a esquerda até que não existam mais colunas para somar (Figura 1.1 (c), (d)). No final, a soma aparece debaixo da linha horizontal.

À medida que crescemos, apercebemo-nos que a descrição de sequências de ações para atingir objetivos tem um papel fundamental na nossa vida quotidiana e está relacionada com a nossa facilidade 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 objetos, etc.

Vamos examinar algumas sequências de instruções utilizadas na nossa vida quotidiana. Consideremos, em primeiro lugar, a receita de “Rebuçados de ovos”<sup>3</sup>:

## REBUÇADOS DE OVOS

500 g de açúcar;

2 colheres de sopa de amêndoas peladas e raladas;

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

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 objetos a manipular; (2) uma descrição das ações que devem ser executadas sobre esses objetos. 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 1.2<sup>4</sup>.

#### 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.

---

<sup>4</sup> Adaptado das instruções para montar um papagaio voador oferecido pela Telecom Portugal.

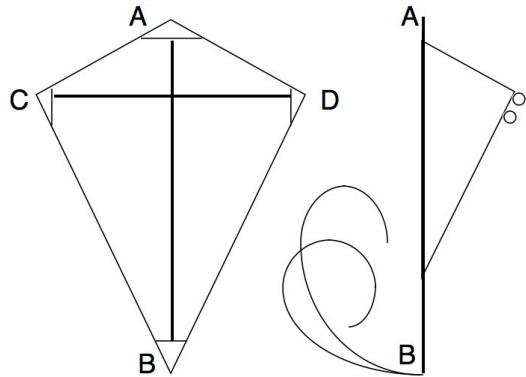


Figura 1.2: Diagrama para montar o papagaio voador.

As instruções para montar o papagaio voador são constituídas por uma descrição implícita dos objetos a manipular (mostrados na Figura 1.2) 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 1.3. Nesta figura, para além de um mapa ilustrativo, aparecem no lado esquerdo uma sequência detalhada de instruções 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

A sequência de passos de um algoritmo deve ser executada por um agente, o qual pode ser humano, mecânico, eletrónico, 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

## 10 CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

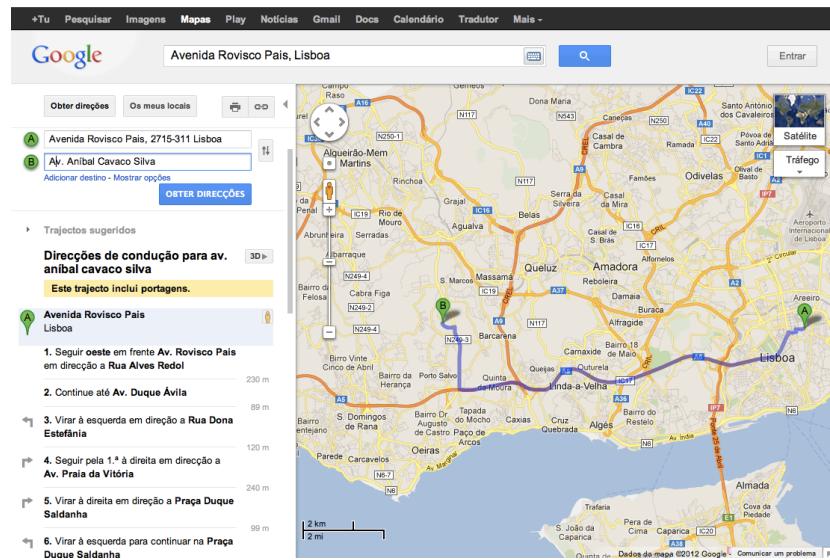


Figura 1.3: Instruções para ir do Campus da Alameda ao do Tagus Parque.

sabe.

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

- Um algoritmo é rigoroso.** Cada instrução do algoritmo deve especificar exata 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 rebuscados 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 (objeto de vidro) dentro de uma panela (ou sobre o lume, interpretando a frase à letra) juntamente com o açúcar.

Para evitar a ambiguidade inerente à linguagem utilizada pelos seres hu-

manos (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 reações 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 aspeto, 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  $23^\circ$ , 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  $23^\circ$ , 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 objetivo 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 objetos a manipular (500 g de açúcar, 5 gemas de ovos) e uma descrição das ações a efetuar sobre esses objetos (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 objetivo 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 efetua 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 efetuar sobre as entidades devem ser compreendidas pelo agente que executa o algoritmo. Essas ações 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 ações como “ações elementares”, isto não significa que elas sejam operações atómicas (isto é, indecomponíveis). Elas podem referir-se a um conjunto de ações 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 diretamente as ações do computador. As instruções em linguagem máquina são constituídas por uma sequência de dois símbolos discretos, correspondendo à existência ou à ausência de sinal (normalmente representados por 1 e por 0, respetivamente) e manipulam diretamente 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 abstração (Figura 1.4): ao nível mais baixo existem os circuitos eletrónicos, o “hardware”, os quais são responsáveis por executar com grande velocidade as ordens dadas ao computador; o “hardware” pode ser diretamente comandado através da linguagem

14 CAPÍTULO 1. COMPUTADORES, ALGORITMOS E PROGRAMAS

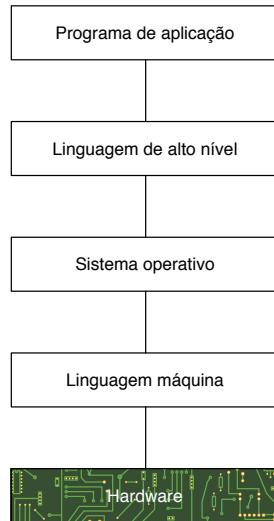


Figura 1.4: Alguns níveis de abstração existentes num computador.

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 interação 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 (recordar-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 ações 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, começamos por analisar o que fazemos para resolver este problema utilizando uma calculadora. O objetivo da nossa apresentação é fornecer uma ideia intuitiva dos passos e do raciocínio envolvidos na criação de um programa e, simultaneamente, mostrar um primeiro programa em Python.

Podemos descrever as ações 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 ações indicam que existe um padrão que se repete ao longo da nossa atuação 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 um 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 deveremos encontrar uma formulação alternativa. Para isso, vamos refletir 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 que executamos sempre que adicionamos um número à soma corrente são:

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

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

Estes passos são executados repetitivamente para todos os números a somar, o que significa que a nossa tarefa corresponde a repetir estas operações *enquanto* o número a somar não exceder 100. Em programação, a repetição de uma sequência de ações chama-se um *ciclo*. 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 efetuamos 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 aspetos 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 ( $=$ ,  $:=$ , ou outro, dependendo da linguagem de programação utilizada). Se adotarmos 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)
```

A última linha do programa anterior tem por finalidade informar o mundo exterior do valor calculado pelo programa e corresponde a uma instrução de saída de dados.

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 Python<sup>5</sup>:

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

---

<sup>5</sup>Em Python, “\*” representa a multiplicação e “//” representa a divisão inteira.

Um aspeto 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. Esses 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 chamados, respetivamente, a *sintaxe* e a *semântica* da linguagem. A sintaxe determina qual a constituição das frases que podem ser fornecidas ao computador e a semântica determina o que o computador vai fazer ao seguir as indicações apresentadas em 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 de gramáticas.

Como a sintaxe apenas se preocupa com o processo de combinação dos símbolos da 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.

O processo de descrição formal da sintaxe de uma linguagem consiste na apresentação de uma gramática para essa linguagem. Uma *gramática* formal é composta por:

1. Um conjunto de símbolos, os *símbolos não terminais*, que não aparecem explicitamente nas frases da linguagem mas que são utilizados para descrever os vários componentes das frases. Um símbolo não terminal está sempre associado a um conjunto de entidades da linguagem;

2. Um símbolo não terminal especial, o *símbolo inicial*, que representa o elemento principal da linguagem;
3. Um conjunto de símbolos, os *símbolos terminais*, que aparecem nas frases da linguagem;
4. Um conjunto de regras, as *regras de produção*, que descrevem a estrutura dos vários componentes da linguagem. Estas regras de produção definem todas as frases da linguagem a partir do símbolo inicial.

Para descrever a sintaxe do Python, escrevemos gramáticas utilizando uma notação conhecida por notação BNF<sup>6</sup>, a qual utiliza as seguintes regras:

1. Os *símbolos não terminais* escrevem-se entre parênteses angulares, “⟨” e “⟩”. Usando um exemplo do Python, ⟨expressão⟩ é um símbolo não terminal que corresponde a uma expressão em Python (cuja definição é apresentada na página 29). Este símbolo não terminal não representa nenhuma expressão em particular, mas sim um componente genérico da linguagem;
2. Os *símbolos terminais* escrevem-se sem qualquer símbolo especial à sua volta. Por exemplo, em Python, + corresponde a um símbolo terminal;
3. As regras de produção escrevem-se, usando as seguintes convenções:
  - (a) O símbolo “::=” (lido “é definido como”) serve para definir componentes da linguagem. Cada regra de produção define o componente que aparece à esquerda do símbolo “::=”, como sendo a descrição que aparece à direita desse mesmo símbolo;
  - (b) O símbolo “|” (lido “ou”) representa possíveis alternativas;
  - (c) A utilização do caráter “+” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido uma ou mais vezes;
  - (d) A utilização do caráter “\*” imediatamente após um símbolo não terminal significa que esse símbolo pode ser repetido zero ou mais vezes;

---

<sup>6</sup>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 and Ullman, 1969] e [Ginsburg, 1966], atribuem ao termo “BNF” o significado “Backus Normal Form”.

- (e) 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, respetivamente 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 exemplo, consideremos uma gramática 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:

$$\begin{aligned}\langle \text{número binário} \rangle &::= \langle \text{dígito binário} \rangle \mid \\ &\quad \langle \text{dígito binário} \rangle \langle \text{número binário} \rangle \\ \langle \text{dígito binário} \rangle &::= 0 \mid 1\end{aligned}$$

Nesta gramática os símbolos terminais são 0 e 1 e os símbolos não terminais são  $\langle \text{número binário} \rangle$  (o símbolo inicial) 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  $\langle \text{número binário} \rangle$ <sup>7</sup>. 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 quisermos (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,

---

<sup>7</sup>É 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.

representado pelo símbolo não terminal dígito binário, como sendo ou 0 ou 1.

Em alternativa, poderíamos também ter apresentado a seguinte gramática para a definição de números binários:

$$\langle \text{número binário} \rangle ::= \langle \text{dígito binário} \rangle^+$$

$$\langle \text{dígito binário} \rangle ::= 0 \mid 1$$

A notação utilizada para definir formalmente uma linguagem, no caso da notação 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 ações 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. Utilizamos 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 sintática e erros de natureza semântica.

Os erros de natureza sintática, ou *erros sintáticos* 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 deteção 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 sintática têm que ser corrigidos antes da execução das instruções, ou seja, o computador não executará nenhuma instrução sintaticamente incorreta. Os programadores novatos passam grande parte do seu tempo a corrigir erros sintáticos, mas à medida que se tornam mais experientes, o número de erros sintáticos que originam é cada vez menor e a sua origem é detetada 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 detetar do que os erros de caráter sintático. Estes erros resultam do facto de o programador não ter expressado corretamente, através da linguagem de programação, as ações 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 deteção e correção dos erros de natureza semântica de um programa.

Ao processo de deteção e correção, tanto dos erros sintáticos 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 sintáticos como semânticos, chamam-se “*bugs*”<sup>8</sup>. 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 insetos (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 América<sup>9</sup>.

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 detetive 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.

## 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, eletrónico 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

---

<sup>8</sup>Do inglês, insetos.

<sup>9</sup>Ver [Taylor, 1984], página 44.

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 and Landweber, 1974] e [Hopcroft and 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, adotamos 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<sup>10</sup>.

## 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

---

<sup>10</sup>Do inglês, “context-free languages”

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, cujo símbolo inicial é  $\langle S \rangle$ :

$\langle S \rangle ::= \langle A \rangle a$

$\langle A \rangle ::= a \langle B \rangle$

$\langle B \rangle ::= \langle A \rangle a \mid b$

- (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, cujo símbolo inicial é  $\langle idt \rangle$ :

$\langle idt \rangle ::= \langle letras \rangle \langle numeros \rangle$

$\langle letras \rangle ::= \langle letra \rangle \mid$

$\quad \langle letra \rangle \langle letras \rangle$

$\langle numeros \rangle ::= \langle num \rangle \mid$

$\quad \langle num \rangle \langle numeros \rangle$

$\langle letra \rangle ::= A \mid B \mid C \mid D$

$\langle num \rangle ::= 1 \mid 2 \mid 3 \mid 4$

- (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 **cddddr**.
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. Dada a seguinte gramática em notação BNF, cujo símbolo inicial é  $\langle S \rangle$ :

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

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

$\langle C \rangle ::= a$

(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 operações de entrada de dados, operações de saída de dados, operações matemáticas, execução condicional e repetição. Estas operações são normalmente aplicadas a variáveis.

Neste capítulo apresentamos alguns dos tipos de valores que podem ser associados a variáveis, o modo de os combinar. Apresentamos também algumas noções básicas associadas a um programa, nomeadamente, algumas das operações correspondentes às operações elementares. No final deste capítulo, estaremos em condições de escrever alguns programas muito simples.

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 interativamente num ambiente em que exista um interpretador do Python – uma “caixa eletrónica” que compreenda as frases da linguagem Python.

A interação 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 efetuar as ações indicadas na frase. Após efetuar as ações 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 interação dá-se o nome de processamento interativo. Em *processamento interativo*, 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 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
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 *caráter de pronto*<sup>1</sup>. A utilização interativa do Python corresponde à repetição de um ciclo em que o Python lê uma frase, efetua as ações indicadas na frase e escreve o resultado dessas ações. Este ciclo é chamado *ciclo lê-avalia-escreve*<sup>2</sup>. Uma sessão em Python corresponde a um ciclo, tão longo quanto o utilizador desejar, de leitura de frases, execução das ações indicadas na frase e apresentação dos resultados.

Ao longo do livro apresentamos exemplos de interações com o Python. Ao apresentar uma interação, 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.

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

---

<sup>1</sup>Do inglês, “prompt character”.

<sup>2</sup>Do inglês, “read-eval-print loop”.

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  $\langle \text{definição} \rangle$  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 a aplicação de uma 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, consideramos expressões correspondentes a constantes e a 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 interação mostra a resposta do Python quando lhe fornecemos algu-

mas constantes:

Da interação 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 carateres*<sup>3</sup>. As quais correspondem a sequências de carateres. As constantes das cadeias de carateres são representadas em Python delimitadas por plicas ou por aspas. Neste livro delimitamos as cadeias de carateres por aspas. O *conteúdo* da cadeia de carateres corresponde a todos os carateres da cadeia, com a exceção das plicas; o *comprimento* da cadeia é o número de carateres do seu conteúdo. Por exemplo `'bom dia'` é uma cadeia de carateres com 7 carateres, `b`, `o`, `m`, <sup>4</sup>, `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*<sup>5</sup>, 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

---

<sup>3</sup>Uma cadeia de carateres é frequentemente designada pelo seu nome em inglês, “string”.

<sup>4</sup>Este caráter corresponde ao espaço em branco.

<sup>5</sup>Do inglês, “built-in” operations.

modo<sup>6</sup>:

```
<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 interação (a qual utiliza operadores cujo significado é óbvio, excetuando 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 efetuadas. 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*

---

<sup>6</sup>Iremos ver outros modos de definir expressões compostas.

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

Tabela 2.1: Prioridade dos operadores em Python.

*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. Na Tabela 2.1 apresentamos a lista de prioridades dos operadores em Python (estas prioridades são, de modo geral, adotadas 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.

## 2.2 Tipos elementares de dados

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 dados*<sup>7</sup> 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 dados disponíveis variam de linguagem de programação para lin-

---

<sup>7</sup>Em inglês “data type”.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$e_1 + e_2$	Inteiros	O resultado de somar $e_1$ com $e_2$ .
$e_1 - e_2$	Inteiros	O resultado de subtrair $e_2$ a $e_1$ .
$-e$	Inteiro	O simétrico de $e$ .
$e_1 * e_2$	Inteiros	O resultado de multiplicar $e_1$ por $e_2$ .
$e_1 // e_2$	Inteiros	O resultado da 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.

guagem de programação. De um modo geral, podemos dizer que os tipos de dados 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.

### 2.2.1 O tipo inteiro

Os números inteiros, em Python designados por `int`<sup>8</sup>, são números sem parte decimal, podendo ser positivos, negativos ou zero<sup>9</sup>. Sobre expressões de tipo inteiro podemos realizar as operações apresentadas na Tabela 2.2. Por exemplo,

```
>>> -12
-12
>>> 032
Syntax Error: 032: <string>
>>> 7 // 2
3
```

---

<sup>8</sup>Do inglês, “integer”.

<sup>9</sup>Em Python um inteiro não pode começar por zero como o mostra a segunda linha da interação.

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

### 2.2.2 O tipo real

Os números reais, em Python designados por `float`<sup>10</sup>, 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.

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, `4.2e5` (`=420000.0`), `-6e-8` (`=-0.00000006`). A notação científica é utilizada principalmente para representar números muito grandes ou muito pequenos.

A seguinte interação mostra algumas constantes reais em Python:

```
>>> 7.7
```

```
7.7
```

---

<sup>10</sup>Designação que está associada à representação de números reais dentro de um computador, a representação em *virgula flutuante* (em inglês, “floating point representation”).

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$e_1 + e_2$	Reais	O resultado de somar $e_1$ com $e_2$ .
$e_1 - e_2$	Reais	O resultado de subtrair $e_2$ a $e_1$ .
$-e$	Real	O simétrico de $e$ .
$e_1 * e_2$	Reais	O resultado de multiplicar $e_1$ por $e_2$ .
$e_1 / e_2$	Reais	O resultado de dividir $e_1$ por $e_2$ .
$\text{abs}(e)$	Real	O valor absoluto de $e$ .

Tabela 2.3: Operações sobre números reais.

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

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

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  $+_{\mathbb{Z}}$  que adiciona números inteiros produzindo um número inteiro e a operação  $+_{\mathbb{R}}$  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*<sup>11</sup>.

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  $+_{\mathbb{Z}}$ , se ambos forem reais, aplica a operação  $+_{\mathbb{R}}$ , se um for inteiro e o outro real, converte o número inteiro para o real correspondente e aplica a operação  $+_{\mathbb{R}}$ . Esta conversão tem o nome de *coerção*<sup>12</sup>, sendo demonstrada na seguinte interação:

```
>>> 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 interação mostra a utilização destas operações:

```
>>> round(3.3)
3
>>> round(3.6)
```

---

<sup>11</sup>Do inglês, “overloaded”.

<sup>12</sup>Do inglês, “coercion”.

<i>Operação</i>	<i>Tipo do argumento</i>	<i>Tipo do valor</i>	<i>Operação</i>
<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.

```

4
>>> int(3.9)
3
>>> float(3)
3.0

```

### 2.2.3 O tipo lógico

O tipo lógico, em Python designado por `bool`<sup>13</sup>, apenas pode assumir dois valores, `True` (*verdadeiro*) e `False` (*falso*).

As operações que se podem efetuar 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).

---

<sup>13</sup>Em honra ao matemático inglês George Boole (1815–1864).

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

Tabela 2.5: Operações de conjunção e disjunção.

## 2.3 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 aspetos importantes em programação corresponde à possibilidade de usar nomes para designar entidades computacionais. A utilização de nomes corresponde a um nível de abstração no qual deixamos de nos preocupar com a indicação direta 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:

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

A primeira alternativa desta definição corresponde à *atribuição simples* e a segunda alternativa à *atribuição múltipla*.

Antes de apresentar a semântica da instrução de atribuição e exemplos da sua utilização, teremos de especificar o que é um nome. Em Python, um *nome* é definido formalmente através das seguintes expressões em notação BNF:

```
 $\langle \text{nome} \rangle ::= \langle \text{nome simples} \rangle |$ 
 $\quad \langle \text{nome indexado} \rangle |$ 
 $\quad \langle \text{nome composto} \rangle$ 
```

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 3.5.

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*.

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

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

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 caráter `_`. 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 caráter, “ ”, que não é permitido) e `igual?` (tem um caráter, “?”, 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.

Comecemos por discutir a primeira alternativa da instrução de atribuição, a qual corresponde à primeira linha da expressão BNF que define  $\langle \text{instrução de atribuição} \rangle$ :

```
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.

atribuição), a que chamamos *atribuição simples*.

Ao encontrar uma instrução da forma  $\langle\text{nome}\rangle = \langle\text{expressão}\rangle$ , 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*<sup>14</sup>) 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 respetivo 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 interação com o Python:

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

Na primeira linha surge uma instrução de atribuição. Ao executar esta ins-

---

<sup>14</sup>Do inglês, “namespace”.



Figura 2.1: Representação de um ambiente.

trução, 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 interação 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 2.1 mostramos a representação do ambiente correspondente a esta interação. Um ambiente é representado por um retângulo 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.

Consideremos agora a seguinte interação com o Python, efetuada depois da interação 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 da interação anterior, 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 2.2. A instrução de atribuição `nota = nota + 1` tem o efeito de atribuir à variável `nota` o valor anterior de `nota` mais `1`. Este último exemplo mostra o caráter 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 interação



Figura 2.2: Ambiente resultante da execução de `nota = nota + 1`.

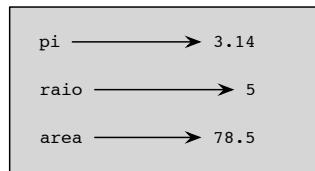


Figura 2.3: Ambiente depois da definição de `pi`, `raio` e `area`.

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 interação 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 2.3.

```

>>> 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 interação 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 2.4).

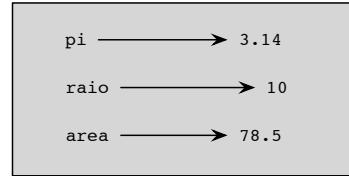


Figura 2.4: Ambiente depois da alteração do valor de `raio`.

Consideremos agora a segunda alternativa 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>` apresentada na página 39. 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 interação:

```

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

Consideremos a seguinte interação

```

>>> 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

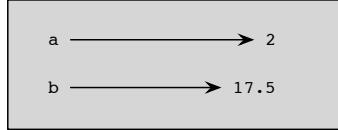


Figura 2.5: Ambiente.

Figura 2.6: Ambiente depois da execução de `a, b = b, a`.

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 42.

Consideremos agora a seguinte interação:

```

>>> a = 2
>>> b = 17.5
>>> a
2
>>> b
17.5
>>> a, b = b, a
>>> 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, respetivamente 17.5 e 2 (Figura 2.5). Em seguida, o valor 17.5 é atribuído à variável `a` e o valor 2 é atribuído à variável `b` (Figura 2.6). Ou seja a instrução `a, b = b, a` tem o efeito de trocar os valores das variáveis `a` e `b`.

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
$e_1 == e_2$	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.

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 ações 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 ações, por exemplo a atribuição de um nome a uma variável. Por esta razão, quando nos referimos às ações efetuadas pelo Python associadas a uma instrução dizemos que a *instrução é executada*.

## 2.4 Predicados e condições

Uma operação que produz resultados do tipo lógico chama-se um *predicado*. Por exemplo, as operações `not`, `and` e `or` apresentadas nas Secção 2.2.3 correspondem a predicados.

Uma expressão cujo valor é do tipo lógico chama-se uma *condição*<sup>15</sup>. 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.

<sup>15</sup>Na 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`.

A seguinte interação 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
```

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 direta 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*<sup>16</sup>.

---

<sup>16</sup>Do inglês, “syntactic sugar”.

## 2.5 Comunicação com o exterior

### 2.5.1 Leitura de dados

Durante a execução de um programa, é vulgarmente necessário obter valores do exterior para efetuar 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 9, consideraremos operações de leitura de dados localizados em ficheiros.

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.

Para exemplificar a utilização da função `input`, consideremos as seguintes interações:

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

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'`.

2. `>>> input()`

<i>Caráter escape</i>	<i>Significado</i>
\\"	Barra ao contrário (\)
\'	Plica (')
\"	Aspas ("")
\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 carateres de escape em Python.

```
estou a escrever sem caráter de pronto
'estou a escrever sem caráter 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 caráter de pronto” seguido de “Return”, a função `input` devolve a cadeia de carateres `'estou a escrever sem caráter de pronto'`, e daí a aparente duplicação das duas últimas linhas na interação anterior.

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

Esta interação introduz algo de novo. Na cadeia de carateres que é fornecida à função `input` aparece um carácter que não vimos até agora, a sequência `\n`. A isto chama-se um carácter de escape<sup>17</sup>. Um *caráter 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 carateres de escape existentes em Python.

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

---

<sup>17</sup>Do inglês, “escape character”.

```
>>> a
'teste'
```

Esta interação mostra a atribuição do valor lido a uma variável, `a`.

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:

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

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

```
>>> 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 efetuar 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 9, 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 ecrã. 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 interação 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 interação anterior atribuirmos à variável `c` o valor devolvido por `print` constatamos o seguinte<sup>18</sup>:

```
>>> 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 objetivo não é a produção de um valor, mas sim a produção

---

<sup>18</sup>No capítulo 13 voltamos a considerar a constante `None`.

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 interação com o Python correspondeu a fornecer-lhe comandos (através de expressões e de uma instrução), 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.

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*<sup>19</sup>, é definido do seguinte modo<sup>20</sup>:

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

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

O conceito de `⟨definição⟩` é apresentado no capítulo 3, 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

---

<sup>19</sup>Do inglês, “script”.

<sup>20</sup>Esta não é uma definição completa de um programa em Python, mas corresponde à definição de programa usada neste livro.

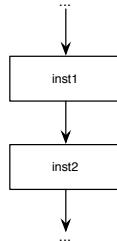


Figura 2.7: Fluxograma para a sequenciação.

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 é considerada neste livro), fornecem estruturas de controle mais sofisticadas, nomeadamente a seleção 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. Este é também o princípio utilizado quando fornecemos diretamente ao interpretador do Python uma sequência de instruções. Sendo  $\langle\text{inst1}\rangle$  e  $\langle\text{inst2}\rangle$  símbolos não 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. Na Figura 2.7 apresentamos esquematicamente o conceito de sequenciação. Esta representação diagramática tem o nome de *fluxograma*<sup>21</sup>.

<sup>21</sup>Os fluxogramas (em inglês “flowchart”) foram inventados em 1921 por Frank Bunker Gilbreth (1868–1924) como um modo de representar processos, tendo sido muito utilizados na descrição de algoritmos. A partir da década de 1970 a sua utilização para a descrição de algoritmos diminuiu, com base no argumento que a sua utilização dava origem a código mal estruturado. Algumas técnicas recentes, como o UML, podem ser consideradas como extensões de fluxogramas.

Um fluxograma corresponde a uma representação gráfica de um determinado processo recorrendo a figuras geométricas normalizadas e a setas ligando essas figuras geométricas. Através desta representação gráfica é possível compreender de forma rápida e fácil a transição entre as entidades que participam no processo em causa. Relativamente a um programa, as instruções representam-se dentro de um retângulo. A Figura 2.7 indica que a instrução `inst1` é executada e, após a sua execução, a instrução `inst2` é executada.

O conceito de sequência de instruções é definido através da seguinte expressão em notação BNF:

$$\langle \text{instruções} \rangle ::= \langle \text{instrução} \rangle \boxed{\text{CR}} \mid \\ \langle \text{instrução} \rangle \boxed{\text{CR}} \langle \text{instruções} \rangle$$

Nesta definição, `\boxed{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{CR}` corresponde ao fim de uma linha,

Consideremos um programa para calcular a área de uma coroa circular, uma região limitada por dois círculos concêntricos. Se representarmos por  $r_e$  o raio da circunferência externa e por  $r_i$  o raio da circunferência interna, a área da coroa circular,  $A$ , é dada pela diferença entre a área do círculo externo e a área do círculo interno:

$$A = \pi * (r_e^2 - r_i^2)$$

Consideremos agora o seguinte programa (sequência de instruções):

```
pi = 3.1416

r_e = eval(input('Qual o valor do raio exterior?\n? '))
r_i = eval(input('Qual o valor do raio interior?\n? '))

area = pi * (r_e * r_e - r_i * r_i)

print('Valor da area:', area)
```

Neste programa inserimos linhas em branco para separar partes do programa, a inicialização de variáveis, a leitura de dados, o cálculo da área 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

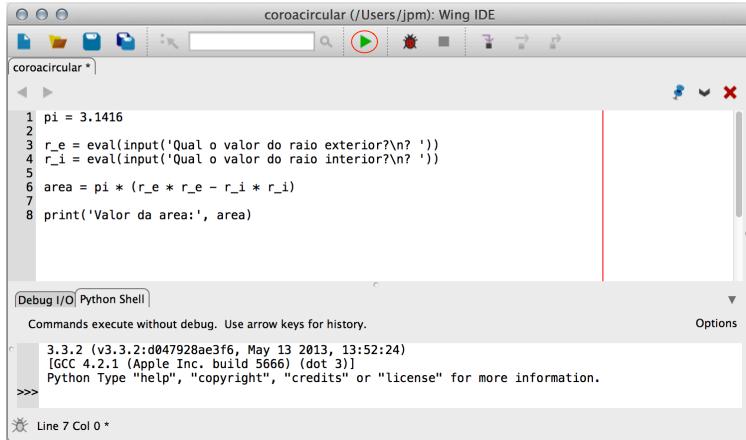


Figura 2.8: Janela do Wing101 antes de executar o programa.

a sua execução. As linhas em branco podem ser consideradas como a *instrução vazia*, a qual é definida pela seguinte sintaxe:

$\langle \text{instrução vazia} \rangle ::=$

Quanto à sua semântica, ao encontrar uma instrução vazia, o Python não toma nenhuma ação.

Para instruir o Python para executar este programa, devemos escrevê-lo na zona superior da janela de interação com o Wing101<sup>22</sup> e carregar no ícone correspondente a um triângulo a verde (indicado na Figura 2.8 dentro de um círculo). Isto origina a execução das instruções do nosso programa como se mostra na Figura 2.9.

## 2.7 Seleção

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.

<sup>22</sup>O Wing101 (obtido a partir de <http://wingware.com/downloads/wingide-101>) é o ambiente de desenvolvimento que usamos com o Python.

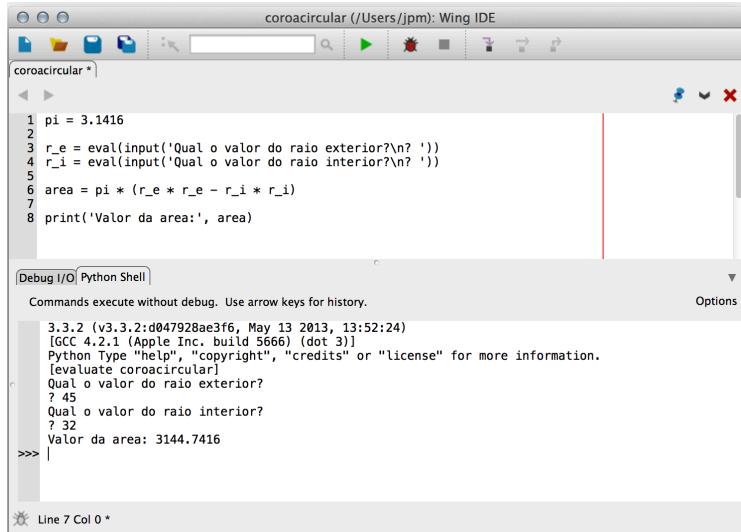


Figura 2.9: Janela do Wing101 depois de executar o programa.

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`<sup>23</sup> permite a seleção entre duas ou mais alternativas. Dependendo do valor de uma condição, esta instrução permite-nos selecionar uma de duas ou mais instruções para serem executadas. A sintaxe da instrução `if` é definida pelas seguintes expressões em notação BNF<sup>24</sup>:

```

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

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

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

```

<sup>23</sup>A palavra “if” traduz-se, em português, por “se”.

<sup>24</sup>Existem outras alternativas para a instrução `if` que não consideramos neste livro.

$\langle \text{instrução composta} \rangle ::= [\text{TAB+}] \langle \text{instruções} \rangle [\text{TAB-}]$

$\langle \text{condição} \rangle ::= \langle \text{expressão} \rangle$

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

Na definição sintática da instrução `if`, a  $\langle \text{condição} \rangle$  representa qualquer expressão do tipo lógico. Este último aspeto, 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 uma instrução `if`, mas

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

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

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

1. Ao encontrar uma instrução da forma

```
if <condição>:  
    <instruções>
```

o Python começa por avaliar a expressão  $\langle \text{condição} \rangle$ . Se o seu valor for `True`, o Python executa as instruções correspondentes a  $\langle \text{instruções} \rangle$ ; se o valor da expressão  $\langle \text{condição} \rangle$  for `False`, o Python não faz mais nada relativamente a esta instrução `if`. Esta semântica é apresentada na Figura 2.10 recorrendo a um fluxograma. Num fluxograma, um losango corresponde a um ponto de decisão: consoante o valor da condição que se

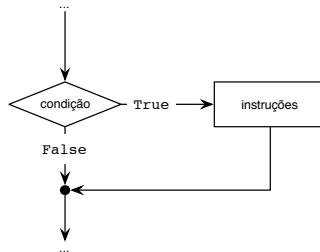


Figura 2.10: Fluxograma para uma forma da instrução `if`.

encontra representada dentro do losangulo, podem ser seguidos um de dois possíveis caminhos. Num fluxograma, um losangulo apresenta um ponto de entrada, representado por uma seta dirigida a um dos seus vértices, e dois pontos de saída, representados por setas saindo dos seus vértices. Estas setas apresentam um dos rótulos `true` ou `false` correspondentes ao caminho tomado de acordo com o valor da condição. Num fluxograma, um círculo negro representa a junção de dois caminhos.

Os efeitos da semântica desta forma da instrução `if` são mostrados com o seguinte programa:

```

n = eval(input('Escreva um número\n? '))
if n % 2 == 0:
    print('O número é par')
print('Adeus')
  
```

Suponhamos que durante a execução deste programa fornecíamos o número 4:

```

Escreva um número
? 4
O número é par
Adeus
  
```

dado que `n % 2 == 0`, o programa executa a instrução `print('O número é par')`, continuando para a instrução após o `if`. No entanto se o número fornecido for 3, obtemos a interação:

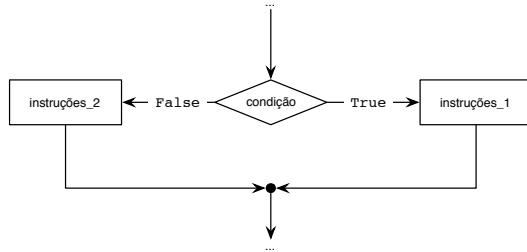


Figura 2.11: Fluxograma para outra forma da instrução if.

```
Escreva um número
```

```
? 3
```

```
Adeus
```

pois o Python não executa a instrução `print('O número é par')` visto que `n % 2 == 0` tem o valor `False`.

2. Ao encontrar uma instrução da forma<sup>25</sup>

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

o Python começa por avaliar a expressão `<cond>`. Se o seu valor for `True`, as instruções correspondentes a `<instruções1>` são executadas e as instruções correspondentes a `<instruções2>` não o são; se o seu valor for `False`, as instruções correspondentes a `<instruções2>` são executadas e as instruções correspondentes a `<instruções1>` não o são. Esta semântica é mostrada na Figura 2.11 recorrendo a um fluxograma.

Consideremos o seguinte programa (este programa está representado através de um fluxograma na Figura 2.12, o qual mostra a utilização de sequenciação e de seleção):

```
n = eval(input('Escreva um número\n? '))
if n % 2 == 0:
    print('O número é par')
```

---

<sup>25</sup>A qual não usa `<outras alternativas>`, utilizando apenas a `<alternativa final>`.

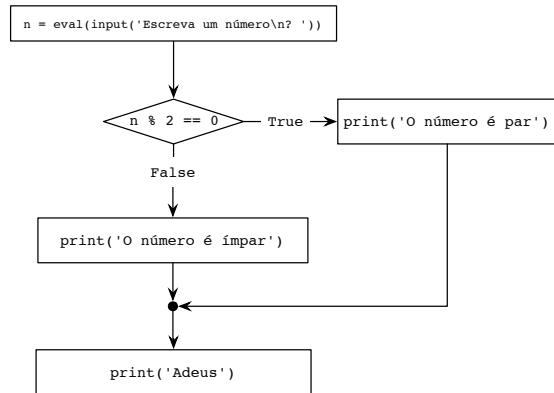


Figura 2.12: Fluxograma para determinar se um número é par ou ímpar.

```

else:
    print('O número é ímpar')
print('Adeus')

```

Neste caso, se fornecermos ao programa um número par, obtemos um comportamento semelhante ao do programa anterior, mas se fornecermos um número ímpar obtemos a interação:

```

Escreva um número
? 7
O número é ímpar
Adeus

```

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

```

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

```

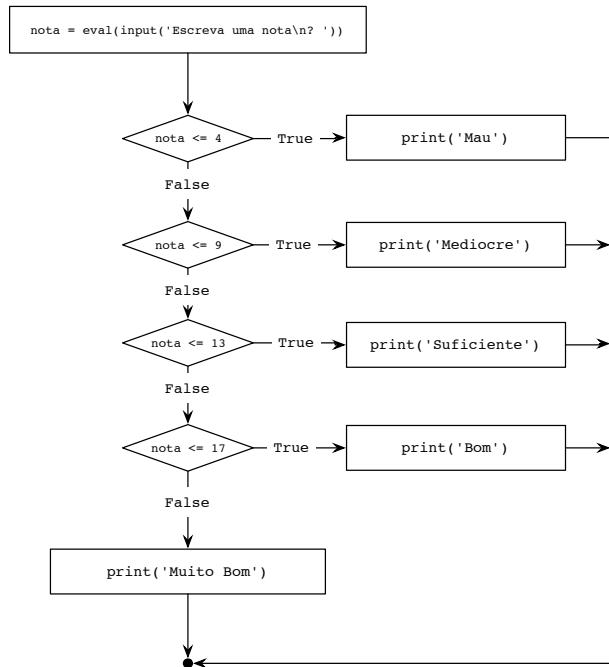


Figura 2.13: Fluxograma para a conversão de notas.

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 o seguinte programa que utiliza uma 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 ga-

mas de notas não seja verificado, esta instrução `if` realiza adequadamente o seu trabalho para qualquer valor de `nota`. Na Figura 2.10 apresentamos o fluxograma correspondente a este programa.

```
nota = eval(input('Escreva uma nota\n? '))
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 8: “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 efetuou 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*<sup>26</sup>. 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 condição tiver o valor verdadeiro. A sintaxe da instrução `while` é definida pela seguinte expressão em notação BNF<sup>27</sup>:

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

Na definição sintática 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 <condição>: [CR] <instruções>`, o símbolo não terminal `<instruções>` corresponde ao corpo do ciclo e o símbolo `<condição>`, 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 <condição>: [CR] <instruções>`, o Python calcula o valor de `<condição>`. Se o seu valor for `True`, o Python efetua uma passagem pelo ciclo, executando as instruções correspondentes a `<instruções>`. Em seguida, volta a calcular o valor de `<condição>` e o processo repete-se enquanto o valor de `<condição>` for `True`. Quando o valor de `<condição>` for `False`, a execução do ciclo termina. Na Figura 2.14 apresentamos um fluxograma correspondente à instrução `while`.

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 BNF<sup>28</sup>:

```
<instrução break> ::= break
```

<sup>26</sup>O conceito de ciclo infinito é teórico, pois, na prática o ciclo acabará por terminar ou porque nós o interrompemos (fartamo-nos de esperar) ou porque os recursos computacionais se esgotam.

<sup>27</sup>A palavra “while” traduz-se em português por “enquanto”.

<sup>28</sup>A palavra “break” traduz-se em português por “fuga”, “quebra”, “pausa”, ou “rutura”.

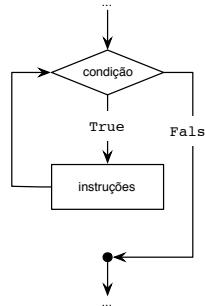


Figura 2.14: Fluxograma correspondente à instrução `while`.

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 zero<sup>29</sup>, após o que efetuam 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.

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, a qual é constituída por duas instruções

---

<sup>29</sup>Esta operação é conhecida por *inicialização da variável*.

`soma = soma + num` (que atualiza 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.

## 2.9 Exemplos

Consideremos 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 ações, 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:

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

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 objetivos de zonas do programa, certos aspectos do algoritmo, etc. Os comentários podem também ser expressões matemáticas provando 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 Python.

Como último exemplo, consideremos o problema de calcular os fatores primos de um número inteiro. Qualquer número inteiro positivo, maior do que um, pode ser escrito univocamente como o produto de vários números primos (chamados *fatores primos*). Ao algoritmo que recebe um número inteiro e calcula os seus fatores primos chama-se *decomposição em fatores primos*.

Para realizar a decomposição de um número, deveremos encontrar números primos que dividem o número a ser decomposto. Realizaremos sucessivas divisões até que o número se torne igual a 1. Como ainda não vimos como calcular números primos, iremos utilizar um outro algoritmo que corresponde a determinar todos os potenciais divisores do número, começando por 2. Se o número for divisível por um determinado divisor, este é um dos fatores primos, dividimo-lo e tentamos novamente com o mesmo divisor; se o número não for divisível pelo divisor, tentamos o divisor seguinte. O processo termina quando o número se tornar igual a 1<sup>30</sup>. Na Tabela 2.9 apresentamos o resultado do nosso algoritmo aplicado ao inteiro 780.

Consideremos agora o programa que corresponde ao algoritmo que acabámos de

---

<sup>30</sup>Como exercício, o leitor deve convencer-se que este algoritmo apenas produz números primos.

<i>Número</i>	<i>Divisor</i>	<i>Divisível?</i>	<i>Escreve</i>
780	2	Sim	2
390	2	Sim	2
195	2	Não	
195	3	Sim	3
65	3	Não	
65	4	Não	
65	5	Sim	5
13	5	Não	
13	6	Não	
13	7	Não	
13	8	Não	
13	9	Não	
13	10	Não	
13	11	Não	
13	12	Não	
13	13	Sim	13
1			

Tabela 2.9: Fatores primos de 780.

descrever. Este programa utiliza duas variáveis, `num` correspondente ao número que queremos decompor em fatores primos e `divisor` que corresponde aos vários divisores que vamos utilizar. O valor de `num` é lido do exterior e começamos com o divisor 2.

Efetuamos depois um ciclo em que vamos verificar se `num` é divisível por `divisor`. Se o for, `divisor` é um dos fatores primos, dividimos o número por `divisor` e continuamos o ciclo; se não for divisível, aumentamos o `divisor` em uma unidade. Este ciclo é repetido até que `num` se torne 1.

O seguinte programa calcula os fatores primos de um número:

```
num = eval(input('Escreva um inteiro: '))
divisor = 2

print('Fatores primos:')
while num != 1:
    if num % divisor == 0: # num é divisível por divisor
        print(divisor)
        num = num // divisor
    else:
        divisor = divisor + 1
```

Com este programa obtemos a interação:

```
Escreva um inteiro: 780
Fatores primos:
2
2
3
5
13
```

## 2.10 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 efetuar certas ações. Ao contrário das expressões, as instruções não têm um valor mas produzem um efeito. Quando o computador efetua as ações 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 seleção 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 efetuarem a entrada e saída de dados.

## 2.11 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 seu programa deve permitir a interação:

```
Escreva um número de segundos  
? 65432998  
O número de dias correspondentes é 757.3263657407407
```

2. Escreva um programa que lê um número inteiro correspondente a um certo número de segundos e que escreve o número de dias, horas, minutos e segundos correspondentes a esse número. Por exemplo,

```
Escreva o número de segundos 345678  
dias: 4 horas: 0 mins: 1 segs: 18
```

3. Escreva um programa em Python que lê três números e que diz qual o maior dos números lidos.
4. Escreva um programa em Python que pede ao utilizador que lhe forneça uma sucessão de inteiros correspondentes a valores em segundos e que calcula o número de dias correspondentes a cada um desses inteiros. O programa termina quando o utilizador fornece um número negativo. O seu programa deve possibilitar a seguinte interação:

```
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  
>>>
```

5. 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  $120^{\circ} F$ .

6. 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  $-1$ . O seu programa deve permitir a interação:

```
Escreva um dígito
(-1 para terminar)
? 3
Escreva um dígito
(-1 para terminar)
? 2
Escreva um dígito
(-1 para terminar)
? 5
Escreva um dígito
(-1 para terminar)
? 7
Escreva um dígito
(-1 para terminar)
? -1
0 número é: 3257
```

7. 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
```

8. 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
```

9. 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 o  $i$ -ésimo termo da série pode ser obtido do termo na posição  $i - 1$ , multiplicando-o por  $x/i$ . O seu programa deve permitir a interação:

```
Qual o valor de x
? 2
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 operações embutidas, 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 aplicação. Por isso, durante o desenvolvimento de um programa, é necessário utilizar operações que não estão previamente definidas na linguagem utilizada. As linguagens de programação fornecem mecanismos para a criação de novas operações e para a sua utilização num programa como se de operações embutidas se tratassem.

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 aspeto está ligado aos conceitos de *função*, *procedimento* e *subprograma*, dependendo da linguagem de programação utilizada. 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 abstração), 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 aspetos 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, a indicação das suas variáveis (ou argumentos) e 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 aspetos 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 modo<sup>1</sup>:

```

⟨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, pode ser definida do seguinte modo:

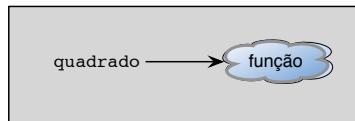
```
def quadrado(x):
    return x * x
```

Nesta definição:

1. O nome da função é `quadrado`;
2. 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;

---

<sup>1</sup>A ⟨definição de função⟩ corresponde em Python a uma ⟨definição⟩ (ver página 28).

Figura 3.1: Ambiente com o nome `quadrado`.

3. A instrução `return x * x` corresponde ao *corpo da função*.

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, adicionando-o ao ambiente, e associa esse nome com o processo de cálculo para os valores da função (o algoritmo correspondente ao corpo da função).

Consideremos a seguinte interação, efetuada imediatamente após o início de uma sessão em Python:

```

Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
Type "help", "copyright", "credits" or "license" for more information.
>>> quadrado
NameError: name 'quadrado' is not defined
>>> def quadrado(x):
...     return x * x
...
>>> quadrado
<function quadrado at 0x10f4618>
  
```

A primeira linha mostra que ao fornecermos ao Python o nome `quadrado` este indica-nos que não o conhece. Nas três linhas seguintes, definimos a função `quadrado`. Nesta interação com o Python, surge um novo *caráter de pronto*, correspondente a “...”. Esta nova forma do caráter 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. 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 a sua definição se encontra armazenada. Na Figura 3.1 mostramos o ambiente gerado pela interação anterior.

## 3.2 Aplicação de funções em Python

Uma vez definida, uma função pode ser usada do mesmo modo que as funções embutidas, ou seja, fornecendo ao Python, numa expressão, o nome da função seguido do número apropriado de argumentos (os *parâmetros concretos*). Depois da definição de uma função, passa a existir em Python uma nova expressão, correspondente a uma aplicação de função (ver a definição apresentada na página 29), a qual é definida do seguinte modo:

```
<aplicação de função> ::= <nome>(<parâmetros concretos>)

<parâmetros concretos> ::= <nada> | <expressões>

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

Nesta definição, *<nome>* corresponde ao nome da função e o número de expressões em *<parâmetros concretos>* é igual ao número de parâmetros formais da função. Este último aspeto 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 interação:

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

A interação 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 resultado da aplicação de uma função, o Python utiliza as seguintes regras:

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 51, 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.

O cálculo do valor 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 efetuadas diretamente após o caráter de pronto. Todos os nomes que são definidos pela operação de atribuição ou através de definições, diretamente executadas após o caráter 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 contêm, 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 interação, efetuada depois da definição da função `quadrado`:

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

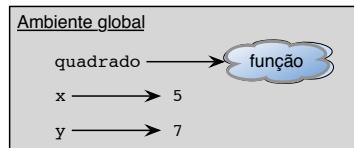


Figura 3.2: Ambiente global com os nomes `quadrado`, `x` e `y`.

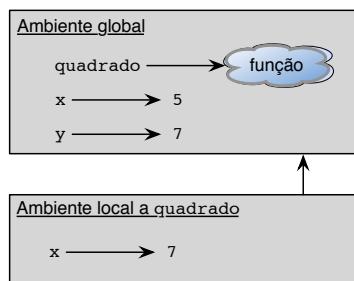


Figura 3.3: Ambiente criado durante a avaliação de `quadrado(y)`.

As duas primeiras linhas dão origem ao ambiente apresentado na Figura 3.2 (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 interação, o Python calcula o valor de `y` e cria um novo ambiente, representado na Figura 3.3 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 3.3 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 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 3.2.

Em resumo, quando uma função é avaliada (ou 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 Abstração procedural

A criação de funções corresponde a uma forma de abstração, a qual consiste em nomear uma sequência de operações que permitem atingir um determinado objetivo (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 *abstração procedural* consiste em dar um nome à sequência de ações que serve para atingir um objetivo (e consequentemente abstrair do modo como as funções realizam as suas tarefas), e em utilizar esse nome, sempre que desejarmos atingir esse objetivo sem termos de considerar explicitamente cada uma das ações individuais que a constituem.

A associação de um nome a uma sequência de ações, 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 *abstração 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 abstração 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 abstração procedural consideraremos funções como caixas pretas – sabemos *o que elas fazem*, mas não nos interessa saber *como o fazem*. Para ilustrar este



Figura 3.4: O conceito de caixa preta.

aspeto, 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):

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

e obtínhamos exatamente 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 3.4). É a isto que corresponde a abstração procedural.

A abstração 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 Tabela de conversão de temperaturas

Tendo em atenção que a conversão de temperatura em graus Farenheit ( $F$ ) para graus centígrados ( $C$ ) é dada através da expressão  $C = \frac{5}{9} \cdot (F - 32)$ , podemos escrever a seguinte função que recebe um inteiro correspondente a uma temperatura em graus Farenheit ( $F$ ) e devolve o valor da temperatura equivalente em graus centígrados:

```
def far_para_cent(F):
    return round(5 * (F - 32) / 9)
```

Usando esta função, podemos escrever o seguinte programa que cria uma tabela de conversão de graus Farenheit para graus centígrados, entre dois valores fornecidos pelo utilizador. Recorde-se da página 52, 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 `far_para_cent`, seguida das instruções que, através de um ciclo, calculam as equivalências entre temperaturas.

```
def far_para_cent(F):
    return round(5 * (F - 32) / 9)

min = eval(input('Qual a temperatura mínima?\n? '))
max = eval(input('Qual a temperatura máxima?\n? '))

while min <= max:
    print(min, 'F =', far_para_cent(min), 'C')
    min = min + 1
```

Com este programa podemos obter a interação:

```
Qual a temperatura mínima?
? 30
Qual a temperatura máxima?
? 40
```

```

30 F = -1 C
31 F = -1 C
32 F = 0 C
33 F = 1 C
34 F = 1 C
35 F = 2 C
36 F = 2 C
37 F = 3 C
38 F = 3 C
39 F = 4 C
40 F = 4 C

```

### 3.4.2 Potênciа

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 interação:

```

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

```

### 3.4.3 Fatorial

Em matemática, o *fatorial*<sup>2</sup> 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 fatorial de um inteiro::

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

Com esta função obtemos a interação:

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

---

<sup>2</sup>Segundo [Biggs, 1979], a função fatorial 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.

### 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 entre dois inteiros, mas, dados dois inteiros, esta definição não nos indica como calcular o seu máximo divisor comum. Este é um dos aspetos 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 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  $n$ ,  $mdc(m, n)$ :

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 a Tabela 2.2) será dado por  $m \% n$  e repetimos o processo com  $m$  igual ao valor de  $n$  (o divisor) e  $n$  igual ao resto da divisão. Este processo é repetido enquanto  $n$  não for zero.

Na Tabela 3.1 apresentamos os passos seguidos no cálculo do máximo divisor comum entre 24 e 16.

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

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

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

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

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

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

### 3.4.5 Raiz quadrada

Suponhamos 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  $x$ ?

Começemos 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.

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  $\sqrt{2}$ .

Iremos utilizar um algoritmo para o cálculo de raízes quadradas, que foi documentado no início da nossa era pelo matemático grego Heron de Alexandria (c. 10–75 d.C.)<sup>3</sup>. Este algoritmo, conhecido por *algoritmo da Babilónia*, 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 do valor aproximado,  $p_i$ , para a raiz quadrada de  $x$ , podemos calcular uma aproximação melhor,  $p_{i+1}$ , através da seguinte fórmula:

$$p_{i+1} = \frac{p_i + \frac{x}{p_i}}{2}. \quad (3.1)$$

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 aspeto que distingue este exemplo dos apresentados nas seções anteriores é o facto do cálculo da raiz quadrada ser realizado por um método aproximado. Isto significa que não vamos obter o valor exato da raiz quadrada<sup>4</sup>, 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_palpito(x, palpito):
```

<sup>3</sup>Embora Heron de Alexandria tenha sido o primeiro a documentar este algoritmo (ver [Kline, 1972]), acredita-se que este já era conhecido muitos séculos antes, nos tempos da Babilónia [Guttag, 2013].

<sup>4</sup>Na realidade, para o nosso exemplo, este é um número irracional.

```

    palpate = novo_palpite(x, palpate)
    return palpate

```

Esta função é bastante simples: fornecendo-lhe um valor para o qual calcular a raiz quadrada (`x`) e um palpite (`palpite`), enquanto não estivermos perante um bom palpite, 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 palpite 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*<sup>5</sup>. Ao desenvolver a solução de um problema utilizando a abordagem do topo para a base, começamos por dividir esse problema noutros 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 3.1, a qual é traduzida pela função:

```

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

```

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

---

<sup>5</sup>Do inglês, “top-down design”.

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 limiar<sup>6</sup>. Utilizando esta medida, diremos que  $p_i$  é uma boa aproximação de  $\sqrt{x}$  se  $| (p_i)^2 - x | < \delta$ , em que  $\delta$  é um valor suficientemente pequeno. Supondo que  $\delta$  corresponde ao nome `delta`, o qual define o limiar de aproximação exigido pela nossa aplicação, podemos escrever a função:

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

Esta função avalia a expressão `abs(x - palpite * palpite) < 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 exatamente o valor da expressão `abs(x - palpite * palpite) < delta`, daí a simplificação que introduzimos na nossa primeira função.

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

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

---

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

Nome	Situação correspondente ao erro
<code>AttributeError</code>	Referência a um atributo não existente num objeto.
<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 incorreta.
<code>ValueError</code>	Erro gerado quando uma função recebe um argumento de tipo correto 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.

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):
    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:

```
<instrução raise> ::= raise <nome> (<mensagem>)
<mensagem> ::= <cadeia de caracteres>
```

O símbolo não terminal `<nome>` foi definido na página 76. A utilização de `<nome>`

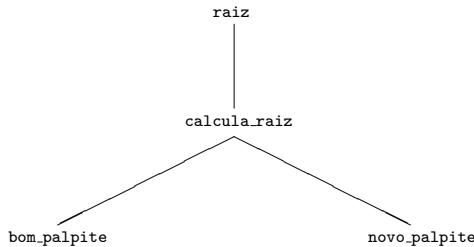


Figura 3.5: Funções associadas ao cálculo da raiz.

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 *{mensagem}* corresponde à mensagem que é mostrada pelo Python com a indicação do erro.

Com a função `raiz`, podemos obter a interação:

```

>>> 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 palpite é suficientemente bom; como calcular um novo palpite; etc. Podemos olhar para a função `raiz` como sendo definida através de um agrupamento de outras funções (Figura 3.5), o que corresponde à abstração 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.

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

Nesta secção 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 retângulo com um de seus ângulos internos igual a  $\alpha$ , 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:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \quad (3.2)$$

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 3.2 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 escrever a seguinte 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 fator  $(-1)^n$ , 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  $n$ .<sup>7</sup> 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.

---

```
def calc_termo(x, n):
```

<sup>7</sup>A função `potencia` foi definida na Secção 3.4.2 e a função `fatorial` na Secção 3.4.3.

```

    return sinal(n) * \
           potencia(x, 2 * n + 1) / fatorial(2 * n + 1)

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

```

Com estas funções podemos gerar a interação:

```
>>> sin(1.57)
0.9999996270418701
```

É importante notar que a função `calc_termo` pode ser escrita de um modo muito mais eficiente. Na realidade, cada termo da série pode ser obtido a partir do termo anterior multiplicando-o por

$$-1 \cdot \frac{x^2}{(2n+1) \cdot 2n}$$

Com base nesta observação, podemos evitar o cálculo repetitivo de fatorial e de potência em cada termo, desde que saibamos qual o valor do termo anterior. Podemos assim escrever uma nova versão das funções `sin` e `calc_termo`:

```

def sin(x):
    n = 0 # ordem do termo da série em consideração
    termo_ant = x # primeiro termo
    seno = termo_ant

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

def calc_termo(x, n, termo_ant):
    return -(termo_ant * x * x / ((2 * n + 1) * 2 * n))

```

### 3.5 Módulos

Na Secção 3.4 desenvolvemos algumas funções matemáticas elementares, potência, fatorial, 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 coleção 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>
<nomes> ::= <nome> |
                           <nome>, <nomes>

```

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

Python	Matemática	Significado
<code>pi</code>	$\pi$	Uma aproximação de $\pi$
<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`.

da seguinte expressão em notação BNF:

`<nome composto> ::= <nome simples> . <nome simples>`

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 interação:

```
>>> 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 interação:

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

```
>>> pi  
3.141592653589793  
>>> 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 interação:

```
>>> 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 *
```

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

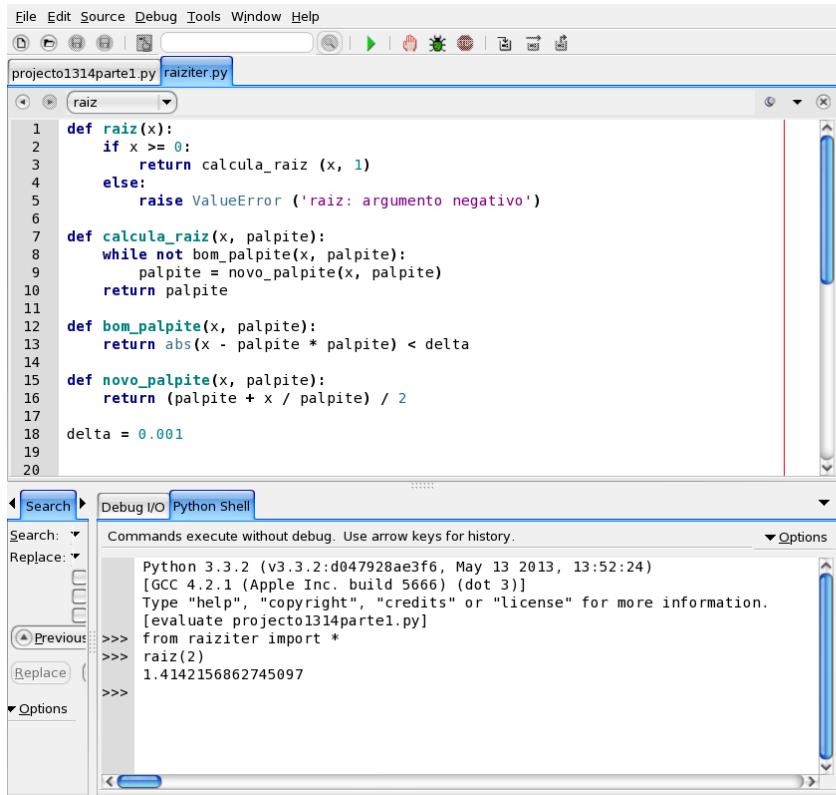


Figura 3.6: Utilização do módulo `raiziter` no Wing 101.

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 3.6 apresentamos a utilização do módulo `raiziter`, o qual contém a definição da função raiz quadrada, tal como apresentada na Secção 3.4.5.

## 3.6 Notas finais

Apresentámos o modo de definir e utilizar novas funções em Python, e o conceito subjacente à sua utilização, a abstração 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 exatos 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 and Greif, 2011].

### 3.7 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. A *congruência de Zeller*<sup>8</sup> é um algoritmo inventado pelo matemático alemão Julius Christian Zeller (1822–1899) para calcular o dia da semana para qualquer dia do calendário. Para o nosso calendário, o *calendário Gregoriano*, a congruência de Zeller é dada por:

$$h = \left( q + \left\lfloor \frac{13(m+1)}{5} \right\rfloor + K + \left\lfloor \frac{K}{4} \right\rfloor + \left\lfloor \frac{J}{4} \right\rfloor - 2J \right) \mod 7$$

em que  $h$  é o dia da semana ( $0$  = Sábado,  $1$  = Domingo, ...),  $q$  é o dia do mês,  $m$  é o mês ( $3$  = março,  $4$  = abril, ...,  $14$  = fevereiro) – os meses de janeiro e fevereiro são contados como os meses  $13$  e  $14$  do ano anterior,  $K$  é o ano do século ( $ano \mod 100$ ),  $J$  é o século ( $\lfloor ano/100 \rfloor$ ). Esta expressão utiliza a função matemática, *chão*, denotada por  $\lfloor x \rfloor$ , a qual converte um número real  $x$  no maior número inteiro menor ou igual a  $x$ . A definição formal desta função é  $\lfloor x \rfloor = \max \{m \in \mathbb{Z} \mid m \leq x\}$ . A

---

<sup>8</sup>[Zeller, 1882].

expressão utiliza também a função módulo, em que  $a \bmod b$  representa o resto da divisão de  $a$  por  $b$ .

Escreva uma função em Python, chamada `dia_da_semana`, que recebe um dia, um mês e um ano e que devolve o dia da semana em que calha essa data. A sua função deve utilizar outras funções auxiliares a definir por si. Por exemplo,

```
>>> dia_da_semana(18, 1, 2014)
'sabado'
```

4. 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.
5. 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.
6. 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
```

7. 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 fatorial.

# 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 dados 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 dados, ou seja, tipos de dados em que os seus elementos estão associados a um agregado de valores. Recorde-se que um tipo de dados corresponde a um conjunto de entidades, os elementos do tipo, juntamente com um conjunto de operações aplicáveis a essas entidades. Os tipos de dados cujos elementos estão associados a um agregado de valores são chamados *tipos estruturados de dados*, *tipos de dados não elementares* ou *estruturas de dados*. Sempre que abordamos um tipo estruturado de dados, temos que considerar o modo como os valores estão agregados e as operações que podemos efetuar 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 vetor. Em matemática, para nos

15	6	10	12	12
----	---	----	----	----

Figura 4.1: Representação gráfica de um tuplo.

referirmos aos elementos de um vetor, utilizamos índices que caracterizam univocamente estes elementos. Por exemplo, se  $\vec{x}$  representa um vetor com três elementos, estes são caracterizados, respetivamente, 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 tuplo<sup>1</sup> é definida sintaticamente pelas seguintes expressões em notação BNF<sup>2, 3</sup>:

```

⟨tuplo⟩ ::= () |
            ⟨elemento⟩ , ⟨elementos⟩
⟨elementos⟩ ::= ⟨nada⟩ |
                ⟨elemento⟩ |
                ⟨elemento⟩ , ⟨elementos⟩
⟨elemento⟩ ::= ⟨expressão⟩ |
                ⟨tuplo⟩ |
                ⟨lista⟩ |
                ⟨dicionário⟩
⟨nada⟩ ::=

```

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 sintática de um tuplo exige que um tuplo com um elemento contenha esse elemento seguido de uma vírgula, pelo que `(1)` *não* corresponde a um tu-

<sup>1</sup> 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.

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

<sup>3</sup> As definições de `⟨lista⟩` e `⟨dicionário⟩` são apresentadas, respetivamente, nos capítulos 5 e 7.

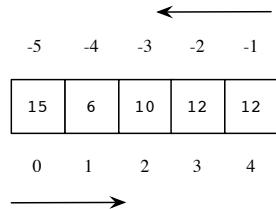


Figura 4.2: Valores dos índices de um tuplo.

pto em Python. O tuplo () não tem elementos e é chamado o *tuplo vazio*. De acordo com as expressões anteriores em notação BNF, (1, 2,) e (1, 2, 3,) também são tuplos, respetivamente com 2 e 3 elementos. O tuplo esquematicamente 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 inteiro<sup>4</sup>) 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 interação:

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

<sup>4</sup>Este 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 interação 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 interação 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 retos) 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 interação:

```
>>> 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 sobrearcadas, 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	<b>True</b> se o elemento $e$ pertence ao tuplo $t$ ; <b>False</b> 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 interação 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 interação 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 interação:

```
>>> 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 interação 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, exceto 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 interação:

```
>>> 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 cujos elementos são números 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
```

Embora a função anterior esteja correta, cada vez que o seu ciclo `while` é executado, o Python vai calcular o valor de `len(t)`, o que é desnecessário dado que este valor não muda. Podemos pensar em escrever uma versão mais eficiente desta função do seguinte modo:

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

Os tuplos podem ser utilizados para representar vetores. O vetor  $\vec{v} = c_1\vec{e}_x + c_2\vec{e}_y + c_3\vec{e}_z$  pode ser representado pelo tuplo  $(c_1, c_2, c_3)$ . A seguinte função em Python recebe dois vetores como argumentos (com um número arbitrário de dimensões) e devolve o vetor correspondente à sua soma:

```
def soma_vetores(t1, t2):

    if len(t1) != len(t2):
        raise ValueError ('Não é possível somar')
    else:
        res = ()
        i = 0
        num_els = len(t1)
        while i < num_els:
            res = res + (t1[i] + t2[i], )
            i = i + 1
        return res
```

Com esta função obtemos a interação:

```
>>> soma_vetores((1, 2, 3), (7, 6, 5))
(8, 8, 8)
```

Consideremos agora o problema, bastante mais complicado do que os problemas anteriores, 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 dados (inteiros, reais ou valores lógicos) do tuplo original. Por exemplo, com esta função, pretendemos obter a interação:

```
>>> 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:

$t$	$i$	$t[:i]$	$t[i]$	$t[i+1:]$
$((1, 2), 3, (4, (5)))$	0	$()$	$(1, 2)$	$(3, (4, 5))$
$(1, 2, 3, (4, 5))$	1			
$(1, 2, 3, (4, 5))$	2			
$(1, 2, 3, (4, 5))$	3	$(1, 2, 3)$	$(4, 5)$	$()$
$(1, 2, 3, 4, 5)$	4			

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 corresponder ao seu segundo argumento ou se pertencer ao tuplo que é o 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 109, obriga-nos a inicializar o índice para o valor zero (`i = 0`) e obriga-nos também a atualizar 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*<sup>5</sup> 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 atualizar 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 BNF<sup>6</sup>:

```
<instrução for> ::= for <nome simples> in <expressão>: [CR]
                           <instrução composta>
```

Na definição sintática da instrução `for`, `<nome simples>` corresponde à variável de controle, `<expressão>` representa uma expressão cujo valor corresponde a uma sequência (novamente, este aspeto 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 existam

---

<sup>5</sup>Em inglês, “counted loop”.

<sup>6</sup>A palavra “for” traduz-se em português por “para”.

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 63. 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 interação:

```
>>> 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 109, 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 interação:

```
>>> 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 sequência de elementos correspondente a 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 à sequência de elementos representada pelo 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 a sequência de elementos representada pelo 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 à sequência de elementos representada pelo tuplo  $(5, 6, 7, 8, 9)$  e o valor de `range(-3, 3)` corresponde à sequência de elementos representada pelo 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  $e_2 - 1$  (ou nunca sendo inferiores a  $e_2 + 1$ , no caso de  $e_3 < 0$ ), 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 à sequência de elementos representada pelo tuplo  $(2, 5, 8, 11, 14, 17)$  e o valor de `range(20, 2, -3)` corresponde à sequência de elementos representada pelo tuplo  $(20, 17, 14, 11, 8, 5)$ .

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 inspecionar 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 fazer<sup>7</sup>. 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`.

---

<sup>7</sup>Como exercício, deve exprimir o ciclo `for` em termos do 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 111. Na realidade, nesta função estamos a processar uma variável, cujo tuplo associado é alterado durante o 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 carateres revisitadas

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

Em Python, as cadeias de carateres correspondem a um tipo estruturado de dados, designado por `str`<sup>8</sup>, o qual corresponde a uma sequência de carateres individuais.

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

```
<cadeia de carateres> ::= '<caráter>*' |  
                      "<caráter>*" |  
                      """<caráter>*"""
```

A definição anterior indica que uma cadeia de carateres é uma sequência de zero ou mais carateres delimitados por plicas, por aspas ou por três aspas, devendo os símbolos que delimitam a cadeia de carateres ser iguais (por exemplo "abc" não é uma cadeia de carateres). Como condição adicional, não apresentada na definição em notação BNF, os carateres de uma cadeia de carateres delimitadas por plicas não podem conter a plica<sup>9</sup> e os carateres de uma cadeia de carateres delimitadas por aspas não podem conter aspas<sup>10</sup>. As cadeias de carateres ' ' e " " são chamadas *cadeias de carateres vazias*. Ao longo do livro utilizamos as plicas para delimitar as cadeias de carateres.

---

<sup>8</sup>Da sua designação em inglês, “string”.

<sup>9</sup>Exceto se recorrermos ao carácter de escape apresentado na Tabela 2.8.

<sup>10</sup>*Ibid.*

As cadeias de carateres delimitadas por três aspas, chamadas *cadeias de caracteres de documentação*<sup>11</sup>, são usadas para documentar definições. Quando o Python encontra uma cadeia de carateres 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 carateres à entidade 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 carateres de documentação que lhe está associada.

Por exemplo, suponhamos que em relação à função `soma_elementos` apresentada na página 113, associávamos a seguinte cadeia de carateres 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 interação:

```
>>> 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 carateres são entidades *imutáveis*, no sentido de que não podemos alterar os seus elementos.

---

<sup>11</sup>Do inglês, “docstring”.

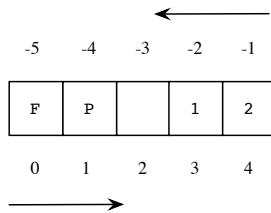


Figura 4.3: Valores dos índices de uma cadeia de caracteres.

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, respetivamente a 'F' e '2'. Cada um destes elementos é uma cadeia de caracteres com apenas um elemento. É importante notar que '2' não é o mesmo que o inteiro 2, é o carácter "2", como o mostra a seguinte interação:

```
>>> 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 efetuar as operações indicadas na Tabela 4.3<sup>12</sup> (note-se que estas operações são semelhantes às apresentadas na Tabela 4.1 e, consequentemente, todas estas operações são sobrecurregadas). A seguinte interação mostra a utilização de algumas destas operações:

```
>>> cumprimento = 'bom dia!'
>>> cumprimento[0:3]
'bom'
>>> 'ola ' + cumprimento
```

---

<sup>12</sup>Nesta tabela, "Universal", designa qualquer tipo.

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

Tabela 4.3: Algumas operações embutidas sobre cadeias de carateres.

```
'ola bom dia!'
>>> cumprimento * 3
'bom dia!bom dia!bom dia!'
>>> 'z' in cumprimento
False
>>> 'ab' in 'abcd'
True
>>> len(cumprimento)
8
>>> str((1, 2))
'(1, 2)'
>>> str(2)
'2'
```

Como primeiro exemplo, iremos abordar um problema que é relevante em situações em que se lida com números muito grandes, por exemplo números de cartões de crédito ou números de identificação de produtos. Nestes números, é comum a introdução de um dígito adicional, chamado *dígito de controle* ou *algarismo de controle*, que permite detetar os erros mais comuns na introdução

ou transmissão de grandes números: (1) a alteração de um único algarismo (por exemplo, escrever 6578 em lugar de 7578) e (2) a troca de pares de algarismos subjacentes (por exemplo, escrever 5778 em lugar de 7578). O dígito de controle é calculado de acordo com determinado algoritmo e permite verificar se o número introduzido está ou não correto. A ideia de utilizar sistemas de identificação com dígitos de controle está vulgarizada num número quase infundável de aplicações,<sup>13</sup> por exemplo, números de cartões de crédito, números de cheques, códigos de barras de produtos, identificadores de livros (código ISBN).

No nosso exemplo utilizamos o código ISBN. O ISBN (uma abreviatura de “International Standard Book Number”) é um código numérico que identifica univocamente um livro baseado no “Standard Book Number” (SBN), um código de 9 dígitos criado em 1966 por Gordon Foster, Professor de Estatística no Trinity College, em Dublin. A configuração do ISBN foi definida em 1967 por David Whitaker e Emery Koltay. Os códigos ISBN usados até ao final de 2006 (conhecidos por ISBN-10) são constituídos por 10 dígitos. Por exemplo, o código ISBN do primeiro livro que escrevi, *Introduction to Computer Science using Pascal* [Martins, 1989], é 0-534-09402-3 (os traços entre os dígitos são meramente convencionais e devem ser ignorados). Num ISBN, o dígito da direita é o dígito de controle que é calculado do seguinte modo: se  $x_i$  representar o dígito na  $i$ -ésima posição a contar da esquerda, o dígito de controle  $x_{10}$  é escolhido de modo que a soma

$$\sum_{i=1}^{10} i \times x_i$$

dê resto zero quando dividida por 11. No nosso exemplo teremos

$$1 \times 0 + 2 \times 5 + 3 \times 3 + 4 \times 4 + 5 \times 0 + 6 \times 9 + 7 \times 4 + 8 \times 0 + 9 \times 2 + 10 \times 3 = 165$$

e  $165 \bmod 11 = 0$ . Como o resto da divisão por 11 é um número entre 0 e 10, convencionou-se que se utiliza a letra “X” para representar 10. Assim, 0-8218-0863-X é um código ISBN correto.

Consideremos agora um programa para validar um ISBN com 10 dígitos. Este programa recebe uma cadeia de caracteres correspondente ao ISBN e valida esse ISBN. Usamos uma cadeia de caracteres e não um inteiro porque o ISBN pode conter a letra “X”. No entanto, mesmo que o ISBN apenas contivesse algarismos, também teríamos que usar uma cadeia de caracteres porque em

---

<sup>13</sup>O artigo [Buescu, 2001] apresenta uma descrição fácil de ler e interessante deste problema.

Python um inteiro não pode começar por zero. A verificação do ISBN é feita através da função `verifica_ISBN`, a qual utiliza a função auxiliar `controle` que calcula o dígito de controle.

```
def verifica_ISBN(n):
    if len(n) != 10:
        return 'ISBN incorreto'
    else:
        soma = 0
        # calcula a soma sem o dígito de controle
        for i in range(len(n)-1):
            soma = (i+1) * eval(n[i]) + soma
        # o dígito de controle é somado
        soma = soma + 10 * controle(n)
        if soma % 11 == 0:
            return 'ISBN correto'
        else:
            return 'ISBN incorreto'

def controle(n):
    if n[9] == 'X':
        return 10
    else:
        return eval(n[9])
```

Como exemplo de utilização das operações existentes sobre cadeias de carateres, a seguinte função recebe duas cadeias de carateres e devolve os carateres 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 carateres vazia
    for s in s1:
        if s in s2:
            s_comum = s_comum + s
    return s_comum
```

Com este programa, obtemos a interação:

```
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 interação 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 interação:

```
>>> 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”)<sup>14</sup>, 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 atualmente obsoletos) e 95 caracteres visíveis, os 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,

---

<sup>14</sup>O código ASCII foi publicado pela primeira vez em 1963, tendo sofrido revisões ao longo dos anos, a última das quais em 1986.

caráter	código	caráter	código	caráter	código	caráter	código
!	32	8	56	P	80	h	104
"	33	9	57	Q	81	i	105
#	34	:	58	R	82	j	106
\$	35	;	59	S	83	k	107
%	36	<	60	T	84	l	108
&	37	=	61	U	85	m	109
,	38	>	62	V	86	n	110
(	39	@	64	W	87	o	111
)	40	A	65	X	88	p	112
*	41	B	66	Y	89	q	113
+	42	C	67	Z	90	r	114
,	43	D	68	[	91	s	115
-	44	E	69	\	92	t	116
.	45	F	70	]	93	u	117
/	46	G	71	^	94	v	118
0	47	H	72	'	96	w	119
1	48	I	73	a	97	x	120
2	49	J	74	b	98	y	121
3	50	K	75	c	99	z	122
4	51	L	76	d	100	{	123
5	52	M	77	e	101		124
6	53	N	78	f	102	}	125
7	54	O	79	g	103	~	126
	55						

Tabela 4.4: Carateres ASCII visíveis.

por exemplo carateres acentuados, foi desenvolvida uma representação, conhecida por *Unicode*,<sup>15</sup> a qual permite a representação dos carateres de quase todas as linguagens escritas. Por exemplo, o caráter acentuado “ã” 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 carateres, o Python fornece duas funções embutidas, `ord` que recebe um caráter (sob a forma de uma cadeia de carateres 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 caráter (sob a forma de uma cadeia de carateres com apenas um elemento) representado por esse número. Por exemplo,

<sup>15</sup>As 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.

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

O código usado na representação de carateres introduz uma ordem total entre os carateres. O Python permite utilizar os operadores relacionais apresentados na Tabela 2.7 para comparar quer carateres quer cadeias de carateres. Com esta utilização, o operador “<” lê-se “aparece antes” e o operador “>” lê-se “aparece depois”. Assim, podemos gerar a seguinte interação:

```
>>> 'a' < 'b'
True
>>> 'a' > 'A'
True
>>> 'abc' > 'adg'
False
>>> 'abc' < 'abcd'
True
```

Utilizando a representação de carateres, vamos escrever um programa que recebe uma mensagem (uma cadeia de carateres) 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<sup>16</sup>. Usando 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:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

---

<sup>16</sup>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 execução. O livro [Sing, 1999] apresenta a história dos vários métodos para cifrar mensagens.

```
FGHIJKLMNOPQRSTUVWXYZABCDE
```

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 `transforma` faz a codificação de um carácter usando a nossa cifra de substituição. Esta função recebe carácter, `c`, e o deslocamento, `desloc`. Para fazer a codificação do carácter, se esse carácter corresponder a uma letra maiúscula, calcula o seu código ASCII, dado por `ord(c)`, calcula a distância a que esse carácter se encontra em relação ao início do alfabeto, `ord(c) - ord('A')`, soma `desloc` a essa distância e determina qual a distância a que a letra resultante está do início do alfabeto, `(ord(c) - ord('A') + desloc) % 26`, e soma essa distância à posição da letra “A”, `ord('A') + (ord(c) - ord('A') + desloc) % 26`, finalmente, usando a função `chr`, calcula qual a letra que substitui o carácter.

```
def transforma(c, desloc):
    if c == ' ':
        return c
    elif 'A' <= c <= 'Z':
        return chr(ord('A') + (ord(c) - ord('A') + desloc) % 26)
    else:
        return '?'
```

Note-se que para descodificar um carácter podemos utilizar a função `transforma` com um deslocamento negativo.

A função `cod_descod` recebe uma frase, `f`, o deslocamento a usar na cifra de substituição, `n`, e o tipo de transformação a efectuar, `tipo` (o qual pode ser ‘c’ para codificar ou ou ‘d’ para descodificar), e devolve a correspondente mensagem codificada ou descodificada. Para guardar a mensagem resultante, a função usa a variável `res` que é inicializada com a cadeia de carateres vazia.

```
def cod_descod(f, n, tipo):
    res = ''
    if tipo == 'c':
```

```

        desloc = n
else:
    desloc = - n

for c in f:
    res = res + transforma(c, desloc)
return res

```

Podemos agora escrever um programa para a codificação e descodificação de mensagens. Este programa, correspondente à função sem argumentos `codificador`, utiliza a função `cod_descod` 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 efetua a operação solicitada e mostra o resultado obtido. O programa usa um deslocamento de 5 na cifra de substituição.

```

def codificador():

    original = input('Introduza uma mensagem\n-> ')
    tipo = input('C para codificar ou D para descodificar\n-> ')

    if tipo in ('C', 'c'):
        print('A mensagem codificada é:\n',
              cod_descod(original, 5, 'c'))
    elif tipo in ('D', 'd'):
        print('A mensagem descodificada é:\n',
              cod_descod(original, 5, 'd'))
    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 ações.

```

>>> codificador()
Introduza uma mensagem
-> O PYTHON E DIVERTIDO

```

```
C para codificar ou D para descodificar  
-> c  
A mensagem codificada é:  
T UDYMTS J INAJWYNIT  
>>> codificador()  
Introduza uma mensagem  
-> T UDYMTS J INAJWYNIT  
C para codificar ou D para descodificar  
-> D  
A mensagem descodificada é:  
O PYTHON E DIVERTIDO
```

## 4.4 Notas finais

Este foi o primeiro capítulo em que abordámos tipos estruturados de dados. Considerámos dois tipos, os tuplos e as cadeias de carateres, 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 em Python a função `explode` que recebe um número inteiro, verificando a correção do seu argumento, e devolve o tuplo contendo os dígitos desse número, pela ordem em que aparecem no número. Por exemplo

```
>>> explode(34500)  
(3, 4, 5, 0, 0)
```

```
>>> explode(3.5)
ValueError: explode: argumento não inteiro
```

3. Escreva em Python a função `implode` que recebe um tuplo contendo algarismos, verificando a correção do seu argumento, e devolve o número inteiro contendo os algarismos do tuplo, pela ordem em que aparecem. Por exemplo

```
>>> implode((3, 4, 0, 0, 4))
34004
>>> implode((2, 'a', 5))
ValueError: implode: elemento não inteiro
```

Escreva duas versões da sua função, uma utilizando um ciclo `while` e outra utilizando um ciclo `for`.

4. Escreva em Python a função `filtra_pares` que recebe um tuplo contendo algarismos, verificando a correção do seu argumento, e devolve o tuplo contendo apenas os algarismos pares. Por exemplo

```
>>> filtra_pares((2, 5, 6, 7, 9, 1, 8, 8))
(2, 6, 8, 8)
```

5. Recorrendo às funções `explode`, `implode` e `filtra_pares`, escreva em Python a função `algarismos_pares` que recebe um inteiro e devolve o inteiro que apenas contém os algarismos pares do número original. Por exemplo,

```
algarismos_pares(6643399766641)
6646664
```

6. 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
```

7. 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
```

8. 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
```

9. 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)
```

10. 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 dados que existem no tuplo original. Por exemplo,

```
>>> soma_els_atomicos((3, (((((6, (7, )), ), ), ), ), 2, 1))  
19  
>>> soma_els_atomicos((((), ), ))  
0
```

11. 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)
```

12. 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
```

13. Escreva em Python duas funções, `cc_para_int` e `int_para_cc`, que convertem, respetivamente, uma cadeia de carateres num inteiro e um inteiro numa cadeia de carateres. Por exemplo

```
>>> cc_para_int('bom dia')
98111109032100105097
>>> int_para_cc(97098)
'ab'
>>> int_para_cc(cc_para_int('bom dia'))
'bom dia'
```

14. A partir do dia 1 de janeiro de 2007, o ISBN foi alterado passando a ser constituído por 13 dígitos. O seu dígito de controle (o dígito mais à direita) é tal que se  $x_i$  representar o dígito na  $i$ -ésima posição a contar da esquerda, o dígito de controle  $x_{13}$  é escolhido de modo que a soma

$$\sum_{i=1}^{13} p(i) \times x_i$$

dê resto zero quando dividida por 10. Nesta expressão

$$p(i) = \begin{cases} 1 & \text{se } i \text{ é par} \\ 3 & \text{em caso contrário} \end{cases}$$

Escreva um programa para validar um ISBN com 13 dígitos.

# 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 dados 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 dados, ou seja, tipos de dados em que os seus elementos estão associados a um agregado de valores. Recorde-se que um tipo de dados corresponde a um conjunto de entidades, os elementos do tipo, juntamente com um conjunto de operações aplicáveis a essas entidades. Os tipos de dados cujos elementos estão associados a um agregado de valores são chamados *tipos estruturados de dados*, *tipos de dados não elementares* ou *estruturas de dados*. Sempre que abordamos um tipo estruturado de dados, temos que considerar o modo como os valores estão agregados e as operações que podemos efetuar 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 vetor. Em matemática, para nos

15	6	10	12	12
----	---	----	----	----

Figura 4.1: Representação gráfica de um tuplo.

referirmos aos elementos de um vetor, utilizamos índices que caracterizam univocamente estes elementos. Por exemplo, se  $\vec{x}$  representa um vetor 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 tuplo<sup>1</sup> é definida sintaticamente pelas seguintes expressões em notação BNF<sup>2, 3</sup>:

```

⟨tuplo⟩ ::= () |
            ⟨elemento⟩ , ⟨elementos⟩
⟨elementos⟩ ::= ⟨nada⟩ |
                ⟨elemento⟩ |
                ⟨elemento⟩ , ⟨elementos⟩
⟨elemento⟩ ::= ⟨expressão⟩ |
                ⟨tuplo⟩ |
                ⟨lista⟩ |
                ⟨dicionário⟩
⟨nada⟩ ::=
```

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 sintática de um tuplo exige que um tuplo com um elemento contenha esse elemento seguido de uma vírgula, pelo que `(1)` *não* corresponde a um tu-

<sup>1</sup> 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.

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

<sup>3</sup> As definições de `⟨lista⟩` e `⟨dicionário⟩` são apresentadas, respectivamente, nos capítulos 5 e 7.

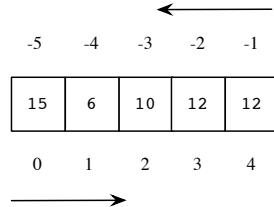


Figura 4.2: Valores dos índices de um tuplo.

pto em Python. O tuplo () não tem elementos e é chamado o *tuplo vazio*. De acordo com as expressões anteriores em notação BNF, (1, 2,) e (1, 2, 3,) também são tuplos, respetivamente com 2 e 3 elementos. O tuplo esquematicamente 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 inteiro<sup>4</sup>) 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 interação:

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

<sup>4</sup>Este 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 interação 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 interação 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 retos) 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 interação:

```
>>> 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 sobreescritas, 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	<b>True</b> se o elemento $e$ pertence ao tuplo $t$ ; <b>False</b> 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 interação 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 interação 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 interação:

```
>>> 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 interação 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, exceto 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 interação:

```
>>> 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 cujos elementos são números 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
```

Embora a função anterior esteja correta, cada vez que o seu ciclo `while` é executado, o Python vai calcular o valor de `len(t)`, o que é desnecessário dado que este valor não muda. Podemos pensar em escrever uma versão mais eficiente desta função do seguinte modo:

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

Os tuplos podem ser utilizados para representar vetores. O vetor  $\vec{v} = c_1\vec{e}_x + c_2\vec{e}_y + c_3\vec{e}_z$  pode ser representado pelo tuplo  $(c_1, c_2, c_3)$ . A seguinte função em Python recebe dois vetores como argumentos (com um número arbitrário de dimensões) e devolve o vetor correspondente à sua soma:

```
def soma_vetores(t1, t2):

    if len(t1) != len(t2):
        raise ValueError ('Não é possível somar')
    else:
        res = ()
        i = 0
        num_els = len(t1)
        while i < num_els:
            res = res + (t1[i] + t2[i], )
            i = i + 1
    return res
```

Com esta função obtemos a interação:

```
>>> soma_vetores((1, 2, 3), (7, 6, 5))
(8, 8, 8)
```

Consideremos agora o problema, bastante mais complicado do que os problemas anteriores, 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 dados (inteiros, reais ou valores lógicos) do tuplo original. Por exemplo, com esta função, pretendemos obter a interação:

```
>>> 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>	0	<code>()</code>	<code>(1, 2)</code>	<code>(3, (4, 5))</code>
<code>(1, 2, 3, (4, 5))</code>	1			
<code>(1, 2, 3, (4, 5))</code>	2			
<code>(1, 2, 3, (4, 5))</code>	3	<code>(1, 2, 3)</code>	<code>(4, 5)</code>	<code>()</code>
<code>(1, 2, 3, 4, 5)</code>	4			

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 corresponder ao seu segundo argumento ou se pertencer ao tuplo que é o 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 109, obriga-nos a inicializar o índice para o valor zero (`i = 0`) e obriga-nos também a atualizar 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*<sup>5</sup> 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 atualizar 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 BNF<sup>6</sup>:

```
<instrução for> ::= for <nome simples> in <expressão>: [CR]
                           <instrução composta>
```

Na definição sintática da instrução `for`, `<nome simples>` corresponde à variável de controle, `<expressão>` representa uma expressão cujo valor corresponde a uma sequência (novamente, este aspeto 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 existam

---

<sup>5</sup>Em inglês, “counted loop”.

<sup>6</sup>A palavra “for” traduz-se em português por “para”.

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 63. 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 interação:

```
>>> 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 109, 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 interação:

```
>>> 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 sequência de elementos correspondente a 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 à sequência de elementos representada pelo 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 a sequência de elementos representada pelo 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 à sequência de elementos representada pelo tuplo  $(5, 6, 7, 8, 9)$  e o valor de `range(-3, 3)` corresponde à sequência de elementos representada pelo 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  $e_2 - 1$  (ou nunca sendo inferiores a  $e_2 + 1$ , no caso de  $e_3 < 0$ ), 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 à sequência de elementos representada pelo tuplo  $(2, 5, 8, 11, 14, 17)$  e o valor de `range(20, 2, -3)` corresponde à sequência de elementos representada pelo tuplo  $(20, 17, 14, 11, 8, 5)$ .

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 inspecionar 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 fazer<sup>7</sup>. 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`.

---

<sup>7</sup>Como exercício, deve exprimir o ciclo `for` em termos do 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 111. Na realidade, nesta função estamos a processar uma variável, cujo tuplo associado é alterado durante o 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 carateres revisitadas

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

Em Python, as cadeias de carateres correspondem a um tipo estruturado de dados, designado por `str`<sup>8</sup>, o qual corresponde a uma sequência de carateres individuais.

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

```
<cadeia de carateres> ::= '<caráter>*' |
                           "<caráter>*" |
                           """<caráter>*"""
```

A definição anterior indica que uma cadeia de carateres é uma sequência de zero ou mais carateres delimitados por plicas, por aspas ou por três aspas, devendo os símbolos que delimitam a cadeia de carateres ser iguais (por exemplo "abc" não é uma cadeia de carateres). Como condição adicional, não apresentada na definição em notação BNF, os carateres de uma cadeia de carateres delimitadas por plicas não podem conter a plica<sup>9</sup> e os carateres de uma cadeia de carateres delimitadas por aspas não podem conter aspas<sup>10</sup>. As cadeias de carateres ' ' e "" são chamadas *cadeias de carateres vazias*. Ao longo do livro utilizamos as plicas para delimitar as cadeias de carateres.

<sup>8</sup>Da sua designação em inglês, “string”.

<sup>9</sup>Exceto se recorrermos ao carácter de escape apresentado na Tabela 2.8.

<sup>10</sup>*Ibid.*

As cadeias de carateres delimitadas por três aspas, chamadas *cadeias de caracteres de documentação*<sup>11</sup>, são usadas para documentar definições. Quando o Python encontra uma cadeia de carateres 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 carateres à entidade 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 carateres de documentação que lhe está associada.

Por exemplo, suponhamos que em relação à função `soma_elementos` apresentada na página 113, associávamos a seguinte cadeia de carateres 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 interação:

```
>>> 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 carateres são entidades *imutáveis*, no sentido de que não podemos alterar os seus elementos.

---

<sup>11</sup>Do inglês, “docstring”.

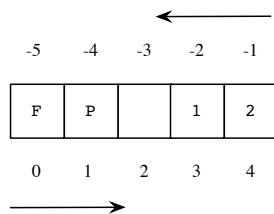


Figura 4.3: Valores dos índices de uma cadeia de caracteres.

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, respetivamente a 'F' e '2'. Cada um destes elementos é uma cadeia de caracteres com apenas um elemento. É importante notar que '2' não é o mesmo que o inteiro 2, é o carácter "2", como o mostra a seguinte interação:

```
>>> 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 efetuar as operações indicadas na Tabela 4.3<sup>12</sup> (note-se que estas operações são semelhantes às apresentadas na Tabela 4.1 e, consequentemente, todas estas operações são sobrecurregadas). A seguinte interação mostra a utilização de algumas destas operações:

```
>>> cumprimento = 'bom dia!'
>>> cumprimento[0:3]
'bom'
>>> 'ola ' + cumprimento
```

<sup>12</sup>Nesta tabela, "Universal", designa qualquer tipo.

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

Tabela 4.3: Algumas operações embutidas sobre cadeias de carateres.

```
'ola bom dia!'
>>> cumprimento * 3
'bom dia!bom dia!bom dia!'
>>> 'z' in cumprimento
False
>>> 'ab' in 'abcd'
True
>>> len(cumprimento)
8
>>> str((1, 2))
'(1, 2)'
>>> str(2)
'2'
```

Como primeiro exemplo, iremos abordar um problema que é relevante em situações em que se lida com números muito grandes, por exemplo números de cartões de crédito ou números de identificação de produtos. Nestes números, é comum a introdução de um dígito adicional, chamado *dígito de controle* ou *algarismo de controle*, que permite detetar os erros mais comuns na introdução

ou transmissão de grandes números: (1) a alteração de um único algarismo (por exemplo, escrever 6578 em lugar de 7578) e (2) a troca de pares de algarismos subjacentes (por exemplo, escrever 5778 em lugar de 7578). O dígito de controle é calculado de acordo com determinado algoritmo e permite verificar se o número introduzido está ou não correto. A ideia de utilizar sistemas de identificação com dígitos de controle está vulgarizada num número quase infindável de aplicações,<sup>13</sup> por exemplo, números de cartões de crédito, números de cheques, códigos de barras de produtos, identificadores de livros (código ISBN).

No nosso exemplo utilizamos o código ISBN. O ISBN (uma abreviatura de “International Standard Book Number”) é um código numérico que identifica univocamente um livro baseado no “Standard Book Number” (SBN), um código de 9 dígitos criado em 1966 por Gordon Foster, Professor de Estatística no Trinity College, em Dublin. A configuração do ISBN foi definida em 1967 por David Whitaker e Emery Koltay. Os códigos ISBN usados até ao final de 2006 (conhecidos por ISBN-10) são constituídos por 10 dígitos. Por exemplo, o código ISBN do primeiro livro que escrevi, *Introduction to Computer Science using Pascal* [Martins, 1989], é 0-534-09402-3 (os traços entre os dígitos são meramente convencionais e devem ser ignorados). Num ISBN, o dígito da direita é o dígito de controle que é calculado do seguinte modo: se  $x_i$  representar o dígito na  $i$ -ésima posição a contar da esquerda, o dígito de controle  $x_{10}$  é escolhido de modo que a soma

$$\sum_{i=1}^{10} i \times x_i$$

dê resto zero quando dividida por 11. No nosso exemplo teremos

$$1 \times 0 + 2 \times 5 + 3 \times 3 + 4 \times 4 + 5 \times 0 + 6 \times 9 + 7 \times 4 + 8 \times 0 + 9 \times 2 + 10 \times 3 = 165$$

e  $165 \bmod 11 = 0$ . Como o resto da divisão por 11 é um número entre 0 e 10, convencionou-se que se utiliza a letra “X” para representar 10. Assim, 0-8218-0863-X é um código ISBN correto.

Consideremos agora um programa para validar um ISBN com 10 dígitos. Este programa recebe uma cadeia de caracteres correspondente ao ISBN e valida esse ISBN. Usamos uma cadeia de caracteres e não um inteiro porque o ISBN pode conter a letra “X”. No entanto, mesmo que o ISBN apenas contivesse algarismos, também teríamos que usar uma cadeia de caracteres porque em

---

<sup>13</sup>O artigo [Buescu, 2001] apresenta uma descrição fácil de ler e interessante deste problema.

Python um inteiro não pode começar por zero. A verificação do ISBN é feita através da função `verifica_ISBN`, a qual utiliza a função auxiliar `controle` que calcula o dígito de controle.

```
def verifica_ISBN(n):
    if len(n) != 10:
        return 'ISBN incorreto'
    else:
        soma = 0
        # calcula a soma sem o dígito de controle
        for i in range(len(n)-1):
            soma = (i+1) * eval(n[i]) + soma
        # o dígito de controle é somado
        soma = soma + 10 * controle(n)
        if soma % 11 == 0:
            return 'ISBN correto'
        else:
            return 'ISBN incorreto'

def controle(n):
    if n[9] == 'X':
        return 10
    else:
        return eval(n[9])
```

Como exemplo de utilização das operações existentes sobre cadeias de carateres, a seguinte função recebe duas cadeias de carateres e devolve os carateres 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 carateres vazia
    for s in s1:
        if s in s2:
            s_comum = s_comum + s
    return s_comum
```

Com este programa, obtemos a interação:

```
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 interação 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 interação:

```
>>> 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”)<sup>14</sup>, 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 atualmente obsoletos) e 95 caracteres visíveis, os 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,

---

<sup>14</sup>O código ASCII foi publicado pela primeira vez em 1963, tendo sofrido revisões ao longo dos anos, a última das quais em 1986.

caráter	código	caráter	código	caráter	código	caráter	código
!	32	8	56	P	80	h	104
"	33	9	57	Q	81	i	105
#	34	:	58	R	82	j	106
\$	35	;	59	S	83	k	107
%	36	<	60	T	84	l	108
&	37	=	61	U	85	m	109
,	38	>	62	V	86	n	110
(	39	@	64	W	87	o	111
)	40	A	65	X	88	p	112
*	41	B	66	Y	89	q	113
+	42	C	67	Z	90	r	114
,	43	D	68	[	91	s	115
-	44	E	69	\	92	t	116
.	45	F	70	]	93	u	117
/	46	G	71	^	94	v	118
0	47	H	72	'	96	w	119
1	48	I	73	a	97	x	120
2	49	J	74	b	98	y	121
3	50	K	75	c	99	z	122
4	51	L	76	d	100	{	123
5	52	M	77	e	101		124
6	53	N	78	f	102	}	125
7	54	O	79	g	103	~	126
	55						

Tabela 4.4: Carateres ASCII visíveis.

por exemplo carateres acentuados, foi desenvolvida uma representação, conhecida por *Unicode*,<sup>15</sup> a qual permite a representação dos carateres de quase todas as linguagens escritas. Por exemplo, o caráter acentuado “â” 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 carateres, o Python fornece duas funções embutidas, `ord` que recebe um caráter (sob a forma de uma cadeia de carateres 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 caráter (sob a forma de uma cadeia de carateres com apenas um elemento) representado por esse número. Por exemplo,

<sup>15</sup>As 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.

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

O código usado na representação de carateres introduz uma ordem total entre os carateres. O Python permite utilizar os operadores relacionais apresentados na Tabela 2.7 para comparar quer carateres quer cadeias de carateres. Com esta utilização, o operador “<” lê-se “aparece antes” e o operador “>” lê-se “aparece depois”. Assim, podemos gerar a seguinte interação:

```
>>> 'a' < 'b'
True
>>> 'a' > 'A'
True
>>> 'abc' > 'adg'
False
>>> 'abc' < 'abcd'
True
```

Utilizando a representação de carateres, vamos escrever um programa que recebe uma mensagem (uma cadeia de carateres) 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<sup>16</sup>. Usando 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:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

---

<sup>16</sup>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 execução. O livro [Sing, 1999] apresenta a história dos vários métodos para cifrar mensagens.

```
FGHIJKLMNOPQRSTUVWXYZABCDE
```

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 `transforma` faz a codificação de um caráter usando a nossa cifra de substituição. Esta função recebe caráter, `c`, e o deslocamento, `desloc`. Para fazer a codificação do caráter, se esse caráter corresponder a uma letra maiúscula, calcula o seu código ASCII, dado por `ord(c)`, calcula a distância a que esse caráter se encontra em relação ao início do alfabeto, `ord(c) - ord('A')`, soma `desloc` a essa distância e determina qual a distância a que a letra resultante está do início do alfabeto, `(ord(c) - ord('A') + desloc) % 26`, e soma essa distância à posição da letra “A”, `ord('A') + (ord(c) - ord('A') + desloc) % 26`, finalmente, usando a função `chr`, calcula qual a letra que substitui o caráter.

```
def transforma(c, desloc):
    if c == ' ':
        return c
    elif 'A' <= c <= 'Z':
        return chr(ord('A') + (ord(c) - ord('A') + desloc) % 26)
    else:
        return '?'
```

Note-se que para descodificar um caráter podemos utilizar a função `transforma` com um deslocamento negativo.

A função `cod_descod` recebe uma frase, `f`, o deslocamento a usar na cifra de substituição, `n`, e o tipo de transformação a efectuar, `tipo` (o qual pode ser ‘c’ para codificar ou ou ‘d’ para descodificar), e devolve a correspondente mensagem codificada ou descodificada. Para guardar a mensagem resultante, a função usa a variável `res` que é inicializada com a cadeia de carateres vazia.

```
def cod_descod(f, n, tipo):
    res = ''
    if tipo == 'c':
```

```

    desloc = n
else:
    desloc = - n

for c in f:
    res = res + transforma(c, desloc)
return res

```

Podemos agora escrever um programa para a codificação e descodificação de mensagens. Este programa, correspondente à função sem argumentos **codificador**, utiliza a função **cod\_descod** 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 efetua a operação solicitada e mostra o resultado obtido. O programa usa um deslocamento de 5 na cifra de substituição.

```

def codificador():

    original = input('Introduza uma mensagem\n-> ')
    tipo = input('C para codificar ou D para descodificar\n-> ')

    if tipo in ('C', 'c'):
        print('A mensagem codificada é:\n',
              cod_descod(original, 5, 'c'))
    elif tipo in ('D', 'd'):
        print('A mensagem descodificada é:\n',
              cod_descod(original, 5, 'd'))
    else :
        print('Oops .. não sei o que fazer')

```

A seguinte interação 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 ações.

```

>>> codificador()
Introduza uma mensagem
-> O PYTHON E DIVERTIDO

```

```
C para codificar ou D para descodificar
-> c
A mensagem codificada é:
T UDYMTS J INAJWYNIT
>>> codificador()
Introduza uma mensagem
-> T UDYMTS J INAJWYNIT
C para codificar ou D para descodificar
-> D
A mensagem descodificada é:
O PYTHON E DIVERTIDO
```

## 4.4 Notas finais

Este foi o primeiro capítulo em que abordámos tipos estruturados de dados. Considerámos dois tipos, os tuplos e as cadeias de carateres, 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 em Python a função `explode` que recebe um número inteiro, verificando a correção do seu argumento, e devolve o tuplo contendo os dígitos desse número, pela ordem em que aparecem no número. Por exemplo

```
>>> explode(34500)
(3, 4, 5, 0, 0)
```

```
>>> explode(3.5)
ValueError: explode: argumento não inteiro
```

3. Escreva em Python a função `implode` que recebe um tuplo contendo algarismos, verificando a correção do seu argumento, e devolve o número inteiro contendo os algarismos do tuplo, pela ordem em que aparecem. Por exemplo

```
>>> implode((3, 4, 0, 0, 4))
34004
>>> implode((2, 'a', 5))
ValueError: implode: elemento não inteiro
```

Escreva duas versões da sua função, uma utilizando um ciclo `while` e outra utilizando um ciclo `for`.

4. Escreva em Python a função `filtra_pares` que recebe um tuplo contendo algarismos, verificando a correção do seu argumento, e devolve o tuplo contendo apenas os algarismos pares. Por exemplo

```
>>> filtra_pares((2, 5, 6, 7, 9, 1, 8, 8))
(2, 6, 8, 8)
```

5. Recorrendo às funções `explode`, `implode` e `filtra_pares`, escreva em Python a função `algarismos_pares` que recebe um inteiro e devolve o inteiro que apenas contém os algarismos pares do número original. Por exemplo,

```
algarismos_pares(6643399766641)
6646664
```

6. 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
```

7. 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
```

8. 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
```

9. 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)
```

10. 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 dados que existem no tuplo original. Por exemplo,

```
>>> soma_els_atomicos((3, (((((6, (7, )), ), ), ), ), 2, 1))
19
>>> soma_els_atomicos((((), ), ))
0
```

11. 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)
```

12. 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
```

13. Escreva em Python duas funções, `cc_para_int` e `int_para_cc`, que convertem, respectivamente, uma cadeia de caracteres num inteiro e um inteiro numa cadeia de caracteres. Por exemplo

```
>>> cc_para_int('bom dia')
98111109032100105097
>>> int_para_cc(97098)
'ab'
>>> int_para_cc(cc_para_int('bom dia'))
'bom dia'
```

14. A partir do dia 1 de janeiro de 2007, o ISBN foi alterado passando a ser constituído por 13 dígitos. O seu dígito de controle (o dígito mais à direita) é tal que se  $x_i$  representar o dígito na  $i$ -ésima posição a contar da esquerda, o dígito de controle  $x_{13}$  é escolhido de modo que a soma

$$\sum_{i=1}^{13} p(i) \times x_i$$

dê resto zero quando dividida por 10. Nesta expressão

$$p(i) = \begin{cases} 1 & \text{se } i \text{ é par} \\ 3 & \text{em caso contrário} \end{cases}$$

Escreva um programa para validar um ISBN com 13 dígitos.

# 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 dados, 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 de certo modo semelhantes às listas do Python, sendo conhecidos por *vetores* ou por *tabelas*<sup>1</sup>.

### 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 sintaticamente pelas seguintes expressões em notação BNF<sup>2</sup>:

---

<sup>1</sup>Em inglês, “array”.

<sup>2</sup>Devemos notar que esta definição não está completa e que a definição de `{dicionário}` é apresentada no Capítulo 7.

Operação	Tipo dos argumentos	Valor
$l_1 + l_2$	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 $i_2 - 1$ .
<del>(els)</del>	Lista e inteiro(s)	Em que <i>els</i> 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	<b>True</b> se o elemento $e$ pertence à lista $l$ ; <b>False</b> 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⟩

```

As seguintes entidades representam listas em Python [1, 2, 3], [2, (1, 2)], [‘a’], []. A lista [] tem o nome de *lista vazia*. 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 [1, [2], [[3]]].

Sobre as listas podemos efetuar as operações apresentadas na Tabela 5.1. Note-se a semelhança entre estas operações e as operações sobre tuplos e sobre cadeias de caracteres, apresentadas, respetivamente, nas tabelas 4.1 e 4.3, sendo a exceção 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 interação 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 interação:

```
>>> 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]
```

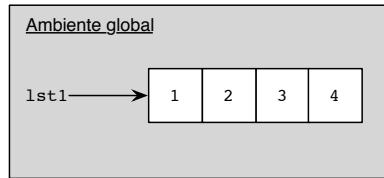


Figura 5.1: Ambiente após a definição da lista lst1.

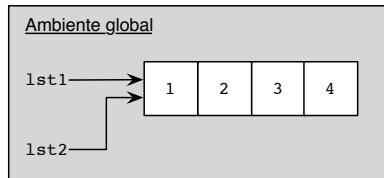


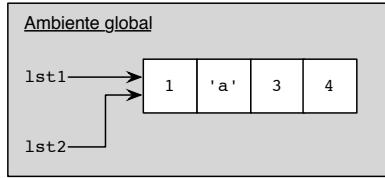
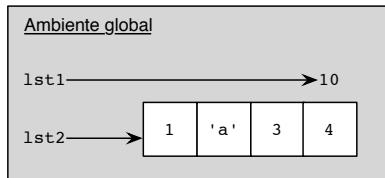
Figura 5.2: Ambiente após a atribuição lst2 = lst1.

```
>>> lst1
[1, 'a', 3, 4]
```

Nesta interação, começamos por definir a lista lst1, após o que definimos a lista lst2 como sendo igual a lst1. Ao alterarmos a lista lst2 estamos indiretamente a alterar a lista lst1. Este comportamento, aparentemente estranho, é explicado quando consideramos o ambiente criado por esta interação. 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 especifica 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 localização em memória da lista [1, 2, 3, 4]), após o que cria o nome lst2, associando-o ao valor desta expressão. Esta instrução origina o ambiente apresentado na Figura 5.2. As listas lst1 e lst2 correspondem a *pseudónimos*<sup>3</sup> 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.

---

<sup>3</sup>Do 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 interação anterior do seguinte modo:

```
>>> lst1 = 10
>>> lst1
10
>>> lst2
[1, 'a', 3, 4]
```

Esta nova interação 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 apenas a passagem por valor.

### 5.2.1 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 a 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 da 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):
    print('antes da troca: x =', x, 'y =', y)
    x, y = y, x # os valores são trocados
    print('depois da troca: x =', x, 'y =', y)
```

e a seguinte interação que utiliza a função `troca`:

```
>>> x = 3
>>> y = 10
>>> troca(x, y)
antes da troca: x = 3 y = 10
depois da troca: x = 10 y = 3
>>> x
3
>>> y
10
```

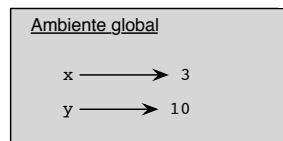


Figura 5.5: Ambiente global antes da invocação de `troca`.

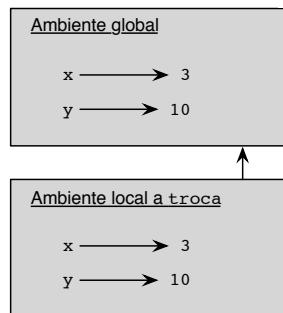


Figura 5.6: Ambiente local criado com a invocação de `troca`.

As duas primeiras linhas desta interação têm como efeito a criação dos nomes `x` e `y` no ambiente global como se mostra na Figura 5.5. Quando o Python 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, não modificando o ambiente global. 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, respectivamente, os valores 10 e 3, após o que atribui estes valores, respectivamente, às variáveis `x` e `y`. 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 afetar 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 asso-

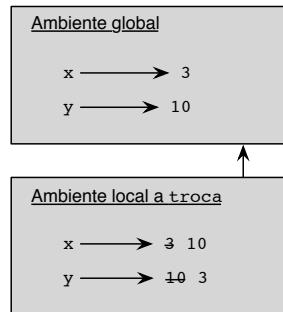


Figura 5.7: Ambientes após a execução da instrução de atribuição.

ciado 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 concretos e os parâmetros formais é uma associação unidirecional de valores. É *unidirecional* 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 reflete-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, ou seja, o parâmetro concreto e o parâmetro formal são pseudónimos da mesma localização em memória.

Embora o Python apenas utilize a passagem por valor, o efeito da passagem por referência pode ser parcialmente<sup>4</sup> ilustrado quando se utiliza um parâmetro concreto correspondente a uma estrutura mutável. Consideremos a função `troca_2` que recebe como argumentos uma lista e dois inteiros e que troca os elementos

---

<sup>4</sup>Parcialmente, porque existem aspectos na passagem por referência que não são simuláveis em Python.

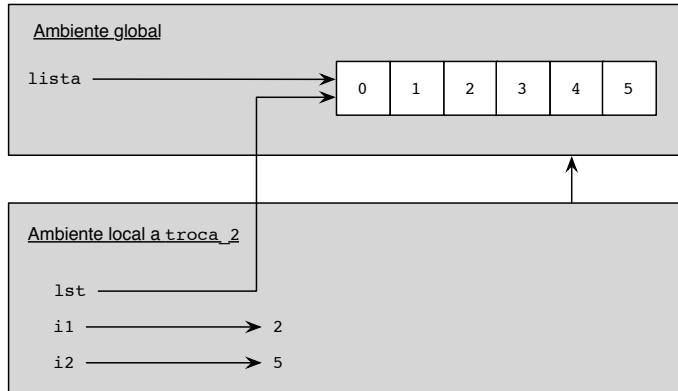


Figura 5.8: Ambientes após o início da execução de `troca_2`.

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 interação:

```
>>> lista = [0, 1, 2, 3, 4, 5]
>>> troca_2(lista, 2, 5)
>>> lista
[0, 1, 5, 3, 4, 2]
```

Neste caso, tendo em atenção a discussão apresentada na página 136, o parâmetro concreto `lst` partilha o mesmo espaço de memória que a variável global `lista`, pelo que qualquer alteração a `lst` reflete-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 seleciona o primeiro elemento da lista, o número 2. Enquanto o número selecionado não for maior que  $\sqrt{n}$  executam-se as seguintes ações: (a) removem-se da lista todos os múltiplos do número selecionado; (b) passa-se ao número seguinte na lista. No final do algoritmo, quando o número selecionado 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  $n$ . Para isso, podemos pensar em utilizar as seguintes instruções:

```
lista = []
for i in range(2, n + 1):
    lista = lista + [i]
```

No entanto, sabendo que a função `range(2, n + 1)` devolve a sequência de inteiros de 2 a  $n$ , podemos recorrer à função embutida `list` para obter o mesmo resultado com a seguinte instrução:

```
lista = list(range(2, n + 1))
```

Após a lista criada, a função 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 índice e modifica essa lista, removendo todos os múltiplos do elemento da lista na posição indicada pelo índice, a função `crivo` será:

```
def crivo(n):
    from math import sqrt
    lista = list(range(2, n+1))
    i = 0
    while lista[i] <= sqrt(n):
        remove_multiplos(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 recorremos a 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]`, removendo os múltiplos do seu primeiro elemento. 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. O ciclo termina no segundo elemento da lista, `lst[1]`.

```
def remove_multiplos(l, i):
    el = l[i]
    for j in range(len(l)-1, i, -1):
        if l[j] % el == 0:
            del(l[j])
```

Com a função `crivo` podemos gerar a interação:

```
>>> 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 atividades quotidianas. Procuramos páginas com determinadas palavras no Google, palavras em dicionários, livros em bibliotecas, produtos em supermercados, etc. Com o aumento constante da quantidade de informação armazenada por computadores, é natural que a procura de informação se tenha tornado uma das atividades 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, desenvolvemos 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). Convencionamos 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*. A função `procura` utilizando a procura sequencial será:

```
def procura(lst, chave):
    for i in range(len(lst)):
        if lst[i] == chave:
            return i
    return -1
```

Com a qual podemos obter a interação:

```
>>> 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 inspeção 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 efetuada se

encontrem ordenados. Utilizando a procura binária, consideramos o elemento que se encontra no meio da lista:

1. Se o elemento no meio da lista for igual ao elemento que estamos a procurar, então a procura termina;
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 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.

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, respetivamente, 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 efetuar a procura, consequentemente, o custo com-

putacional da inserção de um novo elemento na lista é maior se usarmos este método do que se usarmos a procura sequencial. Contudo, como as procuras 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 armazenados 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 (realizado através da função `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 efetuada pela função `ordena`, que recebe uma lista e devolve, por referência, a lista ordenada.

```
def prog_ordena():
    elementos = le_elementos()
    print('Elementos fornecidos: ', elementos)
    ordena(elementos)
    print('Elementos ordenados : ', elementos)
```

Antes de abordar a função `ordena`, vamos considerar a função `le_elementos`. Esta função solicita ao utilizador a introdução dos elementos a ordenar, separados por espaços em branco, e devolve a lista contendo os inteiros fornecidos. 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 começa por ignorar todos os espaços em branco iniciais existentes em `el`. Ao encontrar um símbolo que não corresponda ao espaço em branco, esta função cria a cadeia de caracteres correspondente aos algarismos do número, transformando depois essa cadeia de caracteres num número recorrendo à função `eval`. 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.

```
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):
        # Transforma um elemento num inteiro
        num = ''
        while i < len(el) and el[i] != ' ':
            num = num + el[i]
            i = i + 1
        elementos = elementos + [eval(num)]

        # Ignora espaços em branco depois do número
        while i < len(el) and el[i] == ' ':
            i = i + 1
    return elementos
```

Com uma função de ordenação apropriada, o programa `prog_ordena` permite gerar a interação:

```
>>> 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 da função `ordena`. Esta função recebe como parâmetro uma lista contendo os elementos a ordenar (correspondendo ao parâmetro formal `1st`), 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 consideramos é conhecido por ordenação *por borbulhamento*<sup>5</sup>. A ideia 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 efetuar várias passagens. A lista encontra-se ordenada quando se efetua 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 efetua 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.

---

<sup>5</sup>Em inglês, “bubble sort”.

Dado que em cada passagem pelo ciclo colocamos o maior elemento da lista na sua posição correta<sup>6</sup>, 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):

    maior_indice = len(lst) - 1
    nenhuma_troca = False # garante que o ciclo while é executado

    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*<sup>7</sup> 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*:

---

<sup>6</sup>Como exercício, o leitor deve convencer-se deste facto.

<sup>7</sup>Em inglês, “Shell sort”, em honra ao seu criador Donald Shell [Shell, 1959].

```
def ordena(lst):
    intervalo = len(lst) // 2
    while not intervalo == 0:
        nenhuma_troca = False
        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 seleção

Uma terceira alternativa de ordenação que iremos considerar, a ordenação *por seleção*<sup>8</sup>, consiste em percorrer os elementos a ordenar e, em cada passagem, colocar um elemento na sua posição correta. Na primeira passagem, coloca-se o menor elemento na sua posição correta, na segunda passagem, o segundo menor, e assim sucessivamente. A seguinte função efetua a ordenação por seleção:

```
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]
```

---

<sup>8</sup>Em inglês, “selection sort”.

## 5.6 Exemplo 1

Para ilustrar a utilização de algoritmos de procura e de ordenação, vamos desenvolver um programa que utiliza duas listas, `nomes` e `telefones`, contendo, respetivamente 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 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 9.

O programa começa por ordenar as listas de nomes e de telefones usando a função `ordena_nomes`, após o que interaciona com um utilizador, ao qual fornece o número de telefone da pessoa cujo nome é fornecido ao programa. Esta interação é repetida até que o utilizador forneça, como nome, a palavra `fim`. Para procurar o nome de uma pessoa, 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 seleção 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 seleção. Na realidade, a função `ordena_nomes` utiliza a ordenação por seleção para ordenar os nomes das pessoas, a lista que é usada para procurar os nomes, mas sempre que dois nomes são trocados, a mesma ação é aplicada aos respetivos números de telefone.

```

def lista_tel():

    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? ')

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]

```

## 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 aspeto importante em informática,

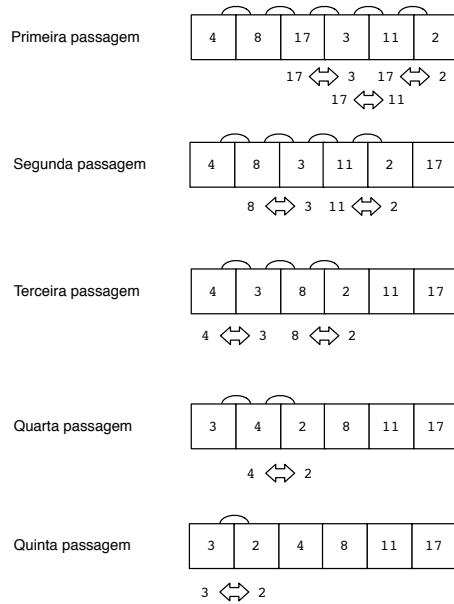


Figura 5.9: Padrão de ordenação utilizando a ordenação por borbulhamento.

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 aspeto 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 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

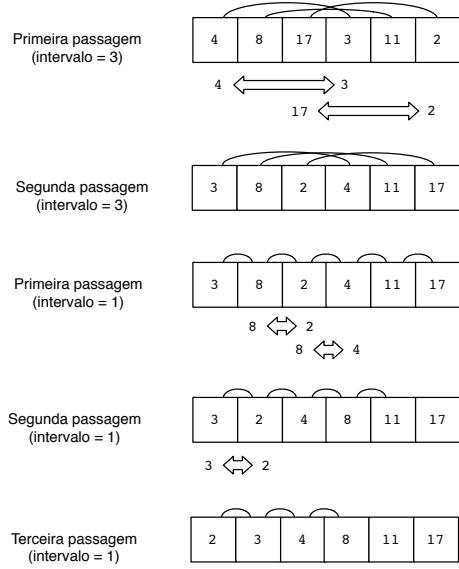
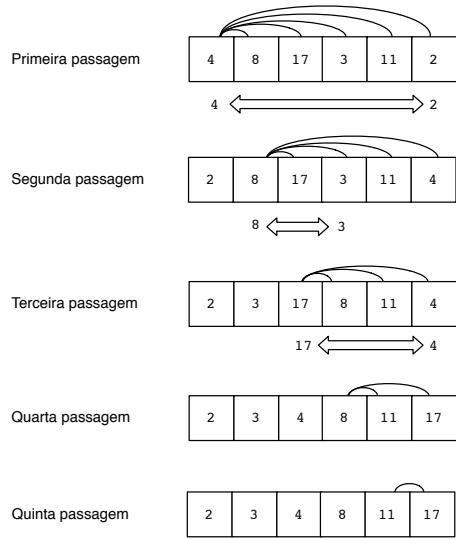
Figura 5.10: Padrão de ordenação utilizando a ordenação *Shell*.

Figura 5.11: Padrão de ordenação utilizando a ordenação por seleção.

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*<sup>9</sup> 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 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 efetuam. Dados dois algoritmos, diremos que o algoritmo que efetuar 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 efetuamos uma comparação (ver a função apresentada na página 146). 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

$$\frac{n}{2^i} < 1$$

---

<sup>9</sup>Do inglês, “order of growth”.

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 efetuamos 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 149 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, efetuam-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 seleção (apresentado na página 151), a operação básica é a seleção do menor elemento de uma lista. O algoritmo de ordenação por seleção 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 seleção é

$$(n - 1) + (n - 2) + \dots + 2 + 1$$

Utilizando a fórmula para calcular a soma de uma progressão aritmética, ob-

temos o seguinte resultado para o número de comparações na ordenação por seleção:

$$(n - 1) \cdot \frac{(n - 1) + 1}{2} = \frac{1}{2} \cdot (n^2 - n).$$

### 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 efetuamos 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 maiúsculo<sup>10</sup>).

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-Landau<sup>11</sup> 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  $o$ ,  $\Omega$ ,  $\omega$  e  $\Theta$  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  $x > x_0$ .

---

<sup>10</sup>Do inglês, “Big-O”.

<sup>11</sup>Em honra aos matemáticos Edmund Landau (1877–1938) e Paul Bachmann (1837–1920).

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 seleção são de ordem  $O(n^2)$ .

Os recursos consumidos por uma função não dependem apenas do algoritmo 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 atualidade. 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  $O(n!)$  requereria mais tempo do que a idade do

$n$	$\log_2(n)$	$n^3$	$6^n$	$n!$
19	0.044 segundos	72.101 segundos	$2.031 \times 10^5$ anos (Homo sapiens)	$4.055 \times 10^7$ anos (tempo dos dinossauros)
20	0.045 segundos	84.096 segundos	$1.219 \times 10^6$ anos	$8.110 \times 10^8$ anos (início da vida na terra)
21	0.046 segundos	97.351 segundos	$7.312 \times 10^6$ anos	$1.703 \times 10^{10}$ anos (superior à idade do universo)
22	0.047 segundos	1.866 minutos	$4.387 \times 10^7$ anos (tempo dos dinossauros)	
23	0.048 segundos	2.132 minutos	$2.632 \times 10^8$ anos	
24	0.048 segundos	2.422 minutos	$1.579 \times 10^9$ anos	
25	0.049 segundos	2.738 minutos	$9.477 \times 10^9$ anos (superior à idade da terra)	
26	0.049 segundos	3.079 minutos	$5.686 \times 10^{10}$ 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.

universo.

## 5.8 Notas finais

Neste capítulo apresentámos o tipo lista, o qual é muito comum em programação, sendo tipicamente conhecido por tipo vetor 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 e de ordenação podem ser consultados em [Knuth, 1973b] e [Cormen et al., 2009].

As listas existentes em Python diferem dos tipos vetor e tabela existentes 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 introduzidos. 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*<sup>12</sup>, as quais são apresentadas no Capítulo 14.2.

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]
```

---

<sup>12</sup>Do inglês, “linked list”.

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 uma função chamada `inverte` que recebe uma lista contendo inteiros e devolve a lista com os mesmos elementos mas 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 interação:

```
>>> 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

$$\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}$$

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 efetuar a ordenação por seleção, criando uma lista com índices. A sua função deve permitir a interação:

```
>>> 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*

Neste capítulo voltamos a considerar as funções em Python, introduzidas no Capítulo 3. Apesar dos programas que temos desenvolvido utilizarem funções, existem muitos aspectos associados à sua utilização que ainda não foram por nós considerados. Começamos por introduzir uma estrutura nas funções, permitindo que estas contenham outras funções, e discutimos as vantagens desta estruturação. Apresentamos o conceito de função recursiva e de programação funcional. Finalmente apresentamos funções que recebem funções como argumentos e funções que devolvem funções.

### 6.1 Estruturação de funções

Para motivar a necessidade de introduzir uma estrutura nas funções, 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
```

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 evitar. Tendo em atenção que  $x^{-n} = 1/x^n$ , podemos pensar na seguinte solução para a função `potencia` 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 interação:

```
>>> 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 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 diretamente utilizada para calcular a potência, eventualmente gerando um ciclo infinito se os seus argumentos não forem os corretos.

O Python apresenta uma solução 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 proteção efetiva dos dados definidos em cada bloco da utilização não autorizada por parte de outros blocos. Podemos

pensar nos blocos como sendo egoístas, não permitindo o acesso aos seus dados 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 ações 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` é usada com os argumentos corretos. 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 ações 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 em função>
```

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: o corpo de cada 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 interação:

```

>>> potencia(3, 2)
9
>>> potencia(3, -2)
0.1111111111111111
>>> potencia
<function potencia at 0x10f45d0>
>>> potencia_aux
NameError: name 'potencia_aux' is not defined

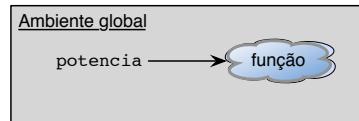
```

Esta interação 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 6.1 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 retângulo, dentro do qual aparece a informação do bloco. Note-se que dentro do bloco da função `potencia` existe um bloco que corresponde à função `potencia_aux`. Nesta ótica, o comportamento que obtivemos na interação 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 efetivamente evita a possibilidade de utilizarmos diretamente a função `potencia_aux`, como a interação anterior o demonstra.

Para compreender o funcionamento da função `potencia`, iremos seguir o funcionamento do Python durante a interação 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 6.2).

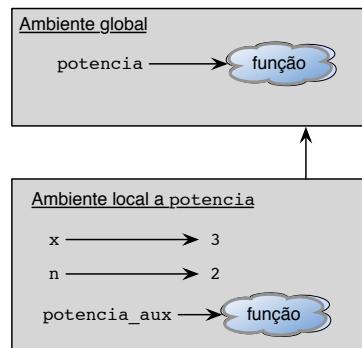
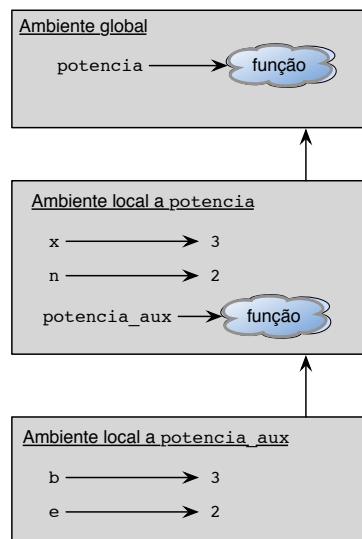
```
def potencia(x, n):
    def potencia_aux(b, e):
        res = 1
        while e != 0:
            res = res * b
            e = e - 1
        return res
    if n >= 0:
        return potencia_aux(x, n)
    else:
        return 1 / potencia_aux(x, -n)
```

Figura 6.1: Estrutura de blocos da função `potencia`.Figura 6.2: Ambiente global com o nome `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` (`x` e `n`) aos parâmetros concretos (`3` e `2`), 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 6.3. A segunda instrução do corpo de `potencia` corresponde a uma instrução `if`, na qual ambas as alternativas levam à execução da função `potencia_aux`, a qual existe no Ambiente local a `potencia`. A execução desta função leva à criação de um novo ambiente (indicado na Figura 6.4 com o nome Ambiente local a `potencia_aux`), no qual esta função é executada. Depois da execução de `potencia` terminar, os dois ambientes locais desaparecem, voltando-se à situação apresentada na Figura 6.2, e o Python deixa de conhecer o significado de `potencia_aux`, o que justifica a interação apresentada.

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 ativi-

Figura 6.3: Ambiente criado pela execução de `potencia`.Figura 6.4: Ambientes durante a execução de `potencia`.

dade 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 atividade 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 6.5 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):
```

```

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

```

Figura 6.5: Estrutura de blocos da função `raiz`.

```

    palpite = novo_palpite(x, palpite)
    return palpite

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

```

Consideremos agora de um modo mais detalhado a noção de *ambiente*. Sabemos do Capítulo 2 (página 41) que um ambiente contém associações entre nomes e valores. Tipicamente, um ambiente não aparece isolado, estando ligado a um outro ambiente, chamado o *ambiente envolvente* (o qual tem sido indicado nas nossas figuras por uma seta que aponta do ambiente em questão para o ambiente envolvente). Existe um (e só um) ambiente especial, o *ambiente global*, que não tem nenhum ambiente envolvente. Os ambientes estão assim organizados numa estrutura hierárquica, em que cada ambiente está associado ao seu ambiente envolvente.

A sequência de ambientes, desde um dado ambiente ao ambiente global, é utilizada para definir o valor de uma variável num determinado ambiente. O *valor de uma variável* em relação a um ambiente é o valor da associação dessa variável no primeiro ambiente que contém uma associação para essa variável. Se uma variável não tem valor num dado ambiente, ou seja, se não existe nenhuma

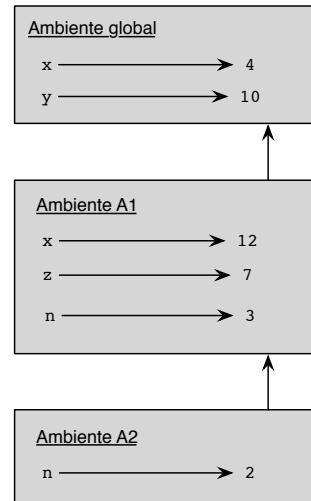


Figura 6.6: Exemplo de ambientes.

ligação para a variável nesse ambiente, a variável diz-se *não ligada* (do inglês “*unbound*”). Esta definição significa que o valor de uma variável *depende* do ambiente em que é considerada.

Consideremos os ambientes apresentados na Figura 6.6. Em relação ao Ambiente A2 a variável n tem o valor 2, a variável z tem o valor 7 e a variável x tem o valor 12; em relação ao Ambiente A1 a variável n tem o valor 3, a variável z tem o valor 7 e a variável x tem o valor 12; em relação ao Ambiente global as variáveis n e z não estão ligadas e a variável x tem o valor 4.

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
```

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 ativa 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 interação:

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

Na interação anterior, ao tentarmos executar a função *potencia\_estranya*, 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\_estranya*. 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 interação anterior), e os que não são globais, a que se dá o nome de *livres*. Consideraremos a seguinte definição da função *potencia\_estranya\_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 objeto computacional, e função `potencia_estranha_2` vai utilizar essa ligação. Na Figura 6.7 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 procura o seu valor no ambiente Ambiente local a `potencia_tambem_estranha`, 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

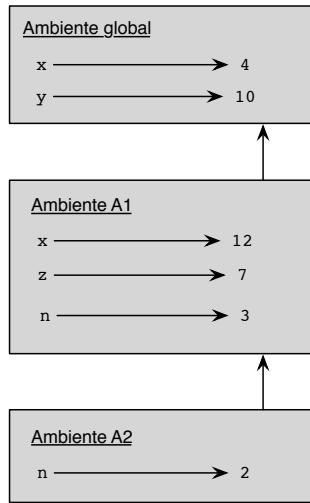


Figura 6.7: Ambientes criados pela avaliação de `potencia_tambem_estranya(4, 2)`.

instruções numa interação 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 aspeto é ilustrado com a seguinte função:

```
def potencia_ainda_mais_estranya(x, n):

    def potencia_estranya_3(n):
        pot = 1
        x = 7
        while n > 0:
            pot = pot * x
```

```

n = n - 1
return pot

print('x=', x)
res = potencia_estranya_3(n)
print('x=', x)
return res

```

a qual permite gerar a seguinte interação:

```

>>> potencia_ainda_mais_estranya(4, 2)
x= 4
x= 4
49

```

Com efeito, a instrução `x = 7` no corpo da função `potencia_estranya_3` cria o nome `x` como uma variável local à função `potencia_estranya_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 BNF<sup>1</sup>:

`<instrução global> ::= global <nomes>`

Quando a instrução `global` aparece no corpo de uma função, o Python considera que `<nomes>` 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_estranya_2(n):

    def potencia_estranya_4(n):
        global x
        pot = 1
        x = 7
        while n > 0:

```

---

<sup>1</sup>Na realidade, esta instrução corresponde a uma *diretiva* para o interpretador do Python.

```

pot = pot * x
n = n - 1
return pot

print('x (antes de potencia_estranya_4) =', x)
res = potencia_estranya_4(n)
print('x (depois de potencia_estranya_4) =', x)
return res

```

a qual permite gerar a interação:

```

>>> x = 4
>>> potencia_ainda_mais_estranya_2(2)
x (antes de potencia_estranya_4) = 4
x (depois de potencia_estranya_4) = 7
49

```

Devido à restrição imposta pela instrução `global` de não poder alterar parâmetros formais, na função `potencia_ainda_mais_estranya_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 abstração 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 objeto computacional associado a esse nome é compartilhado por vários blocos do programa, que o podem consultar e, eventualmente, modificar.

## 6.2 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

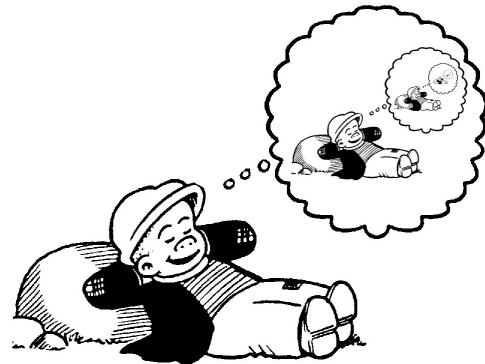


Figura 6.8: Exemplo de uma imagem recursiva.

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, um exemplo das quais apresentamos na Figura 6.8<sup>2</sup>.

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

---

<sup>2</sup>Reproduzido com autorização de United Feature Syndicate. © United Feature Syndicate, Inc. Nancy and Sluggo are a registered trademark of United Feature Syndicate, Inc.

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:

$$n! = n \cdot (n - 1) \cdot (n - 2) \dots 2 \cdot 1.$$

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.

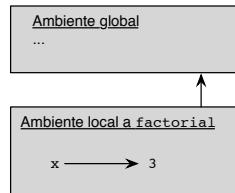
O poder das definições recursivas baseia-se na possibilidade de definir um conjunto infinito utilizando uma frase finita. Do mesmo modo, um número arbitrariamente grande de cálculos pode ser especificado através de uma função recursiva, mesmo que esta função não contenha, explicitamente, estruturas de repetição.

Suponhamos que desejávamos definir a função factorial em Python. Podemos usar a definição não recursiva, descodificando o padrão correspondente às retilicências, 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 diretamente a definição recursiva, dando origem à seguinte função:

```
def factorial(n):
    if n == 0:
```

Figura 6.9: Primeira chamada a `fatorial`.

```

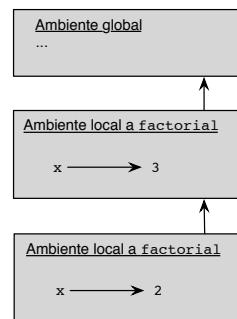
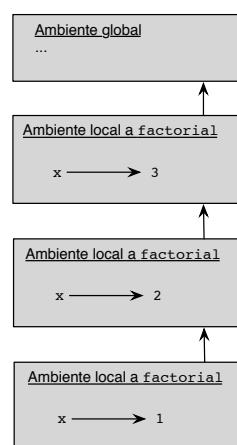
return 1
else:
    return n * fatorial(n-1)
  
```

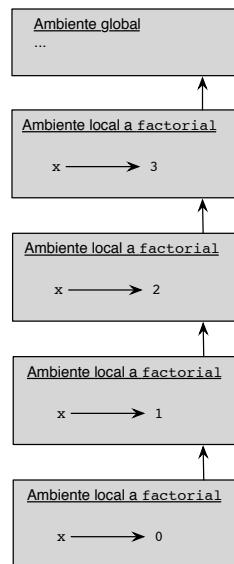
Esta segunda versão de `fatorial` não contém explicitamente nenhum ciclo.

Como exercício, iremos seguir o funcionamento da versão recursiva da função `fatorial` para calcular o fatorial de 3. Ao avaliar `fatorial(3)`, o valor 3 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.9 (nesta figura e nas seguintes, estamos propósitadamente 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 `False`, o valor devolvido será `3 * fatorial(2)`.

Encontramos agora uma outra expressão que utiliza a função `fatorial` e esta expressão inicia o cálculo de `fatorial(2)`. É importante recordar que neste momento existe um cálculo suspenso, o cálculo de `fatorial(3)`. Para calcular `fatorial(2)`, o valor 2 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.10. Como a expressão `2 == 0` tem o valor `False`, o valor devolvido por `fatorial(2)` é `2 * fatorial(1)` e o valor de `fatorial(1)` terá que ser calculado. Dois cálculos da função `fatorial` estão agora suspensos: o cálculo de `fatorial(3)` e o cálculo de `fatorial(2)`.

Encontramos novamente outra expressão que utiliza a função `fatorial` e esta expressão inicia o cálculo de `fatorial(1)`. Para calcular `fatorial(1)`, o valor 1 é associado ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.11. Como a expressão `1 == 0` tem o valor `False`, o valor devolvido por `fatorial(1)` é `1 * fatorial(0)` e o valor de `fatorial(0)` terá que ser calculado. Três cálculos da função `fatorial` estão agora suspensos: o cálculo de `fatorial(3)`, `fatorial(2)` e `fatorial(1)`.

Figura 6.10: Segunda chamada a `fatorial`.Figura 6.11: Terceira chamada a `fatorial`.

Figura 6.12: Quarta chamada a `fatorial`.

Para calcular `fatorial(0)` associa-se o valor 0 ao parâmetro formal `n` da função `fatorial`, criando-se o ambiente local apresentado na Figura 6.12 e executa-se o corpo da função. Neste caso a expressão `0 == 0` tem valor `True`, pelo que o valor de `fatorial(0)` é 1.

Pode-se agora continuar o cálculo de `fatorial(1)`, que é `1 * fatorial(0) = 1 * 1 = 1`, o que, por sua vez, permite calcular `fatorial(2)`, cujo valor é `2 * fatorial(1) = 2 * 1 = 2`, o que permite calcular `fatorial(3)`, cujo valor é `3 * fatorial(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.

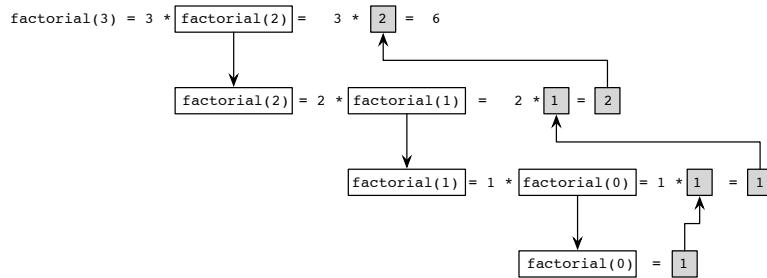


Figura 6.13: Encadeamento das operações no cálculo de `fatorial(3)`.

Como vimos, durante a avaliação de uma expressão cujo operador é `fatorial`, gera-se um encadeamento de operações suspensas à espera do valor de outras operações cujo operador é o próprio `fatorial`. Na Figura 6.13 mostramos o encadeamento das operações geradas durante o cálculo de `fatorial(3)`.

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 fatorial 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 fatorial, 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. O máximo divisor comum pode ser definido do seguinte modo<sup>3</sup>:

$$mdc(m, n) = \begin{cases} m & \text{se } n = 0 \\ mdc(n, m \% n) & \text{se } n > 0 \end{cases}$$

Podemos agora aplicar diretamente 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
```

Para ilustrar o poder das funções recursivas, consideremos a função `alisa` apresentada na página 110, 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 dados (inteiros, reais ou valores lógicos) do tuplo original. Esta função pode ser definida recursivamente do seguinte modo:

---

<sup>3</sup>Nesta definição usamos o símbolo do resto da divisão inteira do Python.

```
def alisa(t):
    if t == ():
        return t
    elif isinstance(t[0], tuple):
        return alisa(t[0]) + alisa(t[1:])
    else:
        return (t[0],) + alisa(t[1:])
```

Se o tuplo recebido pela função for o tuplo vazio, o seu valor é o tuplo vazio; se o primeiro elemento do tuplo for um tuplo, o valor da função corresponde a juntar o tuplo que resulta de “alisar” o primeiro elemento ao tuplo que resulta de “alisar” os restantes elementos do tuplo; em caso contrário, o resultado da função é o tuplo que se obtém adicionando o primeiro elemento do tuplo ao resultado de “alisar” os restantes elementos do tuplo.

### 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, atualiza 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 aspeto 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 objetivos 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 diretamente esta definição para a seguinte função em Python<sup>4</sup>:

```
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 repetição 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.

---

<sup>4</sup>Propositalmente, não estamos a fazer nenhuma verificação em relação aos possíveis valores fornecidos a esta função.

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 88 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 Funções como parâmetros

Nos capítulos anteriores argumentámos que as funções correspondem a abstrações 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. Poderíamos escrever programas sem nunca recorrer à função `quadrado` mas sempre que precisássemos de utilizar o quadrado de uma entidade particular teríamos que recorrer à expressão que calcula o quadrado, `25 * 25`, `x * x`, `y * y`. A abstração 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 abstração através da qual podemos referir-nos à operação capturada pelo conceito.

Nesta secção apresentamos conceitos mais abstratos 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 três funções que, embora distintas, englobam um padrão de cálculo comum<sup>5</sup>. Estas funções calculam, respetivamente, a soma dos números naturais entre `l_inf` e `l_sup`

```
def soma_naturais(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + l_inf
        l_inf = l_inf + 1
    return soma
```

a soma dos quadrados dos números naturais entre `l_inf` e `l_sup`

---

<sup>5</sup>Este exemplo foi adaptado de [Abelson et al., 1996], páginas 57 e 58.

```
def soma_quadrados(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + quadrado(l_inf)
        l_inf = l_inf + 1
    return soma
```

e a soma dos quadrados dos números naturais ímpares entre `l_inf` e `l_sup`<sup>6</sup>

```
def soma_inv_quadrados_impares(l_inf, l_sup):
    soma = 0
    while l_inf <= l_sup:
        soma = soma + 1 / quadrado(l_inf)
        l_inf = l_inf + 2
    return soma
```

Estas funções permitem gerar a interação:

```
>>> soma_naturais(1, 40)
820
>>> soma_quadrados(1, 40)
22140
>>> soma_inv_quadrados_impares(1, 40)
1.2212031520286797
```

Estas funções partilham um padrão de cálculo em comum, diferindo entre si nos seguintes aspetos: (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. Poderíamos criar cada uma destas funções substituindo os símbolos não terminais no seguinte padrão de cálculo:

```
def <nome>(linf, lsup):
    soma = 0
    while linf <= lsup:
        soma = soma + <calc_termo>(linf)
        linf = <prox>(linf)
    return soma
```

---

<sup>6</sup>Esta função assume que `l_inf` e `l_sup` são números ímpares.

Os símbolos não terminais no padrão anterior assumem valores particulares para cada uma das funções. A existência deste padrão mostra que existe uma abstração escondida subjacente a este cálculo. Os matemáticos identificaram esta abstração através do conceito de somatório:

$$\sum_{n=l_{inf}}^{l_{sup}} f(n) = f(l_{inf}) + \cdots + f(l_{sup})$$

O poder da abstração correspondente ao somatório permite lidar com o *conceito de soma* em vez de tratar apenas com somas particulares. A existência da abstração correspondente ao somatório leva-nos a pensar em definir uma função correspondente a esta abstração 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 efetuar sobre os objetos 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 (`l_inf` e `l_sup`), 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 interação:

```
>>> 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. As funções que utilizam funções como argumentos são conhecidas por *funções de ordem superior* ou *funcionais*.

Existem dois aspectos que é importante observar em relação à função anterior. Em primeiro lugar, a função `somatorio` não foi exatamente 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*<sup>7</sup>. 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  $\lambda$  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:

$\langle\text{função anónima}\rangle ::= \text{lambda } \langle\text{parâmetros formais}\rangle : \langle\text{expressão}\rangle$

Nesta definição,  $\langle\text{expressão}\rangle$  corresponde a uma expressão em Python.

Ao encontrar uma  $\langle\text{função anónima}\rangle$ , o Python cria uma função cujos parâmetros formais correspondem a  $\langle\text{parâmetros formais}\rangle$  e cujo corpo corresponde a  $\langle\text{expressão}\rangle$ . Quando esta função anónima é executada, os parâmetros concretos são associados aos parâmetros formais, e o valor da função é o valor da  $\langle\text{expressão}\rangle$ . Note-se que numa função anónima não existe uma instrução `return` (estando esta implicitamente associada a  $\langle\text{expressão}\rangle$ ).

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 interação:

```
>>> (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 batizada, podemos utilizar a função `somatorio`,

---

<sup>7</sup>Do inglês, “lambda calculus” [Church, 1941].

recorrendo a funções anónimas como mostra a seguinte interação (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 `fatorial` apresentada na página 181, podemos calcular a soma dos fatoriais dos 20 primeiros números naturais através de:

```
>>> somatorio(fatorial, 1, lambda x : x + 1, 20)
2561327494111820313
```

### 6.4.1 Funcionais sobre listas

Com a utilização de listas é vulgar recorrer a um certo número de funcionais, os quais se classificam em transformadores, filtros e acumuladores.

- 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, lst):
    res = list()
    for e in lst:
        res = res + [tr(e)]
    return res
```

Em alternativa, a função `transforma` pode ser escrita como uma função recursiva do seguinte modo:

```
def transforma(tr, lst):
    if lst == []:
        return lst
    else:
        return [tr(lst[0])] + transforma(tr, lst[1:])
```

A seguinte interação 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, lst):
    res = list()
    for e in lst:
        if teste(e):
            res = res + [e]
    return res
```

ou, alternativamente, através da seguinte função recursiva:

```
def filtra(teste, lst):
    if lst == []:
        return lst
    elif teste(lst[0]):
        return [lst[0]] + filtra(teste, lst[1:])
    else:
        return filtra(teste, lst[1:])
```

A seguinte interação 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 função<sup>8</sup>:

```
def acumula(fn, lst):
    res = lst[0]
    for i in range(1, len(lst)):
        res = fn(res, lst[i])
    return res
```

ou, alternativamente, através da seguinte função recursiva:

```
def acumula(fn, lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return fn(lst[0], acumula(fn, lst[1:]))
```

A seguinte interação 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
```

A seguinte utilização de funcionais calcula a soma dos quadrados dos elementos ímpares da lista [2, 4, 5, 29, 30]:

```
acumula(lambda x, y : x + y,
        transforma(quadrado,
                    filtra(lambda x : x % 2 == 1,
                           [2, 4, 5, 29, 30])))
```

---

<sup>8</sup>Como exercício, o leitor deverá explicar a razão da variável `res` ser inicializada para `lst[0]` e não para 0.

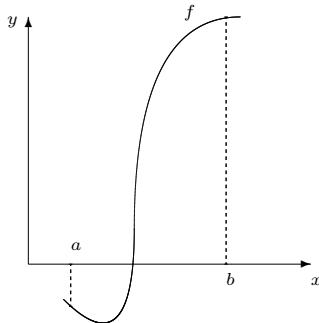


Figura 6.14: Situação à qual o método do intervalo é aplicável.

#### 6.4.2 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.

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 6.14).

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, respetivamente, `p.neg` e `p.pos`. Esta função calcula o zero da função no intervalo especificado, utilizando o método do intervalo.

```
def calcula_raiz(f, p_neg, p_pos):
    while not suf_perto(p_neg, p_pos):
        p_medio = (p_neg + p_pos) / 2
        if f(p_medio) > 0:
            p_pos = p_medio
        elif f(p_medio) < 0:
            p_neg = p_medio
        else:
            return p_medio
    return (p_neg + p_pos) / 2
```

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 seleção serve para lidar com os casos em que `f(p_medio)` não é nem positivo, nem negativo, ou seja, `f(p_medio)` 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 que a função `calcula_raiz` parte do pressuposto que os valores extremos do intervalo correspondem a valores da função, `f`, com sinais opostos e que `f(p_neg) < 0 < f(p_pos)`. Se algum destes pressupostos não se verificar, a função não calcula corretamente 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 interação:

```

>>> 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, p_neg, p_pos):
        while not suf_perto(p_neg, p_pos):
            p_medio = (p_neg + p_pos) / 2
            if f(p_medio) > 0:
                p_pos = p_medio
            elif f(p_medio) < 0:
                p_neg = p_medio
            else:
                return p_medio
        return (p_neg + p_pos) / 2

    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')

```

## 6.5 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 abstrações mais gerais do que as obtidas até agora. Nesta secção discutimos funções como valores produzidos por funções.

### 6.5.1 Cálculo de derivadas

Suponhamos que queremos desenvolver uma função que calcula o valor da derivada de uma função real de variável real,  $f$ , num dado ponto. 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 ao argumento `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 interação calcula a derivada da função `quadrado` (apresentada na página 76) para os pontos 3 e  $10^9$ :

```
>>> derivada_ponto(quadrado, 3)
6.000009999951316
>>> derivada_ponto(quadrado, 10)
20.00000999942131
```

Em Matemática, após a definição de derivada num ponto, é possível 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 abstração. Repare-se que a derivada num ponto arbitrário  $x$  é calculada substituindo  $x$  por  $a$  na função que apresentámos. Consideremos a seguinte função:

```
def fn_derivada(x):
    return (f(x + dx) - f(x)) / dx
```

Esta função utiliza uma variável livre, `f`, correspondente a uma função e calcula o valor da derivada de `f` para qualquer ponto do seu domínio. Com esta função podemos gerar a interação, a qual pressupõe a definição da função `quadrado`:

```
>>> f = quadrado
>>> dx = 0.00001
>>> fn_derivada(3)
6.000009999951316
>>> fn_derivada(10)
20.00000999942131
```

---

<sup>9</sup>Note-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.

O que ainda 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 `derivada` recebe como parâmetro uma função, `f`, e devolve como valor a função (ou seja, o valor de `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 interação:

```
>>> derivada(quadrado)
<function fn_derivada at 0x10f45d0>
>>> derivada(quadrado)(3)
6.00000999951316
>>> derivada(quadrado)(10)
20.00000999942131
```

Nada nos impede de dar um nome a uma função derivada, obtendo a interação:

```
>>> der_quadrado = derivada(quadrado)
>>> der_quadrado(3)
6.00000999951316
```

### 6.5.2 Raízes pelo método de Newton

Na Secção 6.4.2 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

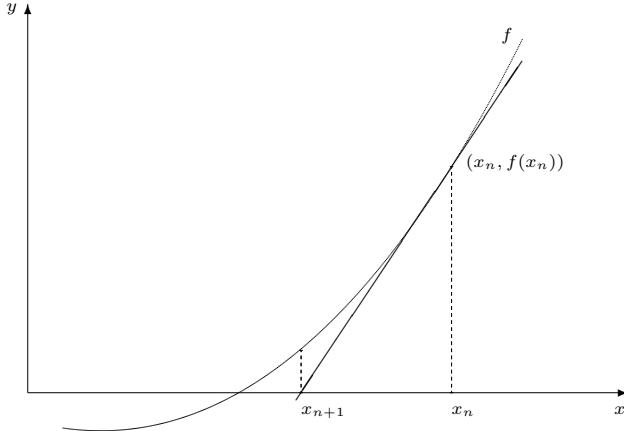


Figura 6.15: Representação gráfica subjacente ao método de Newton.

a tangente ao gráfico da função no ponto  $(x_n, f(x_n))$  intersetra o eixo dos  $xx$  (Figura 6.15). 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 palpito),  $x_0$ , e gerar os valores:

$$\begin{aligned} &x_0 \\ &x_1 = t_N(x_0) \\ &x_2 = t_N(x_1) \\ &\vdots \\ &x_n = t_N(x_{n-1}) \end{aligned}$$

Podemos agora definir as seguintes funções para calcular a raiz de uma função

(representada por  $f$ ), com base num palpite inicial ( $\text{palpite}$ ):

```
def met_newton(f, palpite):
    tf_N = transf_newton(f)
    while not boa_aprox(f(palpite)):
        palpite = tf_N(palpite)
    return palpite

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  $\arctg(x)$  é o número  $y$  tal que  $x = \tg(y)$ , ou seja, para um dado  $x$ , o valor de  $\arctg(x)$  corresponde ao zero da equação  $\tg(y) - x = 0$ . 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 interação:

```
>>> from math import *
>>> arctg(0.5)
0.4636478065118169
```

## 6.6 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 aspetos, a utilização de funções como argumentos para funções e a utilização de funções que produzem objetos computacionais que correspondem a funções. Esta generalização permite-nos definir abstrações 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.

## 6.7 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:

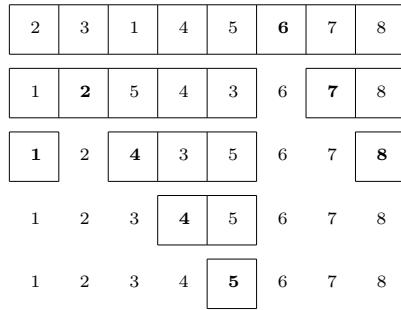
$$\sin(x) = x \left(1 - \left(\frac{x}{\pi}\right)^2\right) \left(1 - \left(\frac{x}{2\pi}\right)^2\right) \left(1 - \left(\frac{x}{3\pi}\right)^2\right) \dots$$

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
```

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

Figura 6.16: Passos seguidos no *quick sort*.

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 6.16 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 efetuar a ordenação de uma lista utilizando *quick sort*.

8. Considere a função **derivada** apresentada na página 203. 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 interação:

```
>>> 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 6.17):

$$\sum_{i=1}^n f\left(\frac{x_i + x_{i-1}}{2}\right) (x_i - x_{i-1})$$

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 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.

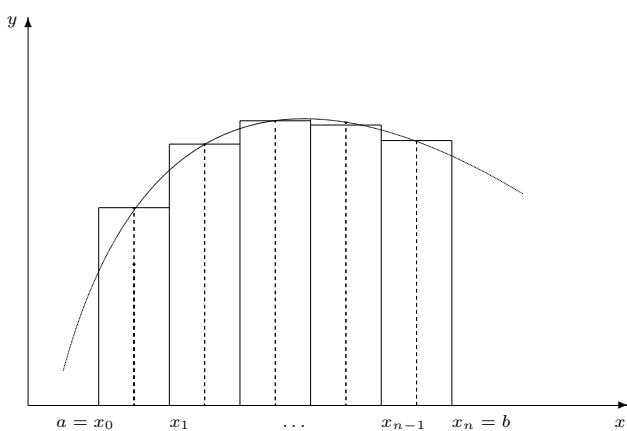


Figura 6.17: Área aproximada sob a curva.

## 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 vários aspectos da programação, estudámos o modo de criar funções, utilizámos alguns tipos de dados existentes em Python e estudámos o modo de definir novos tipos de dados. Embora sejamos já capazes de escrever programas com alguma complexidade, 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 subjacente à computação é o processo computacional. Na atividade de programação planeamos a sequência de ações a serem executadas por um programa. Para podermos desenvolver programas adequados a um dado fim é essencial que tenhamos uma compreensão clara dos tipos de processos computacionais gerados pelos diferentes tipos de funções. Este aspeto 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 função<sup>1</sup>. 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 `fatorial` apresentada na Secção 6.2:

```
def fatorial(n):
    if n == 0:
        return 1
    else:
        return n * fatorial(n - 1)
```

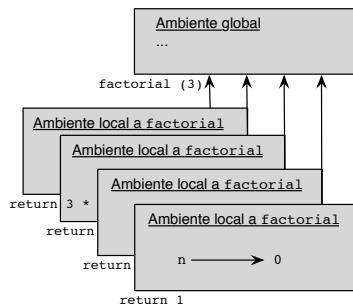
Vimos que durante a avaliação de uma expressão cujo operador é `fatorial`, se gera uma sequência de ambientes locais, todos referentes a esta função (representados na Figura 7.1 para o cálculo de `fatorial(3)`), cada um dos quais contém uma operação adiada: a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica.

Este encadeamento de ambientes começa por gerar uma expansão da memória ocupada durante a execução da função, expansão essa que vai ocorrendo até que se encontra o caso terminal da função, sendo seguida por uma fase de contração à medida que os ambientes vão sendo libertados (ver os diagramas de ambientes apresentados nas páginas 182 a 184).

O processo gerado pela função `fatorial` pode ser representado pelo seguinte encadeamento de chamadas à função `fatorial`, em que cada chamada recursiva é escrita mais para a direita:

---

<sup>1</sup>Note-se que, na realidade, a função também define a evolução global do processo.

Figura 7.1: Ambientes criados pela avaliação de `fatorial(3)`.

```

fatorial(3)
| fatorial(2)
| | fatorial(1)
| | | fatorial(0)
| | | return 1
| | return 1
| return 2
return 6
  
```

Consideremos a função `potencia` apresentada na Secção 6.3:

```

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

Podemos observar que durante a avaliação de uma expressão cujo operador é `potencia`, também se gera uma sequência de ambientes, todos referentes à função `potencia`. Nestes ambientes, a execução da operação de multiplicação vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função, após o que se efetuam as operações que foram sendo adiadas. Por exemplo, para o cálculo de `potencia(2, 5)`, este processo pode ser representado do seguinte modo:

```

potencia(2, 5)
| potencia(2, 4)
| | potencia(2, 3)
| | | potencia(2, 2)
| | | | potencia(2, 1)
| | | | | potencia(2, 0)
| | | | | return 1
| | | | | return 2
| | | | | return 4
| | | | | return 8
| | | | | return 16
| | | | | return 32

```

Como terceiro exemplo, consideremos a função, `soma_elementos`, para calcular a soma dos elementos de uma lista<sup>2</sup>. Podemos escrever esta função do seguinte modo:

```

def soma_elementos(lst):
    if lst == []:
        return 0
    else:
        return lst[0] + soma_elementos(lst[1:])

```

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 considerar a forma do processo gerado pela avaliação de `soma_elementos([1, 2, 3, 4])`:

```

soma_elementos([1, 2, 3, 4])
| soma_elementos([2, 3, 4])
| | soma_elementos([3, 4])
| | | soma_elementos([4])

```

---

<sup>2</sup>Consideramos que a lista contém números, pelo que não faremos qualquer teste ao seu conteúdo.

```
| | | soma_elementos([])  
| | | return 0  
| | | return 4  
| | return 7  
| return 9  
return 10
```

Voltamos a deparar-nos com um processo em que se gera uma sequência de ambientes, todos referentes à função `soma_elementos`. Nestes ambientes, a execução da operação de adição vai sendo sucessivamente adiada até se atingir o valor que corresponde à parte básica da função `soma_elementos`, após o que se efetuam as operações que foram sendo adiadas.

Todas as funções que apresentámos nesta secção geram processos computacionais que têm um comportamento semelhante: são caracterizados pela criação de uma sequência de ambientes correspondentes à mesma função, o que origina uma expansão da memória necessária para a execução do processo (*fase de crescimento*), seguida por uma diminuição da memória necessária (*fase de contração*). Este padrão de evolução de um processo é muito comum em programação e tem o nome de processo recursivo. Num *processo recursivo* é criada uma sequência de ambientes correspondentes à mesma função.

Tipicamente, mas não obrigatoriamente, num processo recursivo durante a fase de expansão geram-se operações adiadas e na fase de contração essas operações são executadas.

Em todos os casos que apresentámos, o número ambientes gerados 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 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.

Consideremos a função `fatorial` apresentada na página 181:

```
def fatorial(n):
    fat = 1
    for i in range(n, 0, -1):
        fat = fat * i
    return fat
```

Nesta função, em cada instante, possuímos toda a informação necessária para saber o que já fizemos (o produto acumulado, `fat`) e o que ainda nos falta fazer (o número de multiplicações a realizar, `i`). 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. As variáveis que caracterizam o estado de um processo, podendo recomeçá-lo em qualquer instante dá-se o nome de *variáveis de estado*.

A função `potencia` pode ser escrita do seguinte modo:

```
def potencia(x, n):
    pot = 1
    for i in range(1, n + 1):
        pot = pot * x
    return pot
```

Podemos também escrever a seguinte versão da função `soma_elementos`:

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

As três funções que apresentámos geram processos computacionais que têm um comportamento semelhante. Eles são caracterizados por um certo número de variáveis que fornecem uma descrição completa do estado da computação em cada instante. 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 Wikcionário<sup>3</sup>, a palavra “iterativo” tem o seguinte

---

<sup>3</sup><http://pt.wiktionary.org/wiki/>

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 atualizá-las. Estas variáveis fornecem uma descrição completa do estado da computação em cada momento.

Nos nossos exemplos, o número de operações efetuadas sobre as variáveis de estado cresce linearmente com uma grandeza associada à função (o inteiro para o qual se está a calcular o fatorial, 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 de cauda

O processo de cálculo utilizado na secção anterior para a escrita da função `fatorial` pode também ser traduzido pela função recursiva `fatorial_rec`, na qual `prod_ac` representa o produto acumulado dos números que já multiplicámos e `prox` representa o próximo número a ser multiplicado:

```
def fatorial_rec(prod_ac, prox):
    if prox == 0:
        return prod_ac
    else:
        return fatorial_rec(prod_ac * prox, prox - 1)
```

A função `fatorial_rec` contém uma chamada recursiva, sendo esta chamada, a *última operação* efetuada pela função. Este padrão de computação tem o nome de *recursão de cauda*<sup>4</sup>. Note-se a diferença em relação à função `fatorial` apresentada na página 212, na qual a última operação realizada pela função é uma multiplicação.

Com base na função `fatorial_rec`, podemos agora escrever a seguinte função para o cálculo de `fatorial`, a qual esconde do exterior a utilização das variáveis

---

<sup>4</sup>Do Inglês, “tail recursion”.

```
prod_ac e prox:

def fatorial(n):

    def fatorial_rec(prod_ac, prox):
        if prox == 0:
            return prod_ac
        else:
            return fatorial_rec(prod_ac * prox, prox - 1)

    return fatorial_rec(1, n)
```

Durante a avaliação de uma expressão cujo operador é `fatorial`, gera-se uma sequência de ambientes correspondentes à função `fatorial_rec`, não existindo, neste caso, operações adiadas:

```
fatorial(5)
fatorial_rec(1, 5)
| fatorial_rec(5, 4)
| | fatorial_rec(20, 3)
| | | fatorial_rec(60, 2)
| | | | fatorial_rec(120, 1)
| | | | | fatorial_rec(120, 0)
| | | | | return 120
| | | | | return 120
| | | | return 120
| | | return 120
| | return 120
| return 120
return 120
```

Aplicando o mesmo raciocínio ao cálculo de `potencia`, podemos escrever a função `potencia_rec`, na qual o nome `n_mult` corresponde ao número de multiplicações que temos de executar, e o nome `prod_ac` corresponde ao produto acumulado. Ao iniciar o cálculo, estabelecemos que o produto acumulado é um e o número de multiplicações que nos falta efetuar corresponde ao expoente:

```
def potencia(x, n):

    def potencia_rec(x, n_mult, prod_ac):
        if n_mult == 0:
            return prod_ac
        else:
            return potencia_rec(x, n_mult - 1, x * prod_ac)

    return potencia_rec(x, n, 1)
```

Novamente, a função `potencia_rec` utiliza recursão de cauda e gera um processo em que existe uma fase de crescimento seguida por uma fase de contração.

Deixamos como exercício a escrita da função `soma_elementos` usando um raciocínio análogo.

As funções que geram processos iterativos e as funções que utilizam a recursão de cauda partilham a característica de utilizarem variáveis de estado. Por esta razão, os processadores de algumas linguagens de programação, por exemplo, o Scheme e o PROLOG, transformam automaticamente funções que utilizam recursão de cauda em funções que geram processos iterativos, sendo comum dizer que este tipo de funções gera processos iterativos.

## 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 a 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, exceto 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 escrever a seguinte função:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
```

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 direita<sup>5</sup>, obtemos a seguinte evolução:

```
fib(5)
| fib(4)
| | fib(3)
| | | fib(2)
| | | | fib(1)
| | | | return 1
| | | | fib(0)
```

---

<sup>5</sup>No caso de uma ordem diferente de avaliação, a “forma” do processo é semelhante.

```

| | | | return 0
| | | return 1
| | | fib(1)
| | | return 1
| | return 2
| | fib(2)
| | | fib(1)
| | | return 1
| | | fib(0)
| | | return 0
| | return 1
| return 3
| fib(3)
| | fib(2)
| | | fib(1)
| | | return 1
| | | fib(0)
| | | return 0
| | return 1
| | fib(1)
| | return 1
| return 2
return 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 em que são criados novos ambientes correspondentes à função `fib`, seguidas por fases de contração em que alguns destes ambientes desaparecem.

Ao contrário do que acontece com o processo recursivo linear, estamos perante a existência de múltiplas fases de crescimento e de contração 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 da 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), exceto para os

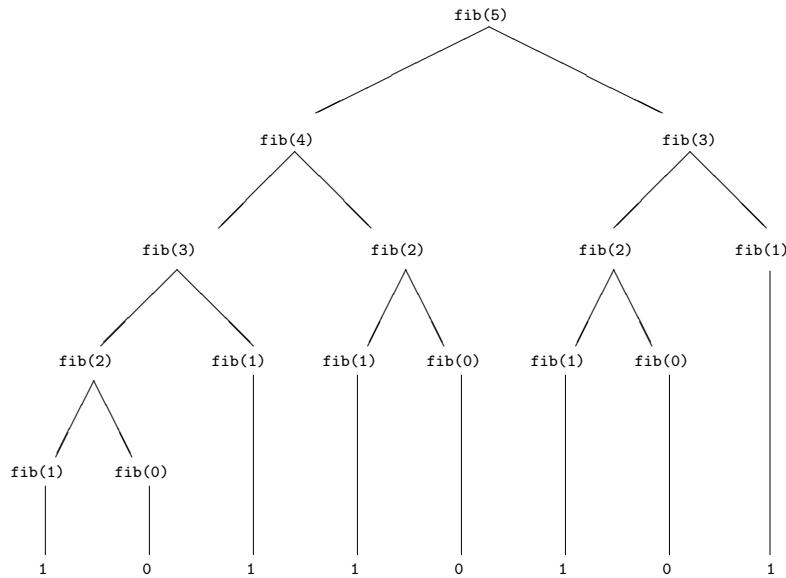


Figura 7.2: Árvore gerada durante o cálculo de `fib(5)`.

dois últimos valores. Na Figura 7.2 apresentamos o resultado da árvore gerada durante o cálculo de `fib(5)`.

Analizando o processo anterior, verificamos que este é muito ineficiente, pois existem 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 usar uma função que utiliza a recursão de cauda. Como cada termo da sequência, exceto os dois primeiros, é a soma dos dois anteriores, para calcular um dos termos 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  $f_{n-1}$  e  $f_{n-2}$ . Neste caso, o próximo número de Fibonacci será dado por  $f_n = f_{n-1} + f_{n-2}$  e a partir de agora os dois últimos termos são  $f_n$  e  $f_{n-1}$ . Podemos agora, usando o mesmo raciocínio, calcular o próximo termo,  $f_{n+1}$ . Para completarmos o nosso processo teremos de explicitar o número de vezes que temos de efetuar 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_1` e `f_n_2`) e o número de operações que ainda temos de efetuar (`cont`).

```
def fib_aux(f_n_1, f_n_2, cont):
    if cont == 0:
        return f_n_2 + f_n_1
    else:
        return fib_aux(f_n_2 + f_n_1, f_n_1, cont - 1)
```

Devemos ainda 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 (1, 0, n - 2)
```

Esta função origina um processo recursivo linear em que não há duplicações de cálculos:

```
fib(5)
fib_aux(1, 0, 3)
| fib_aux(1, 1, 2)
| | fib_aux(2, 1, 1)
| | | fib_aux(3, 2, 0)
| | | return 5
| | return 5
| return 5
return 5
return 5
```

Utilizando um processo iterativo, 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
        for i in range(n):
            f_n_2, f_n_1 = f_n_1, f_n_2 + f_n_1
    return f_n_2
```

Deste exemplo não devemos concluir que a recursão em árvore é um processo inútil. Na próxima secção apresentamos um outro problema para o qual a recursão em árvore corresponde ao método ideal para a sua resolução.

#### 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 respetivos diâmetros, e a finalidade é movimentar todos os discos para um outro poste (tipicamente, o poste da direita), também ordenados por ordem crescente dos respetivos 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 efetuarem movimentos perfeitos à taxa de 1 movimento por segundo, demorarão perto de mil milhões de anos



Figura 7.3: Torre de Hanói com três discos.

para o resolver<sup>6</sup>.

Na Figura 7.3 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 efetuar basicamente três passos:

1. Movimentar  $n - 1$  discos do poste da esquerda para o poste do centro (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 7.4 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:

---

<sup>6</sup>Ver [Raphael, 1976], página 80.

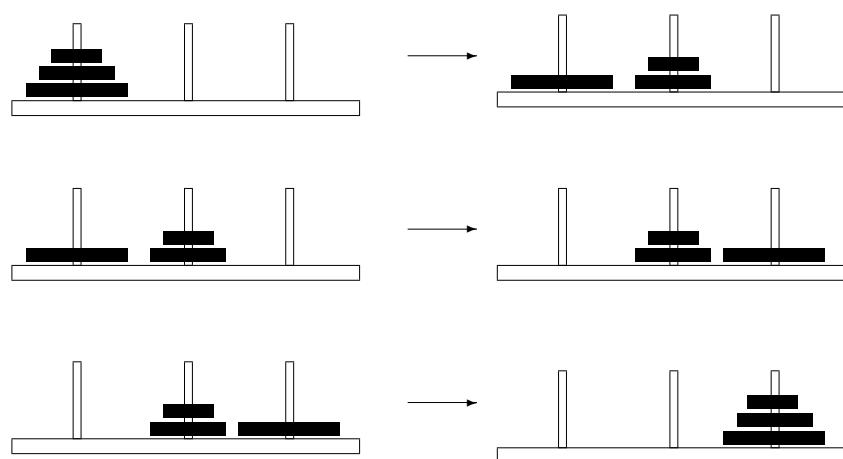


Figura 7.4: Solução da Torre de Hanói com três discos.

```
mov(a(3, 'E', 'D', 'C')
```

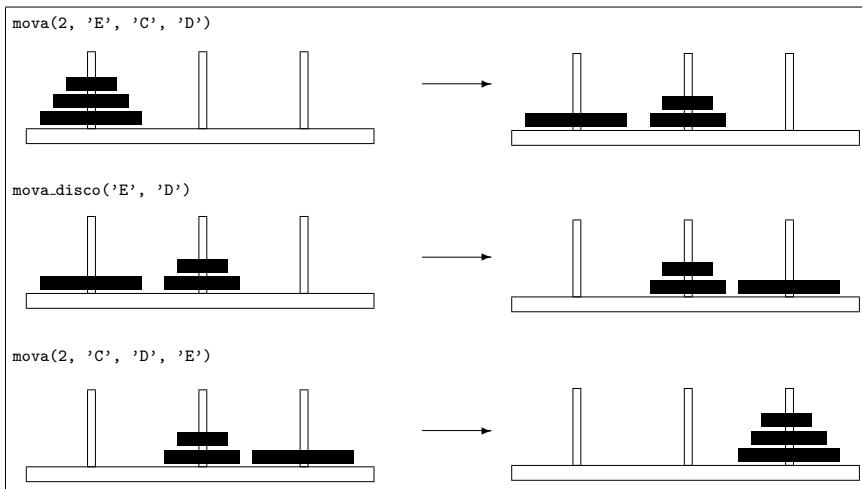


Figura 7.5: Subproblemas gerados por `mov(a(3, 'E', 'D', 'C'))`.

```
def mov(a(n, origem, destino, aux):
    if n == 1:
        mov_a_disc(origem, destino)
    else:
        mov(a(n-1, origem, aux, destino))
        mov_a_disc(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` 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 7.5 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 reflete 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):

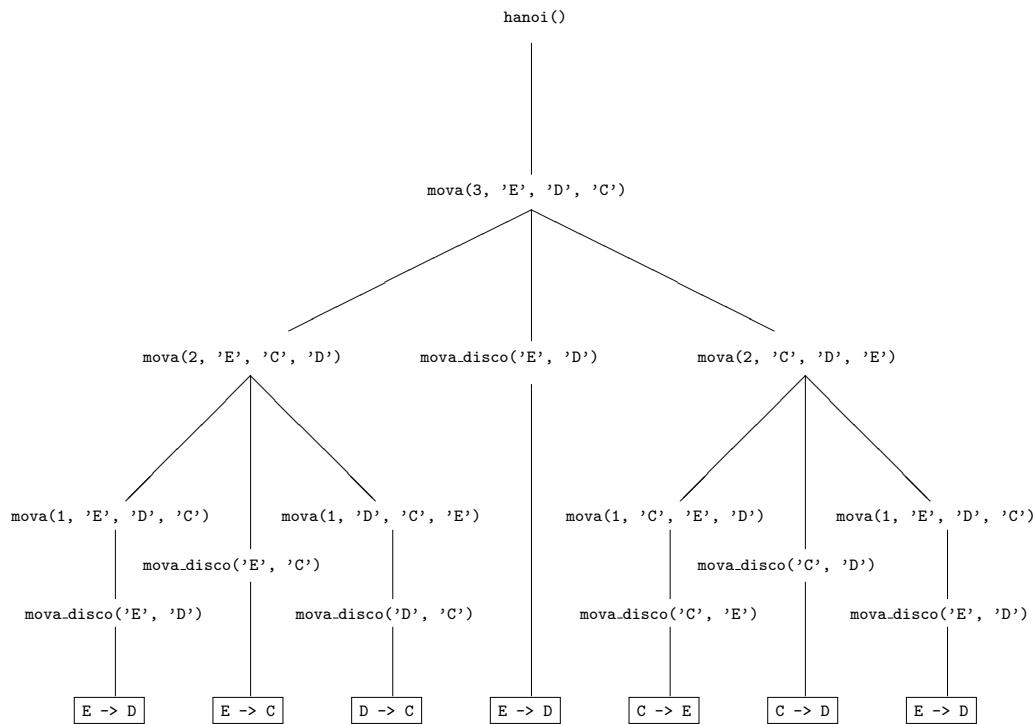
        def move_disco(de, para):
            print(de, '→', para)

        if n == 1:
            move_disco(origem, destino)
        else:
            move(n-1, origem, aux, destino)
            move_disco(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 interação:

```
>>> hanoi()
Quantos discos deseja considerar?
? 3
Solução do puzzle:
E → D
E → C
D → C
E → D
```

Figura 7.6: Árvore gerada por `hanoi()` com três discos`C -> E``C -> D``E -> D`

Na Figura 7.6 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 retângulo 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, mesmo quando calculam a mesma função matemática. 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 consideramos 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.

Dos exemplos apresentados neste capítulo, podemos concluir que os processos recursivos lineares têm ordem de crescimento  $O(n)$ , quer para o espaço, quer para o tempo, e que os processos iterativos lineares têm ordem de crescimento  $O(1)$  para o espaço e  $O(n)$  para o tempo. Os processos recursivos em árvore têm ordem de crescimento  $O(n)$  para o espaço e  $O(k^n)$  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:

$$x^n = \begin{cases} x & \text{se } n = 1 \\ x \cdot (x^{n-1}) & \text{se } n \text{ for ímpar} \\ (x^{n/2})^2 & \text{se } n \text{ for par} \end{cases}$$

o que nos leva a escrever a seguinte função para o cálculo da potência:

```
def potencia_rapida(x, n):
    if n == 0:
        return 1
    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  $x^{2n}$  precisamos apenas de mais uma multiplicação do que as que são necessárias para calcular  $x^n$ . Geramos assim um processo cuja ordem temporal e espacial 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.

Podemos definir a seguinte versão da função `potencia_rapida` que gera um processo iterativo<sup>7</sup>:

```
def potencia_rapida(x, n):
    res = 1
    while n != 0:
        if impar(n):
            res = res * x
            n = n - 1
        else:
            x = x * x
            n = n // 2
    return res
```

## 7.6 Notas finais

Neste capítulo apresentámos as motivações para estudar os processos gerados por funções e caracterizámos alguns destes processos, nomeadamente os processos recursivos lineares, iterativos lineares e recursivos em árvore.

---

<sup>7</sup>Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta função.

## 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) Escreva uma função equivalente que utilize recursão de cauda.

2. Considere a função de Ackermann<sup>8</sup>:

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

- (a) Escreva em Python uma função para calcular o valor da função de Ackermann.
- (b) Siga o processo gerado pelo cálculo de  $A(2, 2)$ .

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) Origina operações adiadas? Justifique.

---

<sup>8</sup>Tal como apresentada em [Machtey and Young, 1978], página 24.

- (c) Se a sua função gerar operações adiadas, escreva uma função equivalente utilizando recursão de cauda; se a sua função utilizar recursão de cauda, escreva uma função equivalente que gera operações adiadas.
4. Considere a função  $g$ , definida para inteiros não negativos do seguinte modo:

$$g(n) = \begin{cases} 0 & \text{se } n = 0 \\ n - g(g(n-1)) & \text{se } n > 0 \end{cases}$$

Escreva uma função recursiva em Python para calcular o valor de  $g(n)$ .

5. O número de combinações de  $m$  objetos  $n$  a  $n$  pode ser dado pela seguinte fórmula:

$$C(m, n) = \begin{cases} 1 & \text{se } n = 0 \\ 1 & \text{se } m = n \\ 0 & \text{se } m < n \\ C(m-1, n) + C(m-1, n-1) & \text{se } m > n, m > 0 \text{ e } n > 0 \end{cases}$$

Escreva uma função em Python para calcular o número de combinações de  $m$  objetos  $n$  a  $n$ .



# Capítulo 8

## 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*

### 8.1 O tipo dicionário

Um *dicionário*, em Python designado por `dict`<sup>1</sup>, é um tipo mutável constituído por 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 direta 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. Em programação, o tipo correspondente a um dicionário é normalmente conhecido como *lista associativa* ou *tabela de dispersão*<sup>2</sup>.

---

<sup>1</sup>Do inglês, “dictionary”.

<sup>2</sup>Do inglês, “hash table”.

Notemos que ao passo que as estruturas de dados que considerámos 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'<sup>3</sup>.

Como os dicionários são *conjuntos*, quando o Python mostra um dicionário ao utilizador, os seus elementos podem não aparecer pela mesma ordem pela qual

---

<sup>3</sup>Na 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].

foram escritos, como mostra a seguinte interação:

```
>>> {'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 105. 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 interação:

```
>>> 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 8.1 como se ilustra na interação seguinte.

```
>>> ex_dic
{'a': 4, 'c': 4, 'b': 2, 'd': 1}
>>> 'f' in ex_dic
```

<i>Operação</i>	<i>Tipo dos argumentos</i>	<i>Valor</i>
<code>del(d[e])</code>	Elemento de dicionário	Remove do dicionário $d$ o elemento com índice $e$ .
<code>c in d</code>	Chave e dicionário	<code>True</code> se a chave $c$ pertence ao dicionário $d$ ; <code>False</code> em caso contrário.
<code>c not in d</code>	Chave e dicionário	A negação do resultado da operação $c \text{ in } d$ .
<code>len(d)</code>	Dicionário	O número de elementos do dicionário $d$ .

Tabela 8.1: Operações sobre dicionários em Python.

```

False
>>> 'a' in ex_dic
True
>>> len(ex_dic)
4
>>> del(ex_dic['c'])
>>> ex_dic
{'b': 2, 'a': 4, 'd': 1}

```

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 interação:

```

>>> 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

```

## 8.2 Contagem de letras

Suponhamos que queremos escrever uma função que recebe uma cadeia de carateres e que determina o número de vezes que cada caráter aparece na cadeia. Esta tarefa é realizada pela função `conta_carateres` que recebe uma cadeia de carateres e devolve um dicionário cujas chaves são carateres, estando cada caráter associado ao número de vezes que este aparece na cadeia.

```
def conta_carateres(cc):
    cont = {}
    for c in cc: # o dicionário é atualizado
        if c not in cont:
            cont[c] = 1
        else:
            cont[c] = cont[c] + 1
    return cont
```

Suponhamos que `Lusiadas` é uma cadeia de carateres contendo o texto completo de *Os Lusíadas*, incluindo os saltos de linha. Por exemplo, para os primeiros 60 carateres desta cadeia, obtemos:

```
>>> Lusiadas[0:60]
'As armas e os barões assinalados,\nQue da ocidental praia Lus'
```

Fornecendo à função `conta_carateres` a cadeia `Lusiadas`, verificamos que o dicionário criado inclui chaves associadas ao caráter de salto de linha (`\n`), ao espaço em branco e a todos os símbolos de pontuação:

```
>>> nc = conta_carateres(Lusiadas)
>>> nc['a']
29833
>>> nc['\n']
9942
>>> nc[' ']
50449
>>> nc['.']
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 ignorar<sup>4</sup>:

```
pontuacao = ['!', '"', "'", '(', ')', '–', ',', '.', \
';', ':', '?', '[', ']', '{', '}', ' ', '\n']
```

Usando esta lista, podemos alterar o ciclo `for` da nossa função do seguinte modo:

```
def conta_letras(cc):
    pontuacao = ['!', '"', "'", '(', ')', '–', ',', '.', \
';', ':', '?', '[', ']', '{', '}', ' ', '\n']
    cont = {}
    for c in cc:
        if c not in pontuacao:
            # o dicionário é atualizado
            if c not in cont:
                cont[c] = 1
            else:
                cont[c] = cont[c] + 1
    return cont
```

Podendo obter a interação:

```
>>> nc = conta_letras(Lusiadas)
>>> nc['a']
29833
>>> '.' in nc
False
>>> nc['A']
1222
```

A nossa função conta separadamente as letras minúsculas e as letra maiúsculas que aparecem na cadeia, como o mostra a interação anterior. Se desejarmos não fazer a distinção entre letras maiúsculas e minúsculas, podemos pensar em

---

<sup>4</sup>Note-se a utilização de dois delimitadores para cadeias de caracteres, uma contendo as aspas e a outra a plica.

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 da 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, à qual vamos recorrer, corresponde a utilizar uma função embutida do Python, a qual para uma dada cadeia de letras, 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()`<sup>5</sup>. Por exemplo:

```
>>> ex_cadeia = 'Exemplo DE CONVERSÃO'
>>> ex_cadeia.lower()
'exemplo de conversão'
```

Podemos agora escrever a seguinte função que conta o número de letras existentes na cadeia de caracteres que lhe é fornecida como argumento, não distinguindo as maiúsculas das minúsculas:

```
def conta_letras(cc):
    pontuacao = ['!', '"', "'", '(', ')', '‐', ',', '.', \
                 ';', ':', '?', '[', ']', '{', '}', ' ', '\n']
    cont = {}
    for c in cc.lower(): # transformação em minúsculas
        if c not in pontuacao:
            # o dicionário é atualizado
            if c not in cont:
                cont[c] = 1
            else:
                cont[c] = cont[c] + 1
    return cont
```

---

<sup>5</sup>O nome desta função segue a sintaxe de `(nome composto)` apresentada na página 97, mas, na realidade, esta corresponde a um tipo de entidade diferente. Este aspecto é abordado na Secção 11.5. Para os efeitos deste capítulo, iremos considerá-la como uma simples função.

Obtendo a interação:

```
>>> nc = conta_letras(Lusiadas)
>>> nc
{'è': 2, 'ã': 1619, 'é': 823, 'ü': 14, 'ê': 378, 'â': 79, 'ç': 863,
'õ': 115, 'í': 520, 'h': 2583, 'i': 12542, 'j': 1023, 'ó': 660,
'l': 6098, 'm': 10912, 'n': 13449, 'o': 27240, 'à': 158,
'a': 31055, 'b': 2392, 'c': 7070, 'd': 12304, 'e': 31581,
'f': 3055, 'g': 3598, 'x': 370, 'y': 7, 'z': 919, 'ô': 52,
'ú': 163, 'á': 1377, 'p': 5554, 'q': 4110, 'r': 16827, 's': 20644,
't': 11929, 'u': 10617, 'v': 4259, 'ò': 1}
```

Esta interação mostra o que já foi referido na página 237, a ordem por que são mostrados os elementos de um dicionário é escolhida pelo Python, pois um dicionário corresponde a um conjunto. No caso da contagem de caracteres, estamos à espera que os resultados sejam ordenados por algum critério, por exemplo, pela sua ordem alfabética. Para obter esse efeito, escrevemos a função `mostra_ordenados`, a qual utiliza a última versão da função `conta_letras` e uma das funções de ordenação, `ordena`, apresentadas na Secção 5.5:

```
def mostra_ordenados(cc):
    res_conta = conta_letras(cc)

    # constroi uma lista de índices e ordena-a
    lista_ind = []
    for c in res_conta:
        lista_ind = lista_ind + [c]
    ordena(lista_ind)

    for c in lista_ind:
        print(c, res_conta[c])
```

Depois da contagem de letras (cujos valores estão guardados no dicionário associado à variável `res_conta`), esta função cria uma lista de todas as letras que existem como chave no resultado da contagem, ordena essa lista e usa essa ordenação para mostrar o número de ocorrências de uma dada letra. A interação seguinte mostra parcialmente o resultado produzido pela função `mostra_ordenados`:

```
>>> mostra_ordenados(Lusiadas)
a 31055
b 2392
c 7070
d 12304
e 31581
...
```

Em muitas aplicações que efetuam contagem de letras, é mais importante a frequência de ocorrência de cada letra do que o valor absoluto da sua ocorrência. Para abordar este tipo de aplicações, vamos escrever a função, `mostra_freq_ordenadas`, que após a contagem das letras na cadeia de carateres que é seu argumento, cria um dicionário, `freqs`, no qual cada índice (letra existente na cadeia de carateres) é associado à sua frequência relativa. Após o cálculo das frequências, a função escreve as letras existentes na cadeia de carateres, ordenadas por ordem decrescente da sua frequência.

Esta última tarefa necessita de alguma explicação adicional. Começamos por criar duas listas paralelas (ver a definição de listas paralelas na página 152) contendo as letras, `lista.ind`, e as respetivas frequências, `lista.freqs`. Recorrendo à ordenação por borbulhamento, a função `ordena_paralelas` ordena estas listas por ordem decrescente das frequências. Finalmente, as listas de índices e frequências, agora ordenadas, são escritas. Como exercício, o leitor deverá compreender o funcionamento da função `ordena_paralelas`.

```
def mostra_freq_ordenadas(cc):
    res_conta = conta_letras(cc)

    # calcula as frequencias
    total_c = 0 # número total de carateres
    for c in res_conta:
        total_c = total_c + res_conta[c]
    freqs = {} # dicionário com frequências
    for c in res_conta:
        freqs[c] = res_conta[c] / total_c

    # constroi lista de índices e freqs e ordena-a
    lista_ind = []
```

```

lista_freqs = []
for c in freqs:
    lista_ind = lista_ind + [c]
    lista_freqs = lista_freqs + [freqs[c]]
ordena_paralelas(lista_freqs, lista_ind)

for i in range(len(lista_ind)):
    print(lista_ind[i], lista_freqs[i])

def ordena_paralelas(lst1, lst2):

    maior_indice = len(lst1) - 1
    nenhuma_troca = False # garante que o ciclo while é executado

    while not nenhuma_troca:
        nenhuma_troca = True
        for i in range(maior_indice):
            if lst1[i] < lst1[i+1]:
                lst1[i], lst1[i+1] = lst1[i+1], lst1[i]
                lst2[i], lst2[i+1] = lst2[i+1], lst2[i]
                nenhuma_troca = False
        maior_indice = maior_indice - 1

```

Com as funções anteriores, podemos gerar a interação:

```

>>> mostra_freq_ordenadas(Lusiadas)
e 0.1278779731294693
a 0.12574809079939425
o 0.1103003700974239
s 0.08359180764652052
r 0.06813598853264875
n 0.054457770831140014
...

```

### 8.3 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 letivo 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 letivo 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 reprovação 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 letivo seguinte. Para representar o registo das classificações obtidas numa dada disciplina, iremos utilizar um dicionário em que cada par contém o ano letivo 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, respetivamente 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 letivos.

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 interação:

```
>>> 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 registo académicos ao lançar as notas de uma dada disciplina. O professor responsável da disciplina fornece à secretaria a pauta da disciplina, uma lista contendo como primeiro elemento a identificação da disciplina, o segundo elemento desta lista contém a indicação do ano letivo ao qual as notas correspondem, seguido por uma lista de pares, em que cada par contém o número de um aluno e a nota respetiva. Por exemplo, a seguinte lista poderá corresponder às notas obtidas em FP no ano letivo de 2014/15:

```
[‘FP’, ‘2014/15’, [(12345, 12), (12346, ‘REP’), (12347, 10),
(12348, 14), (12349, ‘REP’), (12350, 16), (12351, 14)]]
```

A função `introduz_notas` recebe a pauta de uma disciplina e o dicionário contendo a informação sobre os alunos e efetua o lançamento das notas de uma determinada disciplina, num dado ano letivo. O dicionário contendo a informação sobre os alunos é alterado por esta função.

```
def introduz_notas (pauta, registos):

    disc = pauta[0]
    ano_letivo = pauta[1]

    for num_nota in pauta[2]:
        num, nota = num_nota[0], num_nota[1]
        if num not in registos:
            raise ValueError ('O aluno ' + str(num) + ' não existe')
        else:
            registos[num] [‘disc’] [disc] [ano_letivo] = nota
    return registos
```

## 8.4 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 repre-

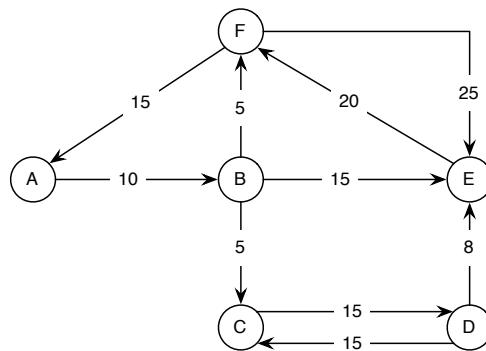


Figura 8.1: Exemplo de um grafo.

sentação de mapas 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 *dirigidos*, 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 8.1, 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, 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 8.1, 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

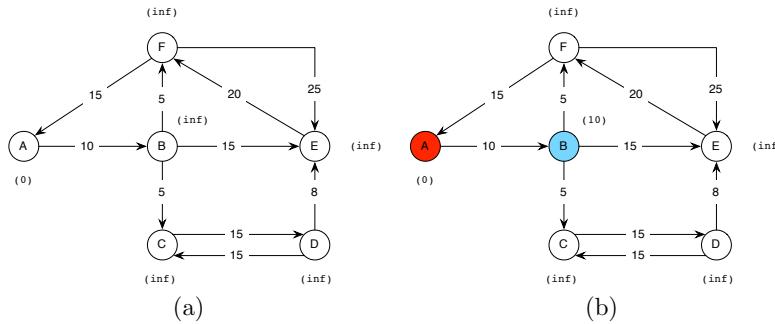


Figura 8.2: Dois primeiros passos no algoritmo de Dijkstra.

Dijkstra<sup>6</sup> (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 sua 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 8.1, 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 8.2 (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 da lista de nós, o nó com menor valor da distância calculada.  
Se existir mais do que um nó nesta situação, escolhe arbitrariamente um deles;
2. Remove o nó escolhido da lista de nós;

<sup>6</sup>[Dijkstra, 1959].

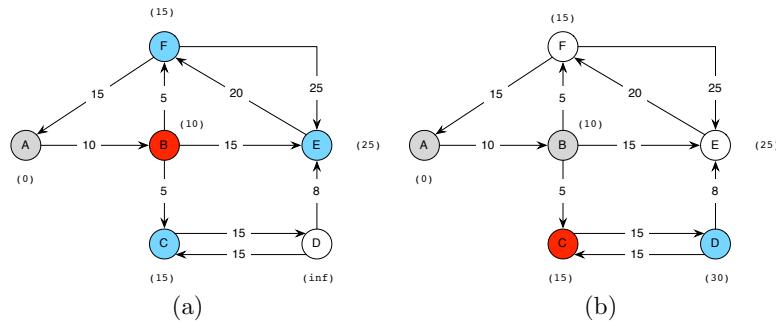


Figura 8.3: Terceiro e quarto passos no algoritmo de Dijkstra.

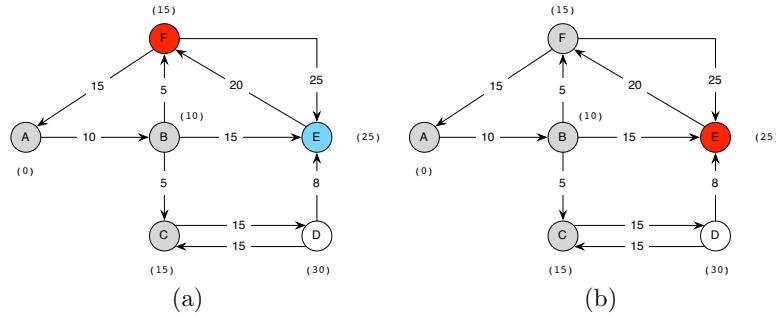


Figura 8.4: Quinto e sexto passos no algoritmo de Dijkstra.

3. Atualiza a distância calculada dos nós adjacentes ao nó escolhido (os nós a que o nó escolhido está diretamente ligado). Esta distância só é atualizada 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 8.2 (b), mostramos o passo correspondente a selecionar o primeiro nó, o nó A, atualizando-se a distância calculada para o nó B. Note-se que A não está ligado a F, mas F está ligado a A. Seleciona-se de seguida o nó B, atualizando-se as distâncias calculadas dos nós C, E e F (Figura 8.3 (a)). Existem agora dois nós com a distância calculada de 15, seleciona-se arbitrariamente o nó C e atualiza-se a distância calculada associada ao nó D (Figura 8.3 (b)). Escolhe-se

de seguida o nó F, o qual não leva à atualização de nenhuma distância calculada (Figura 8.4 (a)). O nó E é então escolhido, não levando também à atualização das distâncias calculadas (Figura 8.4 (b)). Finalmente, o nó D é escolhido, não originando a atualização 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*<sup>7</sup>, 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 respetivas distâncias. No nosso exemplo, o nó B está ligado aos nós C, E e F, respetivamente, 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 8.1, 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)]}
```

A função em Python que corresponde ao algoritmo da distância mínima é:

```
def cdm(g, origem):
    nos = nos_do_grafo(g)
    dc = dist_inicial(nos, origem)
    while nos != []:
        n = dist_min(nos, dc) # seleciona o nó com menor distância
        remove(nos, n)
        for nn in g[n]: # para os nós a que nn está ligado
            atualiza_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 238.

```
def nos_do_grafo(g):
    res = []
```

---

<sup>7</sup>Do inglês, “adjacency list”.

```

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 249.

```

def dist_inicial (nos, origem):
    res = {}
    for n in nos:
        res[n] = 'inf'
    res[origem] = 0
    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 `atualiza_dist` recebe um nó de partida (`no_partida`), um nó a

que o `no_partida` está diretamente ligado (`no_chegada`), a distância entre o `no_partida` e o `no_chegada` e as distâncias calculadas (`dc`) e atualiza, nas distâncias calculadas, a distância associada ao `no_chegada`.

```
def atualiza_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 8.1, podemos originar a seguinte interação:

```
>>> 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}
```

## 8.5 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 respetiva chave. Um dos aspetos 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 and Walker, 1996]. Algoritmos que manipulam grafos podem ser consultados em [McConnell, 2008] e em [Cormen et al., 2009].

## 8.6 Exercícios

1. Considere a seguinte atribuição:

```
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 0BD')]}}}
```

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. Considere a seguinte lista de dicionários na qual os significados dos campos são óbvios:

```
l_nomes = [ {'nome': {'nomep': 'Jose', 'apelido': 'Silva'},  
            'morada': {'rua': 'R. dos douradores', 'num': 34, 'andar': '6 Esq',  
                        'localidade': 'Lisboa', 'estado': '', 'cp': '1100-032',  
                        'pais': 'Portugal'}}, { 'nome': {'nomep': 'John', 'apelido': 'Doe'},  
            'morada': {'rua': 'West Hazeltine Ave.', 'num': 57, 'andar': '',  
                        'localidade': 'Kenmore', 'estado': 'NY', 'cp': '14317', 'pais': 'USA'}}]
```

Diga quais são os valores dos seguintes nomes:

- (a) `l_nomes[1]`
- (b) `l_nomes[1]['nome']`
- (c) `l_nomes[1]['nome'][apelido]`
- (d) `l_nomes[1]['nome'][apelido][0]`

3. Uma carta de jogar é caracterizada 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). Uma carta pode ser representada por um dicionário com duas chaves, '`np`' e '`vlr`', sendo um conjunto de cartas representado por uma lista de cartas.

- (a) Escreva uma função em Python que devolve uma lista contendo todas as cartas de um baralho. Por exemplo,

```
>>> baralho()  
[{'np': 'esp', 'vlr': 'A'}, {'np': 'esp', 'vlr': '2'},  
 {'np': 'esp', 'vlr': '3'}, {'np': 'esp', 'vlr': '4'},  
 {'np': 'esp', 'vlr': '5'}, {'np': 'esp', 'vlr': '6'},  
 {'np': 'esp', 'vlr': '7'}, {'np': 'esp', 'vlr': '8'},  
 {'np': 'esp', 'vlr': '9'}, {'np': 'esp', 'vlr': '10'},  
 {'np': 'esp', 'vlr': 'J'}, {'np': 'esp', 'vlr': 'Q'},  
 {'np': 'esp', 'vlr': 'K'}, {'np': 'copas', 'vlr': 'A'},  
 {'np': 'copas', 'vlr': '2'}, {'np': 'copas', 'vlr': '3'},  
 ... ]
```

- (b) Recorrendo à função `random()`, a qual produz um número aleatório no intervalo  $[0, 1[$ , escreva a 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. Por exemplo,

```
>>> baralha(baralho())
[{'np': 'esp', 'vlr': '3'}, {'np': 'esp', 'vlr': '9'},
 {'np': 'copas', 'vlr': '6'}, {'np': 'esp', 'vlr': 'Q'},
 {'np': 'esp', 'vlr': '7'}, {'np': 'copas', 'vlr': '8'},
 {'np': 'copas', 'vlr': 'J'}, {'np': 'esp', 'vlr': 'K'},
 ... ]
```

- (c) Escreva uma função em Python que recebe um baralho de cartas e as distribui por quatro jogadores, devolvendo uma lista que contém as cartas de cada jogador. Por exemplo,

```
distribui(baralha(baralho()))
[[{'np': 'ouros', 'vlr': 'A'}, {'np': 'copas', 'vlr': '7'},
 {'np': 'paus', 'vlr': 'A'}, {'np': 'esp', 'vlr': 'J'},
 {'np': 'paus', 'vlr': '6'}, {'np': 'esp', 'vlr': '10'},
 {'np': 'copas', 'vlr': '5'}, {'np': 'esp', 'vlr': '6'},
 {'np': 'copas', 'vlr': '8'}, {'np': 'esp', 'vlr': '3'},
 {'np': 'ouros', 'vlr': '5'}, {'np': 'ouros', 'vlr': '8'},
 {'np': 'copas', 'vlr': 'K'}],
[{'np': 'paus', 'vlr': '2'}, {'np': 'esp', 'vlr': '2'},
 ...]]
```

4. Escreva uma função que recebe uma cadeia de caracteres correspondente a um texto e que produz uma lista de todas as palavras que este contém, juntamente com o número de vezes que essa palavra aparece no texto. SUGESTÃO: guarde cada palavra como uma entrada num dicionário contendo o número de vezes que esta apareceu no texto. por exemplo,

```
>>> cc = 'a aranha arranha a ra a ra arranha a aranha' \
+ 'nem a aranha arranha a ra nem a ra arranha a aranha'
>>> conta_palavras(cc)
{'aranha': 4, 'arranha': 4, 'ra': 4, 'a': 8, 'nem': 2}
```

5. Usando a ordenação por borbulhamento, escreva a função `mostra_ordenado` que apresenta por ordem alfabética os resultados produzidos pelo exercício anterior. Por exemplo,

```
>>> mostra_ordenado(conta_palavras(cc))
a 8
aranha 4
arranha 4
nem 2
ra 4
```

6. Suponha que o índice de um livro é representado por um dicionário em que os títulos dos capítulos são as chaves e os seus valores correspondem a uma lista contendo a primeira e a última página do capítulo. Suponha também que o índice remissivo é representado por um dicionário em que cada palavra está associada às páginas em que aparece (representadas por uma lista). Escreva um programa em Python que recebe um índice e um índice remissivo e que devolve um dicionário em que cada palavra no índice remissivo é associada aos títulos dos capítulos em que aparece. Nota: uma palavra não deve ser associada duas vezes ao mesmo capítulo. Por exemplo, sendo

```
ind = {'c1': [1, 34], 'c2': [35, 42], 'c3': [43, 52],
       'c4': [53, 60]}
i_r = {'p1': [2, 54], 'p2': [36, 37, 50], 'p3': [40]}
```

então

```
>>> caps(ind, i_r)
{'p3': ['c2'], 'p2': ['c2', 'c3'], 'p1': ['c1', 'c4']}
```

7. Suponha que `bib` é uma lista cujos elementos são dicionários e que contém a informação sobre os livros existentes numa biblioteca. Cada livro é caracterizado pelo seus autores, título, casa editora, cidade de publicação, ano de publicação, número de páginas e ISBN. Por exemplo, a seguinte lista corresponde a uma biblioteca com dois livros:

```
[{'autores': ['G. Arroz', 'J. Monteiro', 'A. Oliveira'],
  'titulo': 'Arquitectura de computadores', 'editor': 'IST Press',
  'cidade': 'Lisboa', 'ano': 2007, 'numpags': 799,
  'isbn': '978-972-8469-54-2'}, {'autores': ['J.P. Martins'],
  'titulo': 'Logica e Raciocinio', 'editor': 'College Publications',
  'cidade': 'Londres', 'ano': 2014, 'numpags': 438,
  'isbn': '978-1-84890-125-4'}]
```

- (a) Escreva um programa em Python que recebe a informação de uma biblioteca e devolve o número médio de páginas dos livros da biblioteca. Por exemplo:

```
>>> media_pags(bib)
618.5
```

- (b) Escreva um programa em Python que recebe a informação de uma biblioteca e devolve o título do livro mais antigo. Por exemplo:

```
>>> mais_antigo(bib)
'Arquitectura de computadores'
```

8. Uma matriz é dita *esparsa* (ou rarefeita) quando a maior parte dos seus elementos é zero. As matrizes esparsas aparecem em grande número de aplicações em engenharia. Uma matriz esparsa pode ser representada por um dicionário cujas chaves correspondem a tuplos que indicam a posição de um elemento na matriz (linha e coluna) e cujo valor é o elemento nessa posição da matriz. Por exemplo,  $\{(3, 2): 20, (150, 2): 6, (300, 10): 20\}$  corresponde a uma matriz esparsa com apenas três elementos diferentes de zero.

- (a) Escreva uma função em Python que recebe uma matriz esparsa e a escreve sob a forma, na qual os elementos cujo valor é zero são explicitados.

$$\begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}$$

```
>>> escreve_esparsa({(1,5): 4, (2, 3): 9, (4, 1): 1})
0 0 0 0 0
0 0 0 0 0 4
0 0 0 9 0 0
0 0 0 0 0 0
0 1 0 0 0 0
```

- (b) Escreva uma função em Python que recebe duas matrizes esparsa e devolve a sua soma. Por exemplo,

```
e1 = {(1,5): 4, (2, 3): 9, (4, 1): 1}
```

```
e2 = {(1, 6): 2, (4, 1): 2, (5, 4): 2}
>>> escreve_esparsa(soma_esparsa(e1, e2))
0 0 0 0 0 0 0
0 0 0 0 0 4 2
0 0 0 9 0 0 0
0 0 0 0 0 0 0
0 3 0 0 0 0 0
0 0 0 0 2 0 0
```

9. Escreva uma função em Python que recebe um dicionário cujos valores associados às chaves correspondem a listas de inteiros e que devolve o dicionário que se obtém “invertendo” o dicionário recebido, no qual as chaves são os inteiros que correspondem aos valores do dicionário original e os valores são as chaves do dicionário original às quais os valores estão associados. Por exemplo,

```
>>> inverte_dic({'a': [1, 2], 'b': [1, 5], 'c': [9], 'd': [4]})
```

```
{1: ['a', 'b'], 2: ['a'], 4: ['d'], 5: ['b'], 9: ['c']}
```

10. Um tabuleiro de xadrez tem 64 posições organizadas em 8 linhas e 8 colunas. As linhas são numeradas de 1 a 8 e as colunas de A a H, sendo a posição inferior esquerda a 1, A. Neste tabuleiro são colocadas peças de duas cores (brancas e pretas) e de diferentes tipos (rei, rainha, bispo, torre, cavalo e peão). Um tabuleiro de um jogo de xadrez pode ser representado por um dicionário cujos elementos são do tipo  $(l, c): (cor, t)$ . Por exemplo o elemento do dicionário  $(2, C): (\text{branca}, \text{rainha})$  indica que a rainha branca está na segunda linha, terceira coluna.

A situação do jogo de xadrez apresentado na Figura 8.5 é representada pelo seguinte dicionário:

```
j = {(1, 'H'): ('branca', 'torre'), (2, 'F'): ('branca', 'peao'), \
      (2, 'G'): ('branca', 'rei'), (6, 'F'): ('branca', 'bispo'), \
      (5, 'C'): ('branca', 'rainha'), (6, 'G'): ('preta', 'peao'), \
      (7, 'F'): ('preta', 'peao'), (8, 'F'): ('preta', 'torre'), \
      (8, 'G'): ('preta', 'rei'), (2, 'C'): ('preta', 'peao')}
```

Num jogo de xadrez, a rainha movimenta-se na vertical, na horizontal ou nas diagonais e pode atacar qualquer peça da cor contrária que possa ser atingida num dos seus movimentos, desde que não existam outras peças

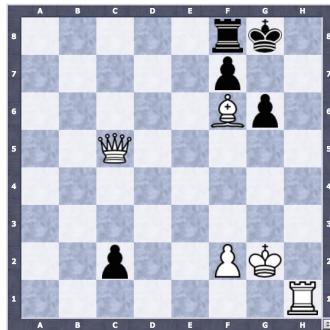


Figura 8.5: Tabuleiro de xadrez.

no caminho. Escreva uma função em Python que recebe um tabuleiro de xadrez e determina quais as peças que podem ser atacadas pelas rainhas. Por exemplo,

```
>>> ataques_rainhas(j)
[['peao', (2, 'C')], ['torre', (8, 'F')]]
```

11. Tendo em conta a representação de um tabuleiro de xadrez apresentada no exercício anterior, escreva um programa que recebe a posição de um cavalo e determina os seus movimentos possíveis. Um cavalo move-se andando duas posições para a frente e uma para o lado. Note-se que um cavalo pode mover-se para uma casa onde exista uma peça do adversário mas não se pode mover para uma casa em que exista uma peça com a sua cor.

# Capítulo 9

## Abstração 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 abstração procedural e temos utilizado a abstração procedural na definição de novas funções. Vimos também que quase qualquer programa que efetue uma tarefa que não seja trivial necessita de manipular entidades computacionais (dados) que correspondam a tipos estruturados de dados. Os programas que temos desenvolvido manipulam diretamente instâncias dos tipos existentes em Python, sem criarem novos tipos de dados.

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 dados que não estejam embutidos na nossa linguagem de programação, introduzindo o conceito

de abstração de dados.

## 9.1 A abstração em programação

Uma *abstração* é uma descrição ou uma especificação simplificada de uma entidade que dá ênfase a certas propriedades dessa entidade e ignora outras. Uma boa abstração especifica 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 abstração” está sempre associado a uma utilização particular da entidade.

Como exemplo, consideremos a informação relativa a um grupo de pessoas. Cada pessoa é caracterizada por um grande número de atributos, entre os quais podemos citar o nome, a morada, o sexo, o estado civil, a data e o local de nascimento, a cor dos olhos e do cabelo, o peso, a altura, a profissão, o salário e o número de contribuinte. Notemos, desde já, que quando falamos de atributos como a cor dos olhos ou a cor do cabelo estamos já a utilizar uma abstração. A íris, a parte do olho que caracteriza a cor, não é uniforme, apresenta tonalidades da mesma cor ou mesmo uma mistura de cores diferentes (por exemplo, verde e castanho). Ao falarmos da cor dos olhos estamos a abstrair, a ignorar, todas estas variações e a referir-nos apenas à cor dominante.

Suponhamos agora que este grupo de pessoas estava a ser considerado por um programa que calculava o valor do IRS a pagar. Neste caso, a cor dos olhos e do cabelo são totalmente irrelevantes. Nesta aplicação, uma boa abstração deveria considerar atributos como o número de contribuinte, o salário, o estado civil, e ignorar atributos como o sexo e a cor dos olhos. Por outro lado, se estivéssemos a desenvolver um programa para uma agência matrimonial, os atributos sexo, cor dos olhos e cor do cabelo seriam de importância fundamental, ao passo que o número de contribuinte seria irrelevante. Em resumo, as propriedades que consideramos, ou abstraímos, da entidade em causa, neste caso, uma pessoa, dependem da aplicação em vista.

A abstração é um conceito essencial em programação. De facto, a atividade de programação pode ser considerada como a construção de abstrações que podem ser executadas por um computador.

Até aqui temos usado a abstração procedural para a criação de funções.

Usando a abstração 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, utilizando outro algoritmo, sem que os programas que utilizam essa função sejam afetados. A abstração procedural permite separar o modo como uma função é utilizada do modo como essa função é realizada.

O conceito equivalente à abstração procedural para dados (ou estruturas de dados) tem o nome de abstração de dados. A *abstração de dados* é uma metodologia que permite separar o modo como uma estrutura de dados é utilizado dos pormenores relacionados com o modo como essa estrutura de dados é construída a partir de outras estruturas de dados. Quando utilizamos um tipo de dados 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 manipular listas. Se numa versão posterior do Python, a representação interna das listas for alterada, os nossos programas não são afetados por essa alteração. A abstração de dados permite a obtenção de um comportamento semelhante para os dados criados no nosso programa.

Analogamente ao que acontece quando recorremos à abstração procedural, com a abstração 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 abstrata.

Recordemos que um *tipo de dados* é caracterizado por um *conjunto de entidades* 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*. Por exemplo, o tipo de dados correspondente aos inteiros é constituído pelas constantes inteiras, tal como apresentadas no Capítulo 2 e por um conjunto de operações que inclui operações que transformam inteiros em inteiros (por exemplo, `+`, `*` e `//`) e operações que comparam inteiros, fornecendo resultados do tipo lógico (por exemplo, `=`, `>`, `>=`).

Neste capítulo abordamos o desenvolvimento de entidades que correspondem a dados. Apresentamos uma metodologia para desenvolver e utilizar estas entidades que tem o nome de tipos abstratos de dados.

## 9.2 Motivação: números complexos

A abstração de dados consiste em considerar que a definição de novos tipos de dados 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 abstração 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 plano<sup>1</sup>. Sabemos também que a soma, subtração, multiplicação e divisão de números complexos são definidas do seguinte modo:

$$(a + bi) + (c + di) = (a + c) + (b + d)i$$

$$(a + bi) - (c + di) = (a - c) + (b - d)i$$

$$(a + bi).(c + di) = (ac - bd) + (ad + bc)i$$

$$\frac{(a + bi)}{(c + di)} = \frac{ac + bd}{c^2 + d^2} + \frac{bc - ad}{c^2 + d^2}i$$

---

<sup>1</sup>Chamado *plano de Argand*, em honra ao matemático francês Jean-Robert Argand (1768–1822) que o introduziu em 1806.

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 dados 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  $a + bi$  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 correta, ela introduz uma complexidade desnecessária no nosso raciocínio. Em qualquer operação que efetuemos 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 efetuam operações aritméticas sobre números complexos, embora não saibamos como estes são representados. Assim, podemos escrever as seguintes funções que, respetivamente, 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 exterior<sup>2</sup> 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).

---

<sup>2</sup>Por “mundo exterior” entenda-se o mundo com que o Python comunica.

Podemos escrever a seguinte função para produzir uma representação externa para números complexos:

```
def escreve_compl(c):
    if p_imag(c) >= 0:
        rep_ext = str(p_real(c)) + '+' + str(p_imag(c)) + 'i'
    else:
        rep_ext = str(p_real(c)) + '-' + str(abs(p_imag(c))) + 'i'
    print(rep_ext)
```

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 mostra a seguinte interação:

```
>>> c1 = cria_compl(3, 5)
>>> p_real(c1)
3
>>> p_imag(c1)
5
>>> escreve_compl(c1)
```

```

3+5i
>>> c2 = cria_compl(1, -3)
>>> escreve_compl(c2)
1-3i
>>> c3 = soma_compl(c1, c2)
>>> escreve_compl(c3)
4+2i
>>> escreve_compl(subtrai_compl(cria_compl(2, -8), c3))
-2-10i

```

Podemos agora imaginar que alguém objeta 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  $a + bi$  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 selecionam 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 266 e seguintes, podemos replicar a interação obtida com a representação de

complexos através de tuplos (apresentada na página 267) sem qualquer alteração. Este comportamento revela a independência entre as funções que efetuam 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 *abstração 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*.

### 9.3 Metodologia dos tipos abstratos de dados

Nesta secção apresentamos os passos a seguir no processo de criação de novos tipos de dados. Sabemos que um tipo de dados é uma coleção de entidades, os elementos do tipo, conjuntamente com uma coleção de operações que podem ser efetuadas sobre essas entidades.

A metodologia que apresentamos tem o nome de *metodologia dos tipos abstratos de dados*<sup>3</sup> ou dos *tipos abstratos de informação* 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 abstratos de dados devem ser seguidos quatro passos sequenciais: (1) a identificação das operações básicas; (2) a axiomatização das operações básicas; (3) a escolha de uma representação para os elementos do tipo; e (4) a concretização das operações básicas para a representação escolhida.

Esta metodologia permite a definição de tipos de dados que são independentes da sua representação. Esta independência leva à designação destes tipos por tipos *abstratos* de dados. Exemplificamos estes passos para a definição do tipo complexo.

---

<sup>3</sup>Em inglês, “abstract data types” ou ADTs.

### 9.3.1 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 dados, o conjunto das operações básicas que podem ser efetuadas sobre os elementos desse tipo.

É evidente que ao criar um tipo de dados 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 seletores, 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 *seletores* são operações que permitem aceder (isto é, selecionar) aos constituintes dos elementos do tipo. Em relação aos tipos embutidos em Python, os seletores 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 seletor pois seleciona o número de elementos de uma lista. Em relação ao tipo complexo, podemos selecionar cada um dos seus componentes, existindo assim dois seletores, `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 110 corresponde a um reconhecedor. Em relação ao tipo complexo podemos identificar o reconhecedor, `e_complexo` (lido é *complexo*), que reconhece números complexos, e os reconhecedores `e_compl_zero` e `e_imag_puro`<sup>4</sup> que reconhecem números complexos com certas características;
6. Os *testes* são operações que efetuam 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`<sup>5</sup> que decide se dois números complexos são iguais.

O papel das operações básicas é construir elementos do tipo (os construtores), selecionarem componentes dos elementos do tipo (os seletores), 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 dados, servindo a metodologia dos tipos abstratos de dados 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 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 correspondem a constantes. Para certos tipos, por exemplo as pilhas apresentadas no Capítulo 12 e as árvores apresentadas no Capítulo 13, é possível decidir se estes

---

<sup>4</sup>Nenhuma destas operações foi ainda por nós definida nem realizada.

<sup>5</sup>Esta operação ainda não foi por nós definida nem realizada.

devem ser mutáveis ou imutáveis. Neste caso, algumas das operações básicas poderão ser classificadas como seletores 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 abstrato o que é o tipo), é utilizada a notação matemática para as caraterizar.

Para o tipo complexo definimos as seguintes operações básicas:

1. *Construtores*:

- $cria\_compl : \text{real} \times \text{real} \mapsto \text{complexo}$   
 $cria\_compl(r, i)$  tem como valor o número complexo cuja parte real é  $r$  e cuja parte imaginária é  $i$ .

2. *Seletores*:

- $p\_real : \text{complexo} \mapsto \text{real}$   
 $p\_real(c)$  tem como valor a parte real do complexo  $c$ .
- $p\_imag : \text{complexo} \mapsto \text{real}$   
 $p\_imag(c)$  tem como valor a parte imaginária do complexo  $c$ .

3. *Reconhecedores*:

- $e\_complexo : \text{universal} \mapsto \text{lógico}$   
 $e\_complexo(arg)$  tem o valor *verdadeiro* se  $arg$  é um número complexo e tem o valor *falso* em caso contrário.
- $e\_compl\_zero : \text{complexo} \mapsto \text{lógico}$   
 $e\_compl\_zero(c)$  tem o valor *verdadeiro* se  $c$  é o complexo  $0 + 0i$  e tem o valor *falso* em caso contrário.
- $e\_imag\_puro : \text{complexo} \mapsto \text{lógico}$   
 $e\_imag\_puro(c)$  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:

- $\text{compl\_iguais} : \text{complexo} \times \text{complexo} \mapsto \text{lógico}$

$\text{compl\_iguais}(c_1, c_2)$  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:

$\text{cria\_compl} : \text{real} \times \text{real} \mapsto \text{complexo}$

$\text{p\_real} : \text{complexo} \mapsto \text{real}$

$\text{p\_imag} : \text{complexo} \mapsto \text{real}$

$\text{e\_complexo} : \text{universal} \mapsto \text{lógico}$

$\text{e\_compl\_zero} : \text{complexo} \mapsto \text{lógico}$

$\text{e\_imag\_puro} : \text{complexo} \mapsto \text{lógico}$

$\text{compl\_iguais} : \text{complexo} \times \text{complexo} \mapsto \text{lógico}$

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 abstratas 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 conceitual 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, seletores, 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 dados e a sua realização através de um programa.

### 9.3.2 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 por<sup>6</sup>:

$$\begin{aligned}
 &e\_complexo(cria\_compl(r, i)) = \text{verdadeiro} \\
 &e\_compl\_zero(c) = compl\_iguais(c, cria\_compl(0, 0)) \\
 &e\_imag\_puro(cria\_compl(0, b)) = \text{verdadeiro} \\
 &p\_real(cria\_compl(r, i)) = r \\
 &p\_imag(cria\_compl(r, i)) = i \\
 &cria\_compl(p\_real(c), p\_imag(c)) = \begin{cases} c & \text{se } e\_complexo(c) \\ \perp & \text{em caso contrário} \end{cases} \\
 &compl\_iguais(cria\_compl(x, y), cria\_compl(w, z)) = (x = w) \wedge (y = z)
 \end{aligned}$$

Neste passo especificam-se as relações obrigatoriamente existentes entre as operações básicas, para que estas definam o tipo de um modo coerente.

### 9.3.3 Escolha da representação

O terceiro passo na definição de um tipo de dados 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  $\mathfrak{R}[a + bi] = \{'r':\mathfrak{R}[a], 'i':\mathfrak{R}[b]\}$ .

---

<sup>6</sup>O símbolo  $\perp$  representa indefinido.

### 9.3.4 Realização das operações básicas

O último passo na definição de um tipo de dados 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))
```

Recorde-se da página página 110, que a função `isinstance` determina se o tipo do seu primeiro argumento pertence ao tuplo que é seu segundo argumento.

2. *Seletores:*

```
def p_real(c):
    return c['r']

def p_imag(c):
    return c['i']
```

3. Reconhecedores:

```
def e_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 `e_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 e_compl_zero(c) :
    return zero(c['r']) and zero(c['i'])

def e_imag_puro(c):
    return zero(c['r'])
```

As funções `e_compl_zero` e `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 zero<sup>7</sup>. Por esta razão, a função `zero` poderá ser definida como:

---

<sup>7</sup>Um teste semelhante foi utilizado na Secção 3.4.5.

```
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 267.

## 9.4 Barreiras de abstração

Depois de concluídos todos os passos na definição de um tipo abstrato de dados (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 respetivas 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 do tipo e que, na realidade, caracterizam o seu comportamento como tipo de dados. Qualquer manipulação efetuada sobre uma entidade de um dado tipo deve *apenas* recorrer às operações básicas para esse tipo.

Ao construir um novo tipo abstrato de dados estamos a criar uma nova camada conceitual na nossa linguagem, a qual corresponde ao tipo definido. Esta camada é separada da camada em que o tipo não existe por barreiras de abstração.

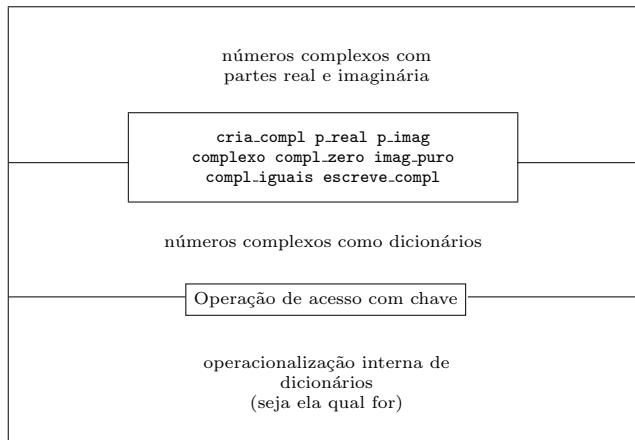


Figura 9.1: Camadas de abstração na definição de complexos.

Conceitualmente, estas barreiras impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas. Na Figura 9.1 apresentamos as camadas e as barreiras de abstração existentes num programa depois da definição do tipo complexo.

Em cada uma destas camadas, a barreira de abstração separa os programas que usam a abstração de dados (que estão situados acima da barreira) dos programas que realizam a abstração 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 abstratos de dados, apenas 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 como o fizemos na secção anterior, nada em Python nos impede de manipular diretamente 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 diretamente 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 diretamente `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 direta 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 abstração tiver sido “quebrada”, o programador teria que percorrer todo o programa e alterar todas as manipulações diretas da estrutura que representa o tipo;
2. O programa torna-se mais difícil de escrever e de compreender, uma vez que a manipulação direta da estrutura subjacente faz com que se perca o nível de abstração correspondente à utilização do tipo.

A definição de complexos que apresentámos neste capítulo não nos permite proibir o acesso direto à representação. Este aspeto é novamente abordado no Capítulo 11, no qual introduzimos um modo de garantir que a representação de um tipo está escondida do exterior.

## 9.5 Notas finais

Neste capítulo, apresentámos a metodologia dos tipos abstratos de dados, que permite separar as propriedades abstratas 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

dados. Esta metodologia foi introduzida por [Liskof and Zilles, 1974] e posteriormente desenvolvida por [Liskof and Guttag, 1986]. O livro [Dale and Walker, 1996] apresenta uma introdução detalhada aos tipos abstratos de dados.

De acordo com esta metodologia, sempre que criamos um novo tipo de dados 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 and Waldinger, 1985].

Os tipos abstratos de dados estão relacionados com o tópico estudado em matemática relacionado com a *teoria das categorias* [Asperti and Longo, 1991], [Simmons, 2011].

## 9.6 Exercícios

- Suponha que desejava criar o tipo vetor em Python. Um vetor num referencial cartesiano pode ser representado pelas coordenadas da sua extremidade  $(x, y)$ , estando a sua origem no ponto  $(0, 0)$ , ver Figura 9.2. Podemos considerar as seguintes operações básicas para vetores:

- *Construtor*:

$$\text{vetor} : \text{real} \times \text{real} \mapsto \text{vetor}$$

$\text{vetor}(x, y)$  tem como valor o vetor cuja extremidade é o ponto  $(x, y)$ .

- *Seletores*:

$$\text{abcissa} : \text{vetor} \mapsto \text{real}$$

$\text{abcissa}(v)$  tem como valor a abcissa da extremidade do vetor  $v$ .

$$\text{ordenada} : \text{vetor} \mapsto \text{real}$$

$\text{ordenada}(v)$  tem como valor a ordenada da extremidade do vetor  $v$ .

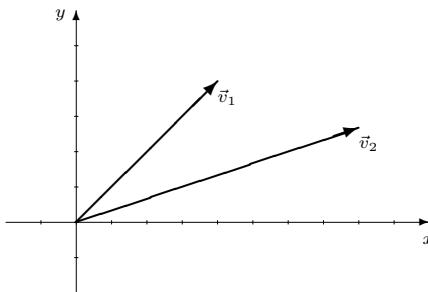


Figura 9.2: Exemplo de vetores.

- *Reconhecedores:*

*e\_vetor : universal  $\mapsto$  lógico*

*e\_vetor(arg)* tem valor *verdadeiro* apenas se *arg* é um vetor.

*e\_vetor\_nulo : vetor  $\mapsto$  lógico*

*e\_vetor\_nulo(v)* tem valor *verdadeiro* apenas se *v* é o vetor  $(0, 0)$ .

- *Teste:*

*vetores\_iguais : vetor  $\times$  vetor  $\mapsto$  lógico*

*vetores\_iguais( $v_1, v_2$ )* tem valor *verdadeiro* apenas se os vetores  $v_1$  e  $v_2$  são iguais.

- (a) Defina uma representação para vetores utilizando tuplos.
  - (b) Escolha uma representação externa para vetores e escreva o transformador de saída.
  - (c) Implemente o tipo vetor.
2. Tendo em atenção as operações básicas sobre vetores da pergunta anterior, escreva funções em Python para:
- (a) Somar dois vetores. A soma dos vetores representados pelos pontos  $(a, b)$  e  $(c, d)$  é dada pelo vetor  $(a + c, b + d)$ .
  - (b) Calcular o produto escalar de dois vetores. O produto escalar dos vetores representados pelos pontos  $(a, b)$  e  $(c, d)$  é dado pelo real  $a.c + b.d$ .
  - (c) Determinar se um vetor 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.
- Especifique as operações básicas do tipo `ponto`.
  - Implemente o tipo `ponto`.
  - 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  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ .
  - 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 o tipo `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 `tempo`, escreva as seguintes funções:
    - *depois : tempo × tempo ↪ 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 ↪ 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 abstrato `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).
- Implemente o tipo `carta`.
  - 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 selecionada aleatoriamente.

6. O tipo de dados *pilha* corresponde a uma pilha de objetos físicos, à qual podemos apenas aceder ao elemento no topo da pilha e apenas podemos adicionar elementos ao topo da pilha.

O tipo pilha é caracterizado pelas operações: *nova\_pilha* (cria uma pilha sem elementos), *empurra* (adiciona um elemento à pilha), *topo* (indica o elemento no topo da pilha), *tira* (remove o elemento no topo da pilha), *e\_pilha* (decide se um objeto computacional é uma pilha), *e\_pilha\_vazia* (testa a pilha sem elementos) e *pilhas\_iguais* (testa a igualdade de pilhas).

- (a) Especifique formalmente estas operações, e classifique-as em construtores, seletores, reconhecedores e testes.
- (b) Escolha uma representação para o tipo pilha, e com base nesta, implemente o tipo *pilha*.



## Capítulo 10

# 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

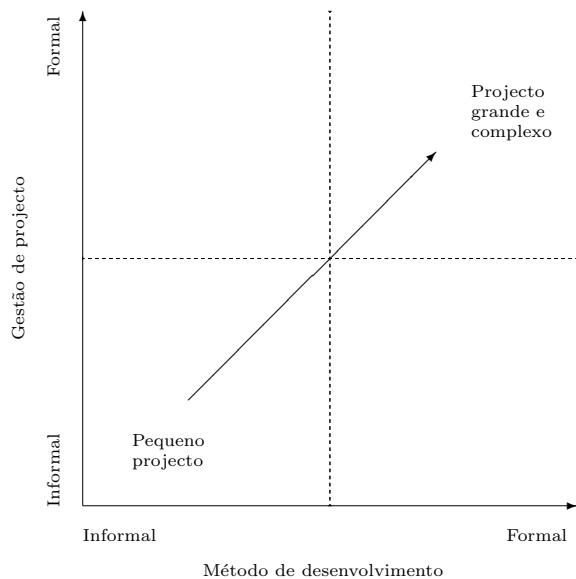


Figura 10.1: O problema da dimensão do programa.

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*<sup>1</sup> que estuda as metodologias para o desenvolvi-

<sup>1</sup>Do inglês “software engineering”.

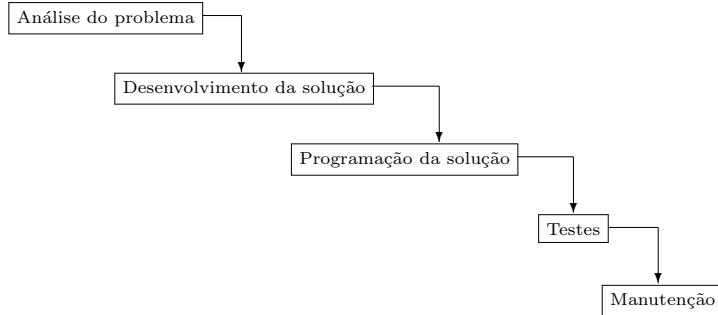


Figura 10.2: Modelo de cascata.

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 10.1). 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 cascata<sup>2</sup>. 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 10.2): a análise do pro-

<sup>2</sup>Do 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.

## 10.1 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 SRS<sup>3</sup>) 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.

## 10.2 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 dados 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, neste modo, descrever rigorosamente como vai ser resolvido o problema sem se entrar, no entanto, nos pormenores inerentes a uma linguagem de programação.

---

<sup>3</sup>Do 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*<sup>4</sup> e a *refinação por passos*<sup>5</sup>, 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

---

<sup>4</sup>Do inglês, “top down design”.

<sup>5</sup>Do inglês, “stepwise refinement”.

frequentemente por SDD<sup>6</sup>, 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 dados que estes manipulam.

A fase de desenvolvimento da solução termina com uma verificação formal dos documentos produzidos.

### 10.3 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:

---

<sup>6</sup>Do 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 dados 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 dados 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 dados.

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.

### 10.3.1 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*”<sup>7</sup>) 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 sintáticos 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

---

<sup>7</sup>Ver a nota de rodapé na página 23.

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 rastreio<sup>8</sup> 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*<sup>9</sup>. 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

---

<sup>8</sup>Do dicionário da Porto Editora: rastrear *v.t.* seguir o rastro de; rastreio *s.m.* acto de rastrear; rasto *s.m.* vestígio que alguém, algum animal ou alguma coisa deixou no solo ou no ar, quando passou.

<sup>9</sup>Do inglês, “bottom-up debugging”.

da fase de desenvolvimento da solução e, consequentemente, à criação de um novo programa.

### 10.3.2 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 10.4). 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 dados 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.

## 10.4 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  $l_1$ ,  $l_2$  e  $l_3$  correspondem aos lados de um triângulo. Para isso terão de verificar as seguintes condições:

1. As variáveis  $l_1$ ,  $l_2$  e  $l_3$  têm valores numéricos.
2. Nenhuma das variáveis  $l_1$ ,  $l_2$  e  $l_3$  é negativa.
3. Nenhuma das variáveis  $l_1$ ,  $l_2$  e  $l_3$  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).

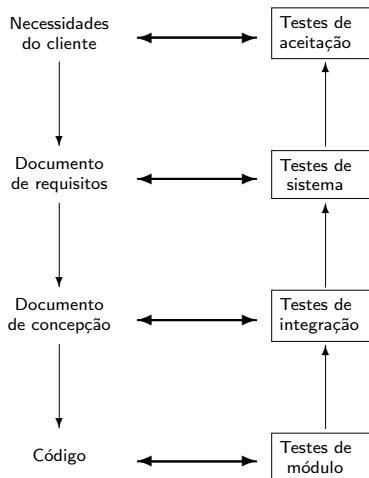


Figura 10.3: Níveis de testes.

Em resumo, os testes de um programa são efectuados a vários níveis (Figura 10.3). 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.

## 10.5 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

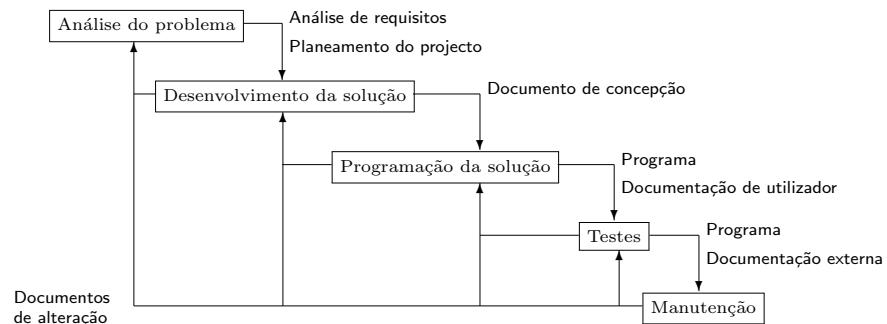


Figura 10.4: Modelo de cascata e documentos associados.

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.<sup>10</sup>, 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.

## 10.6 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 10.4. 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

---

<sup>10</sup>Ver [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 11

## 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, num local existente na “núvem”, numa memória USB ou num CD. A estrutura tipicamente utilizada para armazenar esta informação é chamada um ficheiro.

Um *ficheiro*<sup>1</sup> é um tipo estruturado de dados constituído por uma sequência de elementos, todos do mesmo tipo. Nos ficheiros que consideramos neste capítulo, os *ficheiros sequenciais*, 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.

---

<sup>1</sup>Em inglês, “file”.

Os ficheiros diferem dos outros objetos computacionais considerados até aqui em dois aspetos:

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.

### 11.1 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  $\langle\text{modo}\rangle$ , é uma expressão cujo valor é uma das cadeias de caracteres '`r`', '`w`' ou '`a`'.<sup>2</sup> Ou seja,

$\langle\text{modo}\rangle ::= \text{'r'} \mid \text{'w'} \mid \text{'a'}$

significando a primeira alternativa que o ficheiro é aberto para leitura<sup>3</sup>, a segunda alternativa que o ficheiro é aberto para escrita a partir do início do ficheiro<sup>4</sup> e a terceira alternativa que o ficheiro é aberto para escrita a partir do fim do ficheiro<sup>5</sup>;

- 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.

## 11.2 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 interação:

```
>>> f = open('nada', 'r')
builtins.IOError: [Errno 2] No such file or directory: 'nada'
```

Suponhamos que a diretoria (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 carateres
```

---

<sup>2</sup>Existem outras alternativas que não são tratadas neste livro.

<sup>3</sup>'`r`' é a primeira letra da palavra inglesa "read" (lê).

<sup>4</sup>'`w`' é a primeira letra da palavra inglesa "write" (escreve).

<sup>5</sup>'`a`' é a primeira letra da palavra inglesa "append" (junta).

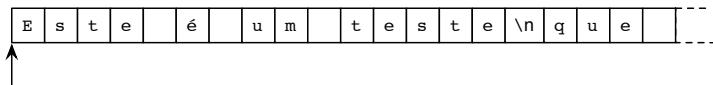


Figura 11.1: Representação do ficheiro `teste.txt`.

A execução pelo Python da instrução

```
t = open('teste.txt', 'r', encoding = 'UTF-16')
```

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 efetua uma leitura, sendo colocado imediatamente após o último símbolo lido, de cada vez que a leitura é feita.

Na Figura 11.1 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 inspecionamos 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 efetuar operações sobre esse ficheiro<sup>6</sup>:

---

<sup>6</sup>Novamente, surge-nos aqui uma situação semelhante à que apareceu na função `s.lower` apresentada na página 243, em que o nome das funções correspondem a um `<nome composto>` em que um dos seus constituintes é o nome de uma variável. Este aspeto é abordado na Secção 12.5.

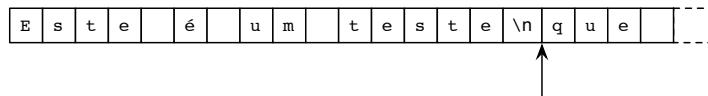


Figura 11.2: Ficheiro `teste.txt` após a execução de `t.readline()`.

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 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 11.2. Repare-se que o carácter de fim de linha faz parte da cadeia de caracteres lida.

A seguinte interação 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 aspeto 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 carateres\n'
>>> ln4 = t.readline()
>>> ln4
''
>>> t.close()
```

A seguinte interação 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 carateres\n'
>>> print(l2)
que mostra como o Python
lê ficheiros de carateres

>>> 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 carateres\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 diretoria (ou pasta) do nosso computador que é utilizada por omissão pelo Python. Contudo, se este ficheiro não existisse na diretoria de omissão, mas sim numa diretoria chamada `exemplos` localizada na diretoria 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 diretórias e do nome de um ficheiro. Embora diferentes sistemas operativos usem símbolos diferentes para especificar caminhos em diretórias (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 carateres é ligeiramente mais complicado. Sabemos que os carateres correspondem a símbolos e que estes são representados internamente em Python utilizando o *Unicode* (este aspeto foi discutido na Secção 4.3). Internamente ao Python, uma cadeia de carateres é uma sequência de zeros e uns correspondente a representações de carateres 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 carateres em *Unicode*, que é identificada como uma cadeia de carateres.

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.

### 11.3 Escrita em ficheiros

De modo a escrevermos informação num ficheiro, teremos primeiro que efetuar 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 efetuar operações sobre esse ficheiro:

1. `<fich>.write(<cadeia de carateres>)`. Esta função escreve, a partir da posição do indicador de escrita, a `<cadeia de carateres>` no ficheiro `<fich>`. O indicador de escrita é movimentado para a posição imediatamente a seguir à cadeia de carateres escrita. Esta função devolve o número de carateres escritos no ficheiro;
2. `<fich>.writelines(<sequência>)`, na qual `<sequência>` é um tuplo ou uma lista cujos elementos são cadeias de carateres. 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 carateres escrita. Esta função não devolve nenhum valor;
3. `<fich>.close()`. Esta função fecha o ficheiro `<fich>`.

A seguinte interação mostra a utilização de operações de escrita e de leitura num ficheiro. Poderá parecer estranho a não utilização da indicação sobre a codificação dos carateres 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 # inspeção 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 interação 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)
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'
```

```
fg 5 25  
ijk
```

```
lmn  
>>> f1.close()
```

## 11.4 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 efetuar no ficheiro, a leitura ou a escrita de informação.

Depois de aberto um ficheiro apenas é possível efetuar operações de leitura ou de escrita, dependendo do modo como este foi aberto.

## 11.5 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 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}
```

4. 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:
```

a 41  
e 45  
i 18  
o 26  
u 20

5. 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 interação 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 ⟨n⟩

Este comando faz com que o espaçamento entre as linhas do texto passe a ser de ⟨n⟩ linhas em branco (o símbolo não terminal ⟨n⟩ corresponde a um inteiro positivo).

\$ sp ⟨n⟩

Este comando insere ⟨n⟩ linhas em branco no texto, tendo em atenção o valor do espaçamento (o símbolo não terminal ⟨n⟩ corresponde a um inteiro positivo).

**\$ ce <texto>**

Este comando causa um salto de linha e centra, na linha seguinte, a cadeia de caracteres representada pelo símbolo não terminal *<texto>*.

**\$ 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 <n>**

Depois da execução deste comando, a margem da esquerda passa a começar na coluna correspondente ao símbolo não terminal *<n>*. Se o valor de *<n>* 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 *<n>* corresponde a um inteiro positivo).

**\$ md <n>**

Depois da execução deste comando, a margem da direita passa a acabar na coluna correspondente ao símbolo não terminal *<n>*. Se o valor de *<n>* 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 *<n>* corresponde a um inteiro positivo).

**\$ pa**

A execução deste comando causa um salto de página, exceto 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.

6. Adicione ao processador de texto do exercício anterior 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.
7. 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 formata dor de texto como palavras.



## Capítulo 12

# Programação com objetos

*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*

Neste capítulo introduzimos o conceito de objeto, uma entidade que contém um conjunto de variáveis internas e um conjunto de operações que manipulam essas variáveis, e abordamos alguns aspectos associados à programação com objetos.

A utilização de objetos em programação dá origem a um paradigma de programação conhecido por *programação com objetos*<sup>1</sup>.

### 12.1 Objetos

Na secção 9.4 discutimos a necessidade de garantir que, após definido um tipo abstrato de dados, é proibido o acesso direto à sua representação. Dissemos também que a abordagem que usámos não nos permite impor esta proibição.

As linguagens de programação desenvolvidas antes do aparecimento da metodologia para os tipos abstratos de dados não possuem mecanismos para garantir que toda a utilização dos elementos de um dado tipo é efetuada recorrendo ex-

---

<sup>1</sup>Do inglês, “object-oriented programming”. Em português, este tipo de programação também é conhecido por programação orientada a objetos, programação orientada por objetos, programação por objetos e programação orientada aos objetos.

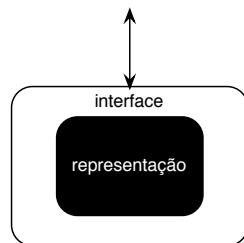


Figura 12.1: Módulo correspondente a um tipo abstrato de dados.

clusivamente às operações específicas desse tipo. Algumas das linguagens de programação mais recentes (por exemplo, Java e C++) fornecem mecanismos que garantem que as manipulações efetuadas sobre os elementos de um tipo apenas utilizam as operações básicas desse tipo. Embora o Python forneça estes mecanismos, os quais serão usados para apresentar os conceitos deste capítulo, alguns deles podem ser contornados.

A proibição do acesso direto à representação de um tipo abstrato é conseguida recorrendo a dois conceitos chamados encapsulação de dados e anonimato da representação.

A *encapsulação de dados*<sup>2</sup> corresponde ao conceito de que o conjunto de funções que corresponde ao tipo de dados 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 o acesso às operações exportadas. O *anonimato da representação*<sup>3</sup> 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 12.1 apresentamos de uma forma esquemática o módulo conceitual correspondente a um tipo abstrato de dados. 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 efetuam a comunicação (*interface*)

---

<sup>2</sup>Do inglês, “data encapsulation”.

<sup>3</sup>Do inglês, “information hiding”.

entre o programa e a representação interna.

A vantagem das linguagens que incorporam os mecanismos da metodologia dos tipos abstratos de dados reside no facto de estas *proibirem efetivamente* a utilização de um tipo abstrato através da manipulação direta 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.

De modo a poder garantir a encapsulação de dados e anonimato da representação em Python, precisamos de recorrer ao conceito de objeto. Um *objeto* é uma entidade computacional que apresenta, não só informação interna (no caso dos tipos de dados, a representação de um elemento de um tipo), como também um conjunto de operações que definem o seu comportamento.

A manipulação de objetos não é baseada em funções que calculam valores, mas sim no conceito de uma entidade que recebe solicitações e que reage apropriadamente a essas solicitações, recorrendo à informação interna do objeto. As funções associadas a um objeto são vulgarmente designadas por *métodos*. É importante distinguir entre solicitações e métodos: uma solicitação é um pedido feito a um objeto; um método é a função utilizada pelo objeto para processar essa solicitação.

Para definir objetos em Python, criamos um módulo (chamado uma *classe*) com o nome do objeto, associando-lhe os métodos correspondentes a esse objeto. Em notação BNF, um objeto em Python é definido do seguinte modo<sup>4</sup>:

```

<definição de objeto> ::= 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 objeto. Existem duas alternativas para esta primeira linha: (1) pode apenas ser definido um nome, ou (2) pode ser fornecido um nome seguido

---

<sup>4</sup>Esta definição corresponde em Python a uma *<definição>*.

de outro nome entre parênteses. O nome imediatamente após a palavra `class` representa o nome do objeto que está a ser definido;

2. A `(definição de método)` corresponde à definição de um método que está associado ao objeto. Podem ser definidos tantos métodos quantos se desejar, existindo tipicamente 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`<sup>5</sup>.

Os objetos 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 97, em que o primeiro nome é sempre `self`.

Para caracterizar a informação interna associada a um objeto 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 267). A razão da nossa decisão prende-se com o facto destas duas variáveis existirem dentro do objeto que corresponde a um número complexo e consequentemente, embora sejam variáveis separadas, elas estão mutuamente ligadas por existirem dentro do mesmo objeto.

Consideremos a seguinte definição de um objeto com o nome `compl`<sup>6</sup> correspondente a um número complexo:

```
class compl:

    def __init__(self, real, imag):
        if isinstance(real, (int, float)) and \
           isinstance(imag, (int, float)):
```

---

<sup>5</sup>A palavra inglesa “self” corresponde à individualidade ou identidade de uma pessoa ou de uma coisa.

<sup>6</sup>Por 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 279).

```

        self.r = real
        self.i = imag
    else:
        raise ValueError ('complexo: argumento errado')

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(str(self.r) + '+' + str(self.i) + 'i')
    else:
        print(str(self.r) + '-' + str(abs(self.i)) + 'i')

```

A execução desta definição pelo Python dá origem à criação do objeto `compl`, estando este objeto 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 objeto é criado, passa a existir uma nova função em Python cujo nome corresponde ao nome do objeto e cujos parâmetros correspondem a todos os parâmetros formais da função `__init__` que lhe está associada, com exceção do parâmetro `self`. O comportamento desta função é definido pela função `__init__` associada ao objeto, a qual devolve um objecto. No nosso exemplo, esta é a função `compl` de dois argumentos. A chamada à função `compl(3, 2)` cria o objeto correspondente ao número complexo  $3 + 2i$ , devolvendo esse objeto. Podemos, assim, originar a interação:

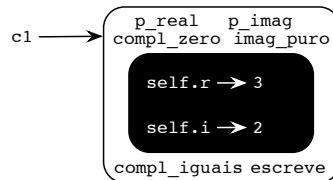


Figura 12.2: Objeto correspondente ao número complexo  $3 + 2i$ .

```
>>> compl
<class '__main__.compl'>
>>> c1 = compl(3, 2)
>>> c1
<__main__.compl object at 0x11fab90>
```

As duas primeiras linhas desta interação mostram que `compl` é uma classe (um tipo de entidades computacionais que existem nos nossos programas) e as duas últimas linhas mostram que o nome `c1` está associado a um objeto, localizado na posição de memória `0x11fab90`.

O resultado da execução da instrução `c1 = compl(3, 2)`, foi a criação do objeto apresentado na Figura 12.2 e a associação do nome `c1` a este objeto. Este objeto “esconde” do exterior a representação do seu número complexo (dado pelos valores das variáveis `self.r` e `self.i`, respectivamente, 3 e 2), 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 objeto correspondem a funções que manipulam as variáveis associadas ao objeto, variáveis essas que estão guardadas *dentro* do objeto, correspondendo a um ambiente local ao objeto, o qual existe enquanto o objeto correspondente existir. Para invocar os métodos associados a um objeto utiliza-se a notação de `(nome composto)` apresentada na página 97, em que o nome é composto pelo nome do objeto, seguido do nome do método<sup>7</sup>. 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 exceção do

<sup>7</sup>Note-se que a função `lower` apresentada na página 243 do Capítulo 8 e as funções que manipulam ficheiros apresentadas no Capítulo 11, na realidade correspondem a métodos.

parâmetro `self`<sup>8</sup>. Assim, podemos originar a seguinte interação:

```
>>> c1 = compl(3, 2)
>>> c1.p_real()
3
>>> c1.p_imag()
2
>>> c1.escreve()
3+2i
>>> c2 = compl(2, -2)
>>> c2.escreve()
2-2i
```

Na interação anterior criámos dois objetos `c1` e `c2` correspondentes a números complexos. Cada um destes objetos 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  $2 - 2i$ .

Devemos fazer as seguintes observações em relação à nossa definição do objeto `compl`:

1. Não definimos um reconhecedor correspondente à operação básica *e\_complexo*. Na realidade, ao criar um objeto, o Python associa automaticamente o nome do objeto à entidade que o representa. A função embutida `type` devolve o tipo do seu argumento, como o mostra a seguinte interação:

```
>>> c1 = compl(3, 5)
>>> type(c1)
<class '__main__.compl'>
```

A função `isinstance`, introduzida na página 277, permite determinar se uma dada entidade corresponde a um complexo:

```
>>> c1 = compl(9, 6)
>>> isinstance(c1, compl)
True
```

---

<sup>8</sup>Na realidade, o nome do objeto é associado ao parâmetro `self`.

Existe assim, uma solicitação especial que pode ser aplicada a qualquer objeto e que pede a identificação do objeto. Por esta razão, ao criarmos um tipo abstrato de dados recorrendo a objetos, não necessitamos de escrever o reconhecedor que distingue os elementos do tipo dos restantes elementos.

2. Na definição das operações básicas para complexos que apresentámos no Capítulo 9, definimos a função *compl\_iguais* (ver página 275) 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 objeto (*self*) com outro complexo qualquer. Assim, por omissão, o primeiro argumento da comparação com outro complexo é o complexo correspondente ao objeto em questão, como a seguinte interação ilustra:

```
>>> c1 = compl(9, 6)
>>> c2 = compl(3, 2)
>>> c3 = compl(9, 6)
>>> c1.compl_iguais(c2)
False
>>> c1.compl_iguais(c3)
True
```

3. Quando no Capítulo 9 definimos as operações básicas para complexos, dissemos que essas operações básicas deveriam ser usadas na utilização de complexos, independentemente da escolha da representação. No entanto, as operações que apresentámos da página 324 à página 325 a interação que apresentámos com números complexos foge à terminologia das operações básicas apresentada no Capítulo 9.

No entanto, definindo as seguintes funções:

```
def cria_compl(r, i):
    return compl(r, i)

def p_real(c):
    return c.p_real()

def p_imag(c):
    return c.p_imag()
```

```

def e_complexo(c):
    return isinstance(c, compl)

def e_compl_zero(c):
    return c.compl_zero()

def e_imag_puro(c):
    return c.imag_puro()

def compl_iguais(c1, c2):
    return c1.compl_iguais(c2)

def escreve_compl(c):
    c.escreve()

```

Podemos obter a mesma interação do que a apresentada na página 9.2<sup>9</sup>:

```

>>> c1 = cria_compl(3, 5)
>>> p_real(c1)
3
>>> p_imag(c1)
5
>>> escreve_compl(c1)
3+5i
>>> c2 = cria_compl(1, -3)
>>> escreve_compl(c2)
1-3i

```

Em programação com objetos existe uma abstração chamada classe. Uma *classe* corresponde a uma infinidade de objetos com as mesmas variáveis e com o mesmo comportamento. É por esta razão, que na definição de um objeto o Python usa a palavra “*class*” (ver a definição apresentada na página 323). Assim, os complexos correspondem a uma classe de objetos 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 interação anterior correspondem

---

<sup>9</sup>Com exceção das operações *soma\_compl* e *subtrai\_compl* que só definimos na Secção 12.6.

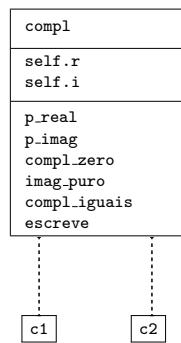


Figura 12.3: Representação da classe `compl` e duas de suas instâncias.

a instanciações do objeto `compl`, sendo conhecidos como *instâncias* da classe `compl`.

Uma classe é um potencial gerador de instâncias, objetos 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 12.3. Uma classe é representada por um retângulo 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 retângulo) e estão ligadas à classe correspondente por uma linha a tracejado.

## 12.2 Objetos mutáveis

Na seção anterior apresentámos um primeiro exemplo de objetos cujas instâncias correspondem a constantes. Nesta seção começamos por apresentar objetos cujas variáveis internas variam ao longo do tempo, por esta razão, correspondentes um *tipo mutável*, e analisamos qual a influência que este aspeto apresenta no seu comportamento.

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 efetuados nessa conta. A situação de uma conta bancária pode ser caracterizada por um dos seguintes aspetos: 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 aspetos, que sabemos estarem associados a contas bancárias, como o número, o titular, o banco, a agência, etc.

Para o tipo conta definimos as seguintes operações básicas<sup>10</sup>:

1. *Construtores*:

- $cria\_conta : \text{inteiro} \mapsto \text{conta}$

$cria\_conta(s)$  tem como valor uma conta bancária cujo saldo inicial é  $s$ .

2. *Seletores*:

- $consulta : \text{conta} \mapsto \text{inteiro}$

$consulta(c)$  tem como valor o saldo da conta  $c$ .

3. *Modificadores*:

- $deposito : \text{conta} \times \text{inteiro} \mapsto \text{inteiro}$

$deposito(c, q)$  altera destrutivamente o valor do saldo,  $s$ , da conta  $c$  para  $s + q$ , tendo como valor  $s + q$ .

- $levantamento : \text{conta} \times \text{inteiro} \mapsto \text{inteiro}$

O valor de  $levantamento(c, q)$  depende da relação entre o saldo da conta  $c$  e a quantia  $q$ : se o saldo,  $s$ , da conta  $c$  for maior ou igual à quantia, significando que a quantia pode ser removida do saldo, sem que o saldo se torne negativo, muda destrutivamente o valor do saldo

---

<sup>10</sup>Assumimos, sem perda de generalidade, que o saldo de uma conta e os valores dos depósitos e levantamentos correspondem a inteiros.

para  $s - q$ , devolvendo o valor do novo saldo,  $s - q$ ; em caso contrário, não altera o saldo da conta e o seu valor é indefinido. Ou seja:

$$\text{levantamento}(c, q) = \begin{cases} \text{consulta}(c) - q & \text{se } \text{consulta}(c) \geq q \\ \perp & \text{se } \text{consulta}(c) < q \end{cases}$$

#### 4. Reconhecedores:

- $e\_conta : \text{universal} \mapsto \text{lógico}$   
 $e\_conta(arg)$  tem o valor *verdadeiro* se  $arg$  é uma conta e tem o valor *falso* em caso contrário.

Para modelar o comportamento de uma conta bancária, utilizamos uma classe, chamada **conta**, cujo estado interno é definido pela variável **saldo** e está associada a funções que efetuam depósitos, levantamentos e consultas de saldo. Esta classe, para além do construtor (`__init__`), apresenta um seletor que devolve o valor do saldo (`consulta`) e dois modificadores, `deposito` e `levantamento`, que, respetivamente, 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 interação:

```
>>> 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
```

De modo análogo ao que fizemos em relação aos números complexos, podemos definir funções que garantem que a interface com contas é feita através da assinatura do tipo conta:

```
def cria_conta(saldo):
    return conta(saldo)

def consulta(c):
    return c.consulta()

def deposito(c, q):
    return c.deposito(q)

def levantamento(c, q):
    return c.levantamento(q)

def e_conta(x):
    return isinstance(x, conta)
```

Obtendo a interação:

```
>>> conta_01 = cria_conta(100)
>>> deposito(conta_01, 50)
```

```
150
>>> consulta(conta_01)
150
>>> levantamento(conta_01, 120)
30
>>> levantamento(conta_01, 50)
Saldo insuficiente
>>> conta_02 = cria_conta(250)
>>> deposito(conta_02, 50)
300
>>> consulta(conta_02)
300
```

Daqui em diante, não apresentaremos mais funções que garantam a interface através da assinatura do tipo e utilizaremos a notação associada a objetos.

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 efetuam depósitos, levantamentos e consultas). A conta bancária evolui de modo diferente quando fazemos um depósito ou um levantamento.

Sabemos que em programação com objetos existe uma abstração chamada classe. Uma *classe* corresponde a uma infinidade de objetos com as mesmas variáveis de estado e com o mesmo comportamento. Por exemplo, a conta bancária que definimos corresponde a uma classe de objetos 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 interação, correspondem a *instâncias* da classe **conta**. Já vimos que as classes e instâncias são representadas graficamente como se mostra na Figura 12.4.

Ao trabalhar com objetos, 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 objetos **conta\_01** e **conta\_02** são iguais? Embora eles tenham

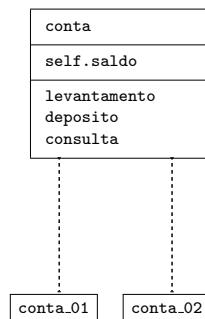


Figura 12.4: Representação da classe `conta` e suas instâncias.

sido definidos exatamente da mesma maneira, como instâncias da mesma classe, estes dois objetos têm estados distintos, como se mostra na seguinte interação:

```

>>> 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* objeto (correspondem na realidade a uma conta conjunta). Por serem o mesmo objeto, qualquer alteração a uma destas contas reflete-se na outra:

```

>>> conta_01 = conta(100)
  
```

```
>>> conta_02 = conta_01
>>> conta_01.consulta()
100
>>> conta_02.levantamento(30)
70
>>> conta_01.consulta()
70
```

### 12.3 Classes, subclasses e herança

Uma das vantagens da programação com objetos 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 efetuar 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 efetuada. Em programação com objetos, 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 efetuar levantamentos, depósitos e consultas ao saldo. Esta classe pressupõe que existe uma variável de estado `self.saldo_min` que define qual o

saldo mínimo após um levantamento.

A classe `conta_gen`, correspondente a uma conta genérica, pode ser realizada através da seguinte definição<sup>11</sup>. Note-se que a definição de `conta_gen` parece contradizer o que dissemos na página 324, no sentido em que não contém o método `__init__`. Este aspeto é clarificado mais à frente.

```
class conta_gen:

    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 >= self.saldo_min:
            self.saldo = self.saldo - quantia
            return self.saldo
        else:
            print('Saldo insuficiente')
```

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 condições:
  - 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 condições:
  - a sua abertura não exige um saldo mínimo;

---

<sup>11</sup>Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta definição.

— 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 respetivos 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 objetos 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, exceto no caso em que comportamento específico está indicado para a subclasse. Diz-se que a subclasse *herda* o comportamento da superclasse, exceto quando este é explicitamente alterado na subclasse.

É na definição de subclasses que temos que considerar a parte opcional da `<definição de objeto>` apresentada na página 323, e reproduzida aqui para facilitar a leitura:

```
<definição de objeto> ::= 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 subclasse de `<nome2>`. Neste caso, a classe `<nome1>` “herda” automaticamente 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 os métodos `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`, sobrepõe-se ao método homônimo 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, saldo_inicial, ordenado):
        if saldo_inicial >= ordenado:
            self.saldo = saldo_inicial
            self.saldo_min = -ordenado
        else:
            print('O saldo deve ser maior que o ordenado')
```

Nesta classe, definimos o método `__init__`, o qual recebe o saldo inicial e o valor do ordenado. O ordenado define que o saldo mínimo corresponde a `-ordenado`. Note-se que associado à classe `conta_ordenado` aparece o método `__init__`, que não existia na classe `conta_gen`. Isto significa que não estamos interessados em criar instâncias da classe `conta_gen`, mas que iremos criar instâncias da classe `conta_ordenado`. Na realidade, a classe `conta_gen` apenas serve para definir o comportamento genérico de uma conta bancária.

Analogamente, a classe `conta_jovem` é definida como uma subclasse de `conta_gen`, definindo apenas o método `__init__`, no qual o valor do saldo mínimo é zero:

```
class conta_jovem(conta_gen):

    def __init__(self, saldo_inicial):
        self.saldo = saldo_inicial
        self.saldo_min = 0
```

Estas classes permitem-nos efetuar a seguinte interação:

```
>>> conta_ord_01 = conta_ordenado(300, 500)
O saldo deve ser maior que o ordenado
>>> conta_ord_01 = conta_ordenado(800, 500)
```

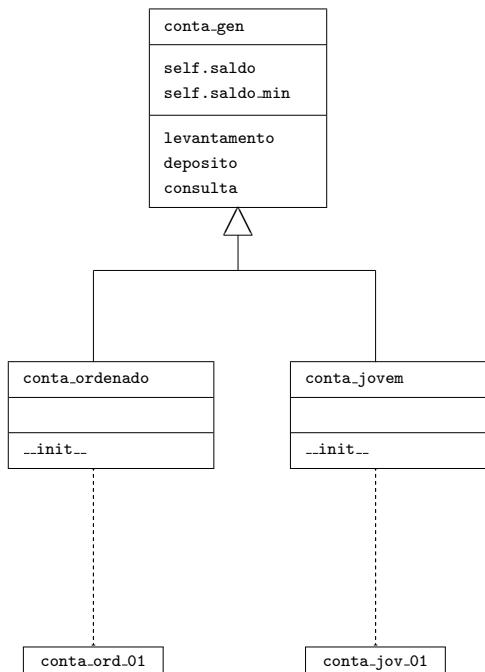


Figura 12.5: Hierarquia de contas e respetivas instâncias.

```

>>> 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
  
```

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 12.5 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, exceto 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 PIN<sup>12</sup>. Esta nova conta, 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 correto, 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 `accede`, o qual, sempre que é fornecido um PIN errado aumenta em uma unidade o valor do contador de tentativas de acesso incorretas. Ao ser efetuado um acesso com o PIN correto, o valor do contador de tentativas incorretas volta a ser considerado zero. O método `accede` devolve `True` se é efetuado um acesso com o PIN correto 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
```

---

<sup>12</sup>Do inglês, “Personal Identification Number”.

```

        return True
    else:
        self.contador = self.contador + 1
        if self.contador == 3:
            print('Conta bloqueada')
        else:
            print('PIN incorreto')
            print('Tem mais', 3 - self.contador, 'tentativas')
    return False

```

Os métodos `levantamento`, `deposito` e `consulta` utilizam o método `acede` para verificar a correção do PIN e, em caso de este acesso estar correto, 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 ação 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 incorreto')
            print('Tem mais', 3 - self.contador, 'tentativas')
    return False

```

Com esta classe, podemos originar a seguinte interação:

```

>>> conta_jcp_01 = conta_jovem_com_pin(120, '1234')
>>> conta_jcp_01.levantamento(30, 'abcd')
PIN incorreto
Tem mais 2 tentativas

```

```
>>> conta_jcp_01.levantamento(30, 'ABCD')
PIN incorreto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
90
>>> conta_jcp_01.altera_codigo('abcd')
PIN incorreto
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 incorreto
Tem mais 2 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
PIN incorreto
Tem mais 1 tentativas
>>> conta_jcp_01.levantamento(30, '1234')
Conta bloqueada
```

## 12.4 Número arbitrário de argumentos

A abordagem que seguimos na seção anterior para definir a classe `conta_jovem_com_pin` não foi a mais elegante: redefinimos os métodos associados à classe `conta_jovem`; teríamos que redefinir estes mesmos métodos para criar qualquer outro tipo de conta com PIN; não capturámos a abstração de entidade com PIN.

O modo mais correto de criar uma conta com PIN seria o de começar por definir o comportamento de uma entidade com PIN, dentro da qual existiria uma conta de qualquer tipo. Uma entidade com PIN é caracterizada por variáveis de estado

correspondentes ao PIN, a um contador de acessos incorretos e a um objeto protegido pelo PIN.

Podemos definir a classe `obj_com_pin` do seguinte modo:

```
class obj_com_pin:

    def __init__(self, pin, obj):
        self.pin = pin
        self.contador = 0
        self.obj = obj

    def verifica_PIN(self, pin):
        if self.contador == 3:
            raise ValueError('Conta bloqueada')
        elif pin == self.pin:
            self.contador = 0
            return True
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
                raise ValueError('Conta bloqueada')
            else:
                print('PIN incorreto')
                mensagem = 'Tem mais ' + str(3-self.contador) + \
                           ' tentativas'
                raise ValueError(mensagem)

    def acede(self, pin):
        if self.verifica_PIN(pin):
            return self.obj

    def altera_PIN(self, pin):

        if self.verifica_PIN(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')

```

Com esta classe podemos gerar a interação:

```

>>> conta = obj_com_pin('ABC', conta_jovem(100))
>>> conta.acede('ABC').consulta()
100
>>> conta.acede('ABC').levantamento(50)
50
>>> conta.acede('XPTO').consulta()
PIN incorreto
Tem mais 2 tentativas
>>> conta.altera_PIN('ABC')
Introduza o novo PIN
--> XPTO
Para verificação
Volte a introduzir o novo PIN
--> XPTO
PIN alterado
>>> conta.acede('XPTO').consulta()
50

```

Uma outra alternativa para definir a classe `obj_com_pin` corresponde, não em criar o objeto protegido pelo PIN como um argumento do método que cria a instância, mas sim criá-lo durante a inicialização das variáveis de estado da instância. Para isso, teremos que fornecer ao método `__init__` o nome da classe correspondente ao objeto a criar e os argumentos necessários para a criação desse objeto. Esta abordagem levanta-nos o problema que instâncias de diferentes classes podem ter que ser criadas com diferente número de argumentos. Veja-se, por exemplo, os argumentos necessários para criar uma `conta_ordenado` (página 339) e os argumentos necessários para criar uma `conta_jovem` (página 339). Vamos precisar da possibilidade de criar funções que aceitam um número

arbitrário de argumentos.

A utilização de funções com um número arbitrário de argumentos já tem sido feita nos nossos programas. Por exemplo, a função embutida `print` 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`.

Vamos considerar um novo aspecto na definição de funções correspondente à utilização de 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 interação, a qual revela este novo aspeto 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`.

Podemos agora escrever a seguinte versão da classe `obj_com_pin`<sup>13</sup>, na qual omitimos propositadamente a possibilidade de alteração do PIN:

```
class obj_com_pin:

    def __init__(self, pin, cria_obj, *args):
        self.pin = pin
        self.contador = 0
        self.obj = cria_obj(*args)

    def acede(self, pin):
        if self.contador == 3:
            raise ValueError('Conta bloqueada')
        elif pin == self.pin:
            self.contador = 0
            return self.obj
        else:
            self.contador = self.contador + 1
            if self.contador == 3:
```

---

<sup>13</sup>Agradeço à Prof. Maria dos Remédios Cravo a sugestão desta abordagem.

```
        raise ValueError('Conta bloqueada')
    else:
        mensagem = 'PIN incorreto\n' + 'Tem mais ' + \
                   str(3-self.contador) + ' tentativas'
        raise ValueError(mensagem)
```

Podemos agora gerar a interação:

```
>>> co = obj_com_pin('ABC', conta_ordenado, 5000, 1000)
>>> co.acede('ABC').deposito(500)
5500
>>> cj = obj_com_pin('XPTO', conta_jovem, 100)
>>> cj.acede('XPTO').levantamento(50)
50
```

## 12.5 Objetos em Python

O Python é uma linguagem baseada em objetos. Embora a nossa utilização do Python tenha fundamentalmente correspondido à utilização da sua faceta imperativa, na realidade, todos os tipos de dados embutidos em Python correspondem a classes. Recorrendo à função `type`, introduzida na página 327, podemos obter a seguinte interação que revela que os tipos embutidos com que temos trabalhado não são mais do que objetos 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 Seção 11.2. Dissemos na altura que o nome desta função podia ser tratado como um nome composto. O que na realidade se passa é que ao abrimos um ficheiro, estamos a criar um objeto, objeto esse em que um dos métodos tem o nome de `readline`.

## 12.6 Polimorfismo

Já vimos que muitas das operações do Python são operações sobrecarregadas, ou seja, aplicam-se a vários tipos de dados. 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*<sup>14</sup>, 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 37, que dissemos que internamente ao Python existiam duas operações, que representámos por `+_Z` e `+_R`, que, respetivamente, 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 polimorfismo<sup>15</sup>. Esta faceta do polimorfismo dá origem a operações sobrecarregadas.

Após definirmos o tipo correspondente a números complexos, no Capítulo 9, 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)
```

---

<sup>14</sup>A palavra “polimorfismo” é derivada do grego e significa apresentar múltiplas formas.

<sup>15</sup>O tipo de polimorfismo que definimos nesta seção corresponde ao conceito inglês de “subtype polymorphism”.

com a qual podemos gerar a interação:

```
>>> 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 325, como o mostra a seguinte interação:

```
>>> 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 interação

```
>>> 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 diretamente 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 complexos<sup>16</sup>. 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
```

---

<sup>16</sup>No 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_imag() * outro.p_real() - \
              self.p_real() * outro.p_imag()) / den
        return compl(r, i)
    except ZeroDivisionError:
        print("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 interação:

```

>>> 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

```

## 12.7 Notas finais

Neste capítulo apresentámos um estilo de programação, conhecido como programação com objetos, que é centrado em objetos, entidades que possuem um estado interno e reagem a mensagens. Os objetos correspondem a entidades, tais como contas bancárias, livros, estudantes, etc. O conceito de objeto agrupa as funções que manipulam dados (os métodos) com os dados que representam o estado do objeto.

O conceito de objeto foi introduzido com a linguagem SIMULA<sup>17</sup>, foi vulgarizado em 1984 pelo sistema operativo do Macintosh, foi adotado 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.

## 12.8 Exercícios

1. 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 objeto 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
```

2. 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

---

<sup>17</sup>[Dahl and 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. Suponha que desejava criar o tipo *conjunto*. Considere as seguintes operações para conjuntos:

*Construtores:*

- *novo\_conj* :  $\{\} \mapsto \text{conjunto}$   
*novo\_conj()* tem como valor um conjunto sem elementos.
- *insere\_conj* :  $\text{elemento} \times \text{conjunto} \mapsto \text{conjunto}$   
*insere\_conj( $e, c$ )* tem como valor o resultado de inserir o elemento  $e$  no conjunto  $c$ ; se  $e$  já pertencer a  $c$ , tem como valor  $c$ .

*Seletores:*

- *retira\_conj* :  $\text{elemento} \times \text{conjunto} \mapsto \text{conjunto}$   
*retira\_conj( $e, c$ )* tem como valor o resultado de retirar do conjunto  $c$  o elemento  $e$ ; se  $e$  não pertencer a  $c$ , tem como valor  $c$ .

*Reconhecedores:*

- *e\_conjunto* :  $\text{universal} \mapsto \text{lógico}$   
*e\_conjunto( $arg$ )* tem o valor *verdadeiro* se  $arg$  é um conjunto, e tem o valor *falso*, em caso contrário.
- *e\_conj\_vazio* :  $\text{conjunto} \mapsto \text{lógico}$   
*e\_conj\_vazio( $c$ )* tem o valor *verdadeiro* se o conjunto  $c$  é o conjunto vazio, e tem o valor *falso*, em caso contrário.

*Testes:*

- *conjuntos\_iguais* :  $\text{conjunto} \times \text{conjunto} \mapsto \text{lógico}$   
*conjuntos\_iguais( $c_1, c_2$ )* tem o valor *verdadeiro* se os conjuntos  $c_1$  e  $c_2$  forem iguais, ou seja, se tiverem os mesmos elementos, e tem o valor *falso*, em caso contrário.
- *pertence* :  $\text{elemento} \times \text{conjunto} \mapsto \text{lógico}$   
*pertence( $e, c$ )* tem o valor *verdadeiro* se o elemento  $e$  pertence ao conjunto  $c$  e tem o valor *falso*, em caso contrário.

*Operações adicionais:*

- *cardinal* :  $\text{conjunto} \mapsto \text{inteiro}$   
 $\text{cardinal}(c)$  tem como valor o número de elementos do conjunto  $c$ .
  - *subconjunto* :  $\text{conjunto} \times \text{conjunto} \mapsto \text{lógico}$   
 $\text{subconjunto}(c_1, c_2)$  tem o valor *verdadeiro*, se o conjunto  $c_1$  for um subconjunto do conjunto  $c_2$ , ou seja, se todos os elementos de  $c_1$  pertencerem a  $c_2$ , e tem o valor *falso*, em caso contrário.
  - *uniao* :  $\text{conjunto} \times \text{conjunto} \mapsto \text{conjunto}$   
 $\text{uniao}(c_1, c_2)$  tem como valor o conjunto união de  $c_1$  com  $c_2$ , ou seja, o conjunto formado por todos os elementos que pertencem a  $c_1$  ou a  $c_2$ .
  - *interseccao* :  $\text{conjunto} \times \text{conjunto} \mapsto \text{conjunto}$   
 $\text{interseccao}(c_1, c_2)$  tem como valor o conjunto intersecção de  $c_1$  com  $c_2$ , ou seja, o conjunto formado por todos os elementos que pertencem simultaneamente a  $c_1$  e a  $c_2$ .
  - *diferenca* :  $\text{conjunto} \times \text{conjunto} \mapsto \text{conjunto}$   
 $\text{diferenca}(c_1, c_2)$  tem como valor o conjunto diferença de  $c_1$  e  $c_2$ , ou seja, o conjunto formado por todos os elementos que pertencem a  $c_1$  e não pertencem a  $c_2$ .
- (a) Escreva uma axiomatização para as operações do tipo conjunto.  
(b) Defina uma representação para conjuntos utilizando listas.  
(c) Defina a classe `conjunto` com base na representação escolhida.

4. 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`.



# Fundamentos de Programação

Resumos da Matéria Teórica fundamental do livro...  
“programação em scheme”



## Capítulo I – Noções Básicas

Pág. 7:

- Um **Algoritmo** é uma sequência finita de instruções, bem definidas e não ambíguas, cada uma das quais pode ser executada mecanicamente num intervalo de tempo finito com uma quantidade de esforço finita.
  - Um *algoritmo* é *rigoroso* (cada instrução do algoritmo deve especificar exacta e rigorosamente o que deve ser feito, não havendo lugar para ambiguidade), *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) e *deve terminar* (o algoritmo deve levar a uma situação em que o objectivo tenha sido atingido e não existam mais instruções para ser executadas).
- Um **Programa** é um algoritmo escrito numa linguagem que é entendida por um computador, chamada linguagem de programação.

Pág. 9:

- A **Abordagem do Topo para a Base** é uma técnica que consiste em identificar os principais subproblemas que constituem o problema a resolver, e em determinar qual a relação entre esses subproblemas. O processo é aplicado repetitivamente para cada um dos subproblemas até se atingirem problemas cuja solução não temos dificuldade em escrever.

Pág.10:

- A **Abstracção** é uma técnica que consiste em ignorar informação irrelevante, num dado contexto.

Pág.12:

- A linguagem Scheme é constituída por formas. O **Interpretador do Scheme** é uma “caixa electrónica” que comprehende as formas da linguagem Scheme, ou seja, sabe reagir apropriadamente a uma forma da linguagem.

Pág. 16:

- O valor de uma **constante** é a própria **constante**. (Doh!)

Pág. 20:

- Para calcular o valor de uma **combinação**, avaliam-se as subexpressões na combinação (por qualquer ordem). Após esta avaliação, aplica-se o procedimento correspondente ao valor da primeira subexpressão (o operador) aos argumentos que correspondem aos valores das restantes subexpressões (os operandos).

Pág. 23:

- Uma operação que produz resultados do tipo lógico chama-se **predicado**. Uma expressão cujo valor é do tipo lógico chama-se **condição**.

Pág. 30:

- As definições recursivas são constituídas por duas partes distintas:
  - Uma parte *básica*, ou *caso terminal*, a qual constitui a versão mais simples do problema para o qual a solução é conhecida.
  - Uma parte *recursiva*, ou *caso geral*, na qual o problema é definido em termos de uma versão mais simples de si próprio.

Pág. 32:

- Um **ambiente** é uma associação entre nomes e objectos computacionais.

Pág. 34:

- O **valor de um nome** é o objecto computacional associado a esse nome no ambiente em consideração.

## Capítulo II – Procedimentos Compostos

Pág. 47:

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

Pág. 73:

- O **ambiente global** contém todos os nomes que foram fornecidos directamente ao interpretador do Scheme.

Pág. 74:

- Um **ambiente local** corresponde a uma associação entre nomes e objectos computacionais, a qual é realizada durante o processo de avaliação de uma forma. Utilizando apenas os conceitos desde capítulo, um ambiente local “desaparece” no momento em que termina a avaliação da forma que deu origem à sua criação.
- Diz-se que um nome está **ligado** num dado ambiente se existe um objecto computacional associado ao nome nesse ambiente.

Pág. 77:

- Um **nome local** tem significado dentro do corpo de uma expressão lambda.

Pág. 78:

- Um **nome** que aparece no corpo de uma expressão lambda e que não é um **nome** local diz-se **não local**. (Doh!)

Pág. 79:

- Define-se o **domínio** de um nome como a gama de formas nas quais o nome é conhecido, ou seja, o conjunto das formas onde o nome pode ser utilizado.
- O tipo de **domínio** utilizado em Scheme é chamado **domínio estático**: o **domínio** de um nome é definido em termos da estrutura do programa (o encadeamento dos seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

### Capítulo III – Processos Gerados por Procedimentos

Pág. 79:

- 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.

Pág. 94:

- A um processo recursivo que cresce linearmente com um valor dá-se o nome de **processo recursivo linear**.

Pág. 98:

- 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.

- A um processo iterativo cujo número de operações cresce linearmente com um valor dá-se o nome de **processo iterativo linear**.

Pág. 99:

- A **recursão em procedimentos** refere-se à definição do procedimento em termos de si próprio, ao passo que a **recursão em processos** refere-se ao padrão de evolução do processo.

Pág. 110:

- A **sequenciação** permite especificar que uma dada sequência de expressões deve ser avaliada pela ordem em que aparece.

Pág. 111:

- A **ordem de crescimento** de um processo é uma medida grosseira dos recursos consumidos pelo processo em função do grau de dificuldade do problema.

Pág. 112:

- $R(n)$  tem ordem de crescimento  $\Theta(f(n))$  se existirem constantes positivas  $k_1$  e  $k_2$ , tais que, para valores suficientemente grandes de  $n$ , os valores de  $R(n)$  estão limitados por  $k_1(f(n))$  e  $k_2(f(n))$ .

## Capítulo IV – Procedimentos de Ordem Superior

Pág. 121:

- Um objecto computacional é um **cidadão de primeira classe** se:
  - Pode ser nomeado;
  - Pode ser utilizado como argumento de procedimentos;
  - Pode ser devolvido por procedimentos;
  - Pode ser utilizado como componente de estruturas de informação;

Pág. 122:

- Um procedimento que recebe outros procedimentos como parâmetros ou cujo valor é um procedimento é chamado um **procedimento de ordem superior** ou, alternativamente, um **funcional**.

## Capítulo V – Abstracção de Dados

Pág. 142:

- Um **tipo de informação** é constituído por um conjunto de objectos e um conjunto de operações aplicáveis a esses objectos.

Pág. 143:

- Os **tipos de informação elementares** contêm elementos que não são decomponíveis; os **tipos de informação estruturados** contêm elementos que são compostos por várias partes.

Pág. 144:

- A **representação interna** de um objecto computacional corresponde à representação manipulada pelo interpretador do Scheme.
- A **representação externa** de um objecto computacional corresponde ao modo como esse objecto computacional é mostrado pelo Scheme ao mundo exterior.

Pág. 148:

- A **abstracção de dados** consiste na separação entre as partes do programa que lidam com o modo como os dados são *utilizados* das partes do programa que lidam com o modo como os dados são *representados*.

Pág. 152:

- Uma operação para combinar entidades satisfaz a **propriedade do fecho** se os resultados da combinação de entidades com essa operação puderem ser combinados através da mesma operação.

Pág. 155:

- Na **Metodologia dos tipos abstractos de informação** são seguidos quatro passos sequenciais:
  1. A identificação das operações básicas.
  2. A axiomatização das operações básicas.
  3. A escolha de uma representação para os elementos do tipo.
  4. A concretização das operações básicas para a representação escolhida.

Pág. 156:

- As **operações básicas** dividem-se em quatro grupos:
  1. Os **construtores** permitem construir novos elementos do tipo.
  2. Os **selectores** permitem aceder (isto é, seleccionar) aos constituintes dos elementos do tipo.
  3. Os **reconhecedores** 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.

4. Os **testes** efectuam comparações entre os elementos do tipo.

Pág. 161:

- A **encapsulação da informação** corresponde ao conceito de que o conjunto de procedimentos que representa o tipo de informação engloba toda a informação referente ao tipo.
- O **anonimato da representação** corresponde ao conceito de que o módulo correspondente aos procedimentos do tipo guarda como segredo o modo escolhido para representar os elementos do tipo.

Pág. 162:

- As **barreiras de abstracção** impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas.

Pág. 165:

- Uma **lista** contém uma colecção ordenada de elementos. Podemos inserir novos elementos em qualquer posição da lista (inserindo um elemento de cada vez) e podemos remover qualquer elemento da lista.

Pág. 181:

- Uma **árvore** é uma estrutura hierárquica composta por uma raiz que domina outras árvores.

Pág. 183:

- Uma **árvore binária** é uma estrutura hierárquica composta por uma raiz que domina exactamente duas árvores binárias.

## Capítulo VI – O Desenvolvimento de Programas

## Capítulo VII – Programação Imperativa

Pág. 229:

- Um objecto tem **estado** se o seu comportamento é influenciado pela sua história. O estado de uma entidade é caracterizado por uma ou mais *variáveis de estado*, as quais mantêm informação sobre a sua história, de modo a poder determinar o comportamento da entidade.

Pág. 233:

- Um **efeito** corresponde à alteração de qualquer entidade associada a um processo computacional. Os procedimentos baseados em efeitos não são avaliados pelo valor que produzem, mas sim pelo efeito que originam.

Pág. 239:

- A programação **imperativa** baseia-se no conceito de efeito. Em programação imperativa, um programa é considerado como uma sequência de instruções, cada uma das quais produz um efeito.

Pág. 242:

- Um **vector** é um tipo estruturado de informação cujos elementos são acedidos indicando a posição do elemento dentro da estrutura.

## Capítulo VIII – Avaliação Baseada em Ambientes

Pág. 259:

- Um **enquadramento** é constituído por um conjunto de ligações, as quais associam um nome a um valor. Num **enquadramento** não podem existir duas ligações distintas para o mesmo nome.

Pág. 261:

- O **ambiente** associado a um enquadramento é a sequência de enquadramentos constituída por esse enquadramento e por todos os seus enquadramentos envolventes.
- O **valor de uma variável** em relação a um ambiente é o valor da ligação dessa variável no primeiro enquadramento que contém uma ligação para essa variável.

## Capítulo IX – Estruturas Mutáveis

Pág. 294:

- Uma estrutura de informação diz-se **mutável** quando é modificada destrutivamente, à medida que o programa é executado.

Pág. 311:

- Um **ponteiro** é um objecto computacional que aponta. Com a utilização de ponteiros, não estamos interessados no valor do ponteiro, mas sim no valor para onde ele aponta.

Pág. 314/315:

- Ver definição de **amontoado**, ver como se cria o **lixo**.
  - A **recolha de lixo** autoriza a criação de lixo durante a execução de um programa. Quando o espaço disponível no amontoado desce abaixo de um certo valor limite, a execução do programa é temporariamente suspensa, e outro programa, o programa da recolha de lixo, é automaticamente executado. Este programa identifica o lixo existente e volta a colocá-lo no amontoado.
- (...)

A recolha do lixo é efectuada em duas fases sequenciais:

1. Na **fase de marcação**(...), toda a zona de memória a que é possível aceder a partir de entradas no ambiente global, ou a partir da informação que é mantida pelo Scheme (nomeadamente a indicação do enquadramento que está a ser considerado no momento da avaliação) é marcada como não sendo lixo.
2. Na **fase de varrimento**(...), 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.

## Capítulo X – Programação com Objectos

Pág. 329:

- Um **objecto** é uma entidade computacional que apresenta não só um estado como também um conjunto de procedimentos que definem o seu comportamento.
- Os objectos reagem a **mensagens**. Os procedimentos de um objecto são designados por **métodos**.

Pág. 330:

- Uma **classe** define as variáveis de estado e o comportamento de todas as entidades que lhe pertencem. Uma **instância** corresponde a um objecto particular de uma dada classe.

Pág.336:

- 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

## **Perguntas e exemplos de respostas da matéria em geral:**

### **I**

**Justifique a seguinte afirmação: “a abstracção de dados aumenta o poder expressivo da nossa linguagem de programação”.**

A abstracção de dados, sistematizada numa metodologia que permite a definição de novo tipos, aumenta de facto o poder expressivo da nossa linguagem de programação, na medida em que permite a utilização de um tipo como se este fosse pré-definido na linguagem. O programa deve apenas recorrer a um conjunto de operações que são específicas ao tipo, podendo mesmo desconhecer a sua representação.

**Apresente a definição de um procedimento em notação BNF. Explique cada um dos constituintes desta definição.**

Um procedimento em Scheme corresponde a uma expressão lambda, a qual tem a seguinte sintaxe:

<expressão lambda> ::= (lambda (<parâmetros formais>) <corpo>)

<parâmetros formais> ::= <nome>\*

<corpo> ::= <definição>\* <expressão>+

em que:

1• a palavra lambda (um símbolo terminal) indica que se trata de uma expressão lambda;

2• <parâmetros formais> corresponde a zero ou mais nomes;

3• <corpo> corresponde a zero ou mais definições seguidas de uma ou mais expressões.

**O que é a ordem de crescimento de um processo? Qual a necessidade de a definir?**

A ordem de crescimento de um processo é uma medida grosseira dos recursos consumidos pelo processo em função do grau de dificuldade do problema.

Uma das preocupações que devemos ter durante o desenvolvimento de um programa é tentar minimizar o número de recursos que este consome, daí a importância do estudo da ordem de crescimento.

**Explique o conceito de ciclo “lê-avalia-escreve”.**

A utilização interactiva do Scheme corresponde à repetição de um ciclo em que o interpretador de Scheme

1• lê uma forma,

2• avalia essa forma (ou executa os passos correspondentes à forma) e

3• escreve o resultado da avaliação.

Este ciclo é chamado ciclo lê-avalia-escreve. Uma sessão em Scheme corresponde a um ciclo, tão longo quanto o utilizador desejar, de leitura de formas, avaliação dessas formas e apresentação dos resultados.

**Diga o que são operações básicas de um tipo abstracto de informação e quais os grupos em que estas são divididas.**

As operações básicas de um tipo abstracto de informação correspondem ao mínimo de operações que permitem caracterizar o tipo.

Estas operações dividem-se em quatro grupos, os construtores, os selectores, os reconhecedores e os testes.

**Diga quais as regras associadas à estrutura de blocos e explique as suas vantagens.**

**Um bloco corresponde a um “conjunto” de instruções, associado às seguintes regras:**

1• Cada bloco corresponde a um sub-problema que o procedimento correspondente tem de resolver. Este aspecto permite modularizar o programa.

2• O algoritmo usado por um bloco está “escondido” do resto do programa. Isto permite controlar a complexidade do programa.

3• 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 permite a protecção efectiva da informação definida em cada bloco da utilização não autorizada por parte de outros blocos.

**Diga qual a diferença entre tipos de informação elementares e tipos de informação estruturados.**

- 1• Os tipos elementares são caracterizados pelo facto das suas constantes (os elementos do tipo) serem tomadas como indecomponíveis (ao nível da utilização do tipo).
- 2• Os tipos estruturados são caracterizados pelo facto das suas constantes serem constituídas por um agregado de valores.

**Numa gramática em notação BNF existem símbolos terminais e símbolos não terminais. Diga o que são estes símbolos e exemplifique utilizando a definição de uma combinação em Scheme.**

Símbolos não terminais designam componentes da linguagem que está a ser definida. Por exemplo, forma, expressão, etc. Representam-se em notação BNF dentro de parêntesis angulares. Símbolos terminais são símbolos que aparecem nas frases da linguagem que está a ser definida.

Exemplo: definição de combinação

`<combinação> ::= (<operador> <expressão>*)`

Nesta definição, os símbolos não terminais são `<combinação>`, `<operador>` e `<expressão>`, e os símbolos terminais são “(“ e “)”.

**Faça a distinção entre programa e processo computacional.**

Um programa é uma sequência 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. Enquanto que um processo computacional é um ente imaterial que existe dentro de um computador durante a execução de um programa, e cuja evolução ao longo do tempo é ditada pelo programa.

**O que é a abstracção procedimental?**

A abstracção procedimental consiste em dar um nome à sequência de acções que serve para atingir um objectivo, e em utilizar esse nome sempre que desejarmos atingir esse objectivo sem termos que considerar explicitamente cada uma das acções individuais que a constituem. A abstracção procedimental consiste assim em abstrair do modo como os procedimentos realizam as suas tarefas, concentrando-se o programador apenas na tarefa que os procedimentos realizam.

**Considere a afirmação: “os procedimentos que geram processos iterativos são sempre melhores que os procedimentos que geram processos recursivos”.**

**Comente-a tendo em conta os seguintes aspectos: (a) Memória ocupada; (b) Tempo gasto; (c) Facilidade de escrita/leitura do código.**

Esta afirmação não está correcta visto que, dependendo dos problemas, existem casos onde os procedimentos que geram processos recursivos são uma melhor opção para a implementação de uma solução. Para a escolha de qual o melhor procedimento, é preciso, entre outros, ter em conta os seguintes factores: memória ocupada, tempo gasto, facilidade de escrita/leitura do código.

Nota: Vamos considerar que estamos a comparar procedimentos com o mesmo grau de complexidade.

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 interpretador mantém informação “escondida” que regista o ponto onde está o processo na cadeia de operações adiadas.

Um processo iterativo não cresce nem contrai. Este é caracterizado por um conjunto de variáveis de estado cuja evolução é ditada pelo conjunto de regras dadas no procedimento. As variáveis de estado fornecem uma descrição completa do estado do processo em cada ponto.

Em termos de espaço de memória ocupada no computador, o procedimento que gerar processos recursivos revelar-se-á uma pior escolha pois constrói uma cadeia de operações adiadas e por tal, tem de manter toda a informação associada a essa cadeia de operações adiadas em memória. Logo, este facto implica ocupar mais memória.

Quanto ao tempo gasto, por definição, somente no limite um procedimento que gere processos recursivos terá um tempo de execução semelhante aos procedimentos que geram processos iterativos. Tal deve-se ao facto da execução das operações pendentes consumir tempo. Por este motivo quase sempre um procedimento que gere processos iterativos terá uma melhor performance.

O caso inverte-se quanto ao terceiro factor analisado: a facilidade de escrita/leitura do código. Um procedimento que gere processos recursivos, em geral, apresenta-se como sendo mais intuitivo, funcionando de uma forma semelhante ao raciocínio humano. Mais ainda, como foi estudado na Torre de Hanoi, a passagem de um processo recursivo para um processo iterativo não é nada trivial. Este exemplo mostra como a recursão em árvore pode dar origem a procedimentos que são fáceis de escrever e compreender.

**Explique a distinção entre representação interna e representação externa de um elemento de um tipo de informação. Apresente um exemplo de cada uma das representações para um tipo à sua escolha.**

Representação interna é a usada pelo Scheme para representar os elementos de um tipo. Representação externa é a usada pelo Scheme para mostrar os elementos de um tipo ao exterior.

Exemplo: tipo complexo

Representação interna: par

Representação externa:  $a + bi$

**Explique os seguintes conceitos: algoritmo, procedimento, processo gerado por um procedimento.**

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. Deve ser rigoroso, deve ser eficaz e deve terminar.

Um procedimento é um algoritmo escrito numa linguagem de programação que neste caso é o Scheme), que se traduz num conjunto de instruções para alcançar um objectivo.

Um processo computacional é originado pela execução de um procedimento pelo interpretador de Scheme. É um ente imaterial que existe dentro de um computador e cuja evolução ao longo do tempo é ditada por esse procedimento.

**Qual o interesse de estudar a ordem de crescimento de um processo? Compare as ordens de crescimento (temporal e espacial) entre processos recursivos lineares e iterativos lineares.**

Tem interesse estudar a ordem de crescimento de um processo visto que esta traduz uma medida grosseira dos recursos requeridos pelo processo à medida que os seus dados se tornam maiores. Os processos recursivos lineares têm ordens de crescimento lineares para o espaço e para o tempo, enquanto que os processos iterativos lineares têm ordem de crescimento constante para o espaço e linear para o tempo.

**Diga quais as fases seguidas na criação de um tipo abstracto de informação e o que se faz em cada uma delas.**

1<sup>a</sup> Fase: Identificação das operações básicas, i.e., do número mínimo de operações que permitem caracterizar o tipo. Estas operações dividem-se em:

1• Construtores: Constroem novos elementos do tipo.

2• Selectores: Acedem aos constituintes dos elementos do tipo.

3• Reconhecedores: Identificam se um dado objecto é do tipo em questão ou se um dado objecto do tipo apresenta certas características especiais.

4• Testes: Comparam elementos do tipo.

2<sup>a</sup> Fase: Axiomatização das operações básicas. Especifica-se o modo como as operações básicas se relacionam entre si.

3<sup>a</sup> Fase: Escolha da representação interna para os elementos do tipo. Essa escolha é feita em termos de outros tipos existentes.

4<sup>a</sup> Fase: Concretização das operações básicas para a representação escolhida. Trata-se da definição (numa linguagem de programação) das operações básicas em termos da representação interna escolhida e de forma a obedecer à axiomatização.

**Explique a razão porque o procedimento primitivo and é uma forma especial.**

O procedimento primitivo "and" é uma forma especial porque tem as suas próprias regras de avaliação. Os seus argumentos (condições) são avaliados pela ordem em que aparecem e assim que a avaliação de um deles resulte em #f (falso), mais nenhum argumento é avaliado e é devolvido o valor #f. Se nenhum

dos argumentos for falso então todos os argumentos são avaliados e o valor devolvido é #t (verdadeiro).

**Diga em que consiste a abstracção de dados. Compare a abstracção de dados com a abstracção procedural.**

A abstracção de dados consiste na separação entre as partes do programa que lidam com o modo como os dados são utilizados das partes do programa que lidam com o modo como os dados são representados.

A abstracção procedural consiste em dar um nome a uma sequência de acções que serve para atingir um objectivo, e em utilizar esse nome sempre que desejarmos atingir esse objectivo sem termos que considerar explicitamente cada uma das acções individuais que a constituem.

Na prática, estas metodologias têm em comum as características da abstracção, permitindo:

- 1• o aumento do nível conceptual em que desenvolvemos os nossos programas;
- 2• o aumento da modularidade dos programas, o que implica um aumento significativo na facilidade de concepção, alteração e manutenção dos programas;
- 3• o aumento do poder expressivo da nossa linguagem.

**Explique em que consiste a abstracção de dados, usando os termos: barreiras de abstracção, encapsulamento da informação e anonimato da representação.**

A abstracção de dados consiste na separação entre as partes do programa que lidam com o modo como os dados são utilizados das partes do programa que lidam com o modo como os dados são representados.

Esta separação é assegurada pelas barreiras de abstracção, que impedem qualquer acesso aos elementos do tipo que não seja feito através das operações básicas.

Considera-se que esta separação garante o anonimato da representação, na medida em que o módulo correspondente aos procedimentos do tipo guarda como segredo o modo escolhido para representar os elementos do tipo.

Por outro lado, este módulo é constituído por um conjunto de procedimentos que representam o tipo de informação e englobam toda a informação referente ao tipo - encapsulação da informação.

**Diga o que se entende por uma definição recursiva e quais são as partes que constituem tal definição.**

Uma definição diz-se recursiva se está definida em termos de si própria, mais concretamente em termos de uma versão mais simples de si própria.

As partes de uma versão recursiva são:

- 1• parte básica ou caso terminal a qual constitui a versão mais simples do problema para o qual a solução é conhecida.
- 2• parte recursiva ou caso geral na qual o problema é definido em termos de uma versão mais simples de si próprio.

**Explique as vantagens da utilização da teoria dos tipos abstractos de informação na definição de novos tipos de informação.**

A teoria dos tipos abstractos de informação (TAIs) permite separar as propriedades abstractas de um tipo do modo como ele é realizado numa linguagem de programação. Esta separação traz algumas vantagens, nomeadamente:

- 1• a melhoria da tarefa de desenvolvimento de programas visto que permite a divisão do programa em vários "módulos". Cada módulo pode corresponder à implementação de um tipo abstracto de informação, e respeitando as barreiras de abstracção subjacentes a esta teoria, em cada momento apenas existem preocupações ao nível do módulo que estamos a implementar (abstracção de dados).
- 2• aumenta a facilidade de leitura de um programa uma vez que tornam a linguagem de programação usada mais expressiva, na medida em que permite a utilização dos novos tipos como se fossem tipos pré-definidos.

II

**Distinga entre o conceito de procedimento e o conceito de processo computacional. Qual a relação entre estes dois conceitos?**

Um processo computacional é um ente imaterial que existe dentro de um computador e cujo comportamento, que evolui ao longo do tempo, é ditado por um programa. Um programa é um algoritmo escrito numa linguagem de programação, e é composto por um ou vários procedimentos. Assim, o procedimento descreve a evolução local do processo computacional.

**Diga o que se entende por processo recursivo e por procedimento recursivo.**

É um procedimento recursivo porque se chama a si próprio. Gera um processo recursivo porque deixa "computações penduradas": antes de poder aplicar o procedimento cons aos seus argumentos, tem que calcular o valor da chamada recursiva. Isto significa que a memória ocupada pelo processo vai aumentar progressivamente à medida que se vão deixando computações pendentes; e depois diminuir progressivamente, à medida que as chamadas recursivas vão retornando o seu valor e as computações que estavam pendentes vão sendo efectuadas.

**Um enquadramento é constituído por uma tabela de ligações, as quais associam um nome a um valor.**

**a) Explique o que é um ambiente associado a um enquadramento.**

O ambiente associado a um enquadramento é a sequência de enquadramentos constituída por esse enquadramento e por todos os seus enquadramentos envolventes.

**b) Qual é o valor de uma variável em relação a um enquadramento?**

O valor de uma variável em relação a um enquadramento é o valor da ligação dessa variável no primeiro enquadramento que contém uma ligação para essa variável.

**Diga o que é abstracção procedural e qual a sua vantagem.**

A abstracção procedural consiste em dar um nome a uma sequência de acções que serve para atingir um objectivo, e em utilizar esse nome sempre que desejamos atingir esse objectivo sem termos que considerar explicitamente cada uma das acções individuais que a constituem. Isto significa que sabemos o que um procedimento faz mas não nos interessa saber como o faz. Por outro lado, a nível do desenvolvimento de programas, podemos construir procedimentos complexos sem nos preocuparmos com os procedimentos mais simples, que são idealizados como caixas pretas.

**Diga o que se entende por uma definição recursiva e quais são as partes que constituem tal definição.**

Uma definição diz-se recursiva se está definida em termos de si própria, mais concretamente em termos de uma versão mais simples de si própria. As partes de uma versão recursiva são:

- parte básica ou caso terminal a qual constitui a versão mais simples do problema para o qual a solução é conhecida.
- parte recursiva ou caso geral na qual o problema é definido em termos de uma versão mais simples de si próprio.

**Qual o interesse de estudar a ordem de crescimento de um processo? Compare as ordens de crescimento (temporal e espacial) entre processos recursivos lineares e iterativos lineares.**

Tem interesse estudar a ordem de crescimento de um processo visto que este traduz uma medida grosseira dos recursos requeridos pelo processo à medida que os seus dados se tornam maiores. Os processos recursivos lineares têm ordens de crescimento lineares para o espaço e para o tempo, enquanto que os processos iterativos lineares têm ordem de crescimento constante para o espaço e linear para o tempo.

**Escreva as regras de avaliação seguidas pelo Scheme, usando o modelo de ambientes, para determinar o valor de uma expressão. Ignore formas especiais.**

A avaliação com base em ambientes, ignorando formas especiais, segue as seguintes regras:

1. O valor de uma constante é a própria constante.
2. O valor de uma variável num ambiente é dado pela ligação da variável no primeiro enquadramento contendo essa variável.
3. Para avaliar uma combinação, avaliam-se as sub-expressões na combinação (por qualquer ordem) e aplica-se o objecto procedural correspondente à primeira sub-expressão às restantes sub-expressões.
4. Para aplicar um objecto procedural a um conjunto de argumentos:
  - a. Cria-se um novo enquadramento;
  - b. Este enquadramento é envolvido pelo enquadramento referido no objecto procedural;
  - c. No novo enquadramento criam-se as ligações entre os parâmetros formais e os parâmetros reais;
  - d. Avalia-se o corpo do procedimento no novo ambiente.

**Explique as vantagens da utilização da teoria dos tipos de abstractos de informação na definição de novos tipos de informação.**

A teoria dos tipos abstractos de informação (TAl) permite separar as propriedades abstractas de um tipo do modo como ele é realizado numa linguagem de programação. Esta separação traz algumas vantagens, nomeadamente:

- a melhoria da tarefa de desenvolvimento de programas visto que permite a divisão do programa em vários "módulos". Cada módulo pode corresponder à implementação de um tipo abstracto de informação, e respeitando as barreiras de abstracção subjacentes a esta teoria, em cada momento apenas existem preocupações ao nível do módulo que estamos a implementar (abstracção de dados).
- aumenta a facilidade de leitura de um programa uma vez que tornam a linguagem de programação usada mais expressiva, na medida em que permite a utilização dos novos tipos como se fossem tipos pré-definidos.
- e finalmente, torna os programas independentes da representação escolhida para os tipos de informação, já que separa as partes do programa que lidam com a forma como os dados são utilizados, das partes do programa que lidam com a forma como os dados são representados.

**Distinga entre programação imperativa e programação funcional. Diga quais são as vantagens e inconvenientes de cada uma delas.**

O que distingue a programação funcional da programação imperativa é a utilização do operador de atribuição. Assim, ao passo que na programação funcional os procedimentos podem ser considerados como métodos de computação de funções matemáticas, uma vez que a avaliação de uma expressão conduz sempre ao mesmo resultado, na programação imperativa isto não acontece. A programação imperativa baseia-se no conceito de efeito, e um programa é considerado uma sequência de instruções, cada uma das quais produzindo um efeito. Um efeito corresponde à alteração de qualquer entidade associada a um processo computacional. Como vantagens da programação funcional temos a simplicidade do modelo de avaliação subjacente, uma vez que as variáveis são apenas associações entre nomes e valores, não sendo estes alteráveis. É este aspecto a sua maior desvantagem: com a imutabilidade das variáveis não é possível a representação de entidades que evoluem ao longo do tempo. A programação imperativa tem a vantagem de poder representar aquelas entidades, à custa da maior complexidade do modelo de avaliação a ela subjacente.

**Uma das fases de desenvolvimento de um programa consiste na programação da solução, que inclui uma fase de depuração.**

**a) Defina erros sintácticos e erros semânticos.**

Os erros sintácticos resultam da não conformidade de uma expressão com as regras sintácticas de uma linguagem de programação. Os erros semânticos resultam do facto de um programa, sintaticamente correcto, ter um significado para o computador que é diferente do significado que o programador desejava que ele tivesse.

**b) Identifique um erro sintáctico e um erro semântico no seguinte procedimento:**  
**(define (factorial x) if (= x 0) 0 (\* x (factorial (- x 1))))**

Erro sintáctico - falta abrir um parêntesis antes do if e consequentemente fechar um parêntesis no final da expressão. Erro semântico - o caso terminal ( $= x 0$ ) deve devolver 1, visto que  $0 \neq 1$  (se assim não fosse, este caso terminal levaria a que o factorial de qualquer inteiro positivo fosse 0). Depois de corrigidos estes erros, obteríamos a expressão: (define (factorial x) (if ( $= x 0$ ) 1 (\* x (factorial (- x 1)))))

**Diga para que serve o processo de recolha de lixo (“garbage collection”) e descreva os passos envolvidos.**

O processo de recolha de lixo serve para devolver o lixo (porção de memória que deixou de ser necessária a um enquadramento) ao amontoado (porção de memória disponível). O processo executa-se em duas fases sequenciais: 1. fase de marcação: as porções de memória acessíveis a partir do ambiente global ou de algum enquadramento existente são marcadas como não sendo lixo; 2. fase de varrimento – toda a memória inicialmente existente no amontoado é percorrida, e todas as zonas de memória que não foram marcadas na fase anterior são devolvidas ao amontoado.

**Diga qual é o tipo de domínio usado em Scheme, estático ou dinâmico, e o que isso significa.**

O tipo de domínio usado em Scheme é chamado domínio estático: o domínio de um nome é definido em termos da estrutura do programa (o encadeamento dos seus blocos) e não é influenciado pelo modo como a execução do programa é feita.

**Enuncie e descreva sucintamente os passos que devem ser seguidos no desenvolvimento de um programa (uma frase para cada passo).**

O desenvolvimento de programas pode ser separado em 5 fases distintas: A análise do problema – pretende-se determinar claramente quais as especificações do problema e saber exactamente quais os objectivos a atingir. O desenvolvimento da solução – descreve-se um algoritmo que constitui a solução do problema a resolver. A programação da solução – codifica-se o algoritmo na linguagem de programação mais adequada, faz-se a depuração sintáctica e semântica, e finaliza-se a documentação técnica e de utilização. A fase de testes – testa-se o algoritmo, para verificar a sua correcção e robustez. A manutenção – depois de terminado o programa, corrigem-se novos erros que apareçam e procede-se a alterações resultantes de modificações na especificação do problema. Em todas estas fases é produzida documentação que descreve as decisões tomadas e serve de apoio ao desenvolvimento das fases subsequentes.

**Quando é que se diz que um objecto tem estado? O que caracteriza o estado de um objecto?**

Um objecto tem estado se o seu comportamento é influenciado pela sua história. O estado de um objecto é caracterizado por uma ou mais variáveis de estado, as quais mantêm informação sobre a sua história, de modo a poder determinar o comportamento do objecto.

**Na implementação de um tipo abstracto de informação, há dois tipos de representação a considerar. Diga quais são e descreva cada um deles.**

Na implementação de um TAI temos a considerar a representação interna e a representação externa. A representação interna corresponde à representação manipulada pelo interpretador de Scheme. A representação externa corresponde ao que é mostrado pelo Scheme ao mundo exterior.

**Diga quais os tipos de documentação de um programa e o seu objectivo.**

**Descreva sucintamente em que deve consistir cada um dos tipos de documentação.**

Num programa temos a considerar a documentação de utilização e a documentação técnica. A documentação de utilização tem a finalidade de fornecer ao utilizador a informação necessária para a utilização do programa: o que o programa faz, o processo de utilização, a informação necessária para o bom funcionamento, a informação produzida pelo programa e as suas limitações (em termos não técnicos). A documentação técnica fornece ao programador que irá modificar o programa a informação necessária para a sua compreensão. É constituída pela documentação externa, que descreve os algoritmos e procedimentos (e eventualmente os TAI's) e a documentação interna, que consiste nos comentários do programa.

**Diga o que entende por linguagem estruturada em blocos. Descreva a regra associada a esta estrutura, e diga qual a sua importância.**

Uma linguagem estruturada em blocos baseia-se na definição de blocos, em que um bloco corresponde a um conjunto de instruções. A regra que está associada a esta estrutura é a abstracção procedural. A abstracção procedural consiste em abstrair do modo como os procedimentos realizam as suas tarefas, concentrando-se apenas na tarefa que os procedimentos realizam. Ou seja, a separação do "como" de "o que". Permite ainda modularizar um programa e controlar a sua complexidade. Por outro lado, garante uma protecção efectiva da informação definida em cada bloco, não autorizada por parte de outros blocos.

**O que é a depuração da base para o topo? Quais as suas vantagens?**

A depuração da base para o topo consiste em testar, em primeiro lugar, os módulos (por módulo entenda-se um procedimento correspondente a um sub problema) que estão ao nível mais baixo, isto é, que não são decomponíveis em sub problemas, 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.

**Em que consiste a programação imperativa?**

A programação imperativa baseia-se no conceito de efeito. Em programação imperativa um programa é considerado como uma sequência de instruções, cada uma das quais produz um efeito.

**Diga o que é um ponteiro. Qual a característica que distingue um ponteiro dos outros tipos de informação?**

Um ponteiro é um objecto computacional que aponta. Com a utilização de ponteiros, não estamos interessados no valor do ponteiro mas sim no valor para onde ele aponta.

**Distinga o comportamento FIFO (“first in first out”) do comportamento LIFO (“last in first out”). Quais os tipos de informação que correspondem a cada um destes comportamentos?**

“FIFO” designa o tipo de comportamento em que o primeiro elemento a entrar numa estrutura de informação será também o primeiro elemento a sair. As filas são a estrutura de informação que seguem este tipo de comportamento. De forma diferente, “LIFO” designa o tipo de comportamento em que o último elemento a entrar numa estrutura de informação será o primeiro elemento a sair. As pilhas são a estrutura de informação que seguem este tipo de comportamento.

**Explique a diferença entre um objecto e um procedimento com estado interno.**

Um procedimento com estado interno comporta variáveis “internas” ao procedimento, sendo este a única forma possível de aceder e modificar essas variáveis. De forma diferente, o conceito de objecto designa uma entidade que contém não só um estado interno, mas também um conjunto de procedimentos, denominados métodos, que manipulam esse estado interno.