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 procedimental e temos utilizado a abstração procedimental 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 caraterizada 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 carateriza 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 procedimental para a criação de funções.

Usando a abstração procedimental, 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 procedimental permite separar o modo como uma função é utilizada do modo como essa função é realizada.

O conceito equivalente à abstração procedimental 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 procedimental, 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 caraterizado 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 $a + b\,i$, 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[1]. Sabemos também que a soma, subtração, multiplicação e divisão de números complexos são definidas do seguinte modo:

$$(a + b\,i) + (c + d\,i) = (a + c) + (b + d)\,i$$

$$(a + b\,i) - (c + d\,i) = (a - c) + (b - d)\,i$$

$$(a + b\,i).(c + d\,i) = (a\,c - b\,d) + (a\,d + b\,c)\,i$$

$$\frac{(a + b\,i)}{(c + d\,i)} = \frac{a\,c + b\,d}{c^2 + d^2} + \frac{b\,c - a\,d}{c^2 + d^2}\,i$$

[1] 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 $a + b\,i$ será representado pelo tuplo `(`a`, `b`)`. Esse programador, tendo duas variáveis que correspondem a números complexos, `c1` e `c2`, irá naturalmente escrever a seguinte expressão para somar esses números, dando origem ao número complexo `c3`:

```
c3 = (c1[0] + c2[0], c1[1] + c2[1]).
```

Embora a expressão anterior esteja 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[2] como $a + b\,i$. 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).

[2]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 seja representado por um dicionário. Podemos dizer que o número complexo $a + b\,i$ 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, $\Re[a + b\,i] = \{$`'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*[3] 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.

[3]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 caraterizar 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`[4] que reconhecem números complexos com certas caraterísticas;

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`[5] que decide se dois números complexos são iguais.

O papel das operações básicas é construírem 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 inerentemente mutáveis, por exemplo as contas bancárias apresentadas no Capítulo 11, e outros que são inerentemente imutáveis, por exemplo os números complexos por corresponderem 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

[4] Nenhuma destas operações foi ainda por nós definida nem realizada.
[5] 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*:
   - $cria\_compl : real \times real \mapsto complexo$
     $cria\_compl(r, i)$ tem como valor o número complexo cuja parte real é r e cuja parte imaginária é i.
2. *Seletores*:
   - $p\_real : complexo \mapsto real$
     $p\_real(c)$ tem como valor a parte real do complexo c.
   - $p\_imag : complexo \mapsto real$
     $p\_imag(c)$ tem como valor a parte imaginária do complexo c.
3. *Reconhecedores*:
   - $e\_complexo : universal \mapsto lógico$
     $e\_complexo(arg)$ tem o valor $verdadeiro$ se arg é um número complexo e tem o valor $falso$ em caso contrário.
   - $e\_compl\_zero : complexo \mapsto lógico$
     $e\_compl\_zero(c)$ tem o valor $verdadeiro$ se c é o complexo $0 + 0i$ e tem o valor $falso$ em caso contrário.
   - $e\_imag\_puro : complexo \mapsto lógico$
     $e\_imag\_puro(c)$ 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*:

   - $compl\_iguais : complexo \times complexo \mapsto lógico$
     $compl\_iguais(c_1, c_2)$ tem o valor $verdadeiro$ se c_1 e c_2 correspondem ao mesmo número complexo e tem o valor $falso$ em caso contrário.

Assim, a assinatura do tipo complexo é:

$$cria\_compl : real \times real \mapsto complexo$$

$$p\_real : complexo \mapsto real$$

$$p\_imag : complexo \mapsto real$$

$$e\_complexo : universal \mapsto lógico$$

$$e\_compl\_zero : complexo \mapsto lógico$$

$$e\_imag\_puro : complexo \mapsto lógico$$

$$compl\_iguais : complexo \times complexo \mapsto lógico$$

Deveremos ainda definir uma *representação externa* para complexos, por exemplo, podemos definir que a notação correspondente aos elementos do tipo complexo é da forma $a + b\,i$, 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 $a + b\,i$.

Ao especificarmos as operações básicas e os transformadores de entrada e de saída para um dado tipo, estamos a criar uma extensão concetual 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[6]:

$$e\_complexo(cria\_compl(r,i)) = verdadeiro$$

$$e\_compl\_zero(c) = compl\_iguais(c, cria\_compl(0,0))$$

$$e\_imag\_puro(cria\_compl(0,b)) = verdadeiro$$

$$p\_real(cria\_compl(r,i)) = r$$

$$p\_imag(cria\_compl(r,i)) = i$$

$$cria\_compl(p\_real(c), p\_imag(c)) = \begin{cases} c & \text{se } e\_complexo(c) \\ \perp & \text{em caso contrário} \end{cases}$$

$$compl\_iguais(cria\_compl(x,y), cria\_compl(w,z)) = (x = w) \wedge (y = z)$$

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 $a + b\,i$ é representado por um dicionário em que o valor da chave `'r'` é a parte real e o valor da chave `'i'` é a parte imaginária, assim $\Re[a+b\,i] = \{$`'r'`$:\Re[a]$`, 'i':`$\Re[b]\}$.

[6] 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[7]. Por esta razão, a função `zero` poderá ser definida como:

[7] 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, caraterizam 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 concetual 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.

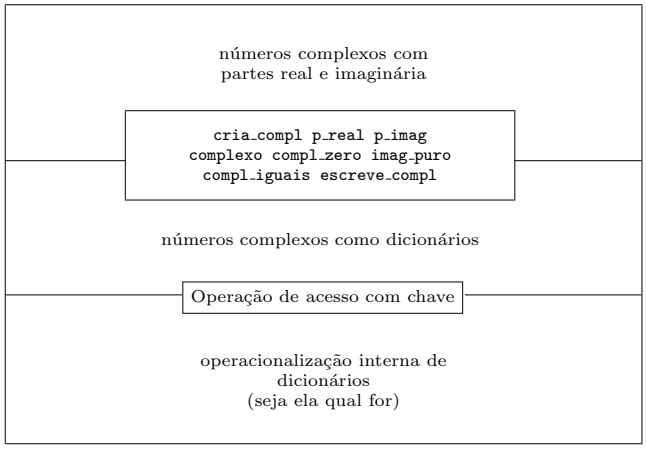

Figura 9.1: Camadas de abstração na definição de complexos.

Concetualmente, 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, escrevíamos 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, decidíamos 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 caraterizam 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

1. 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*:
     $vetor : real \times real \mapsto vetor$
     $vetor(x, y)$ tem como valor o vetor cuja extremidade é o ponto (x, y).
   - *Seletores*:
     $abcissa : vetor \mapsto real$
     $abcissa(v)$ tem como valor a abcissa da extremidade do vetor v.
     $ordenada : vetor \mapsto real$
     $ordenada(v)$ tem como valor a ordenada da extremidade do vetor v.

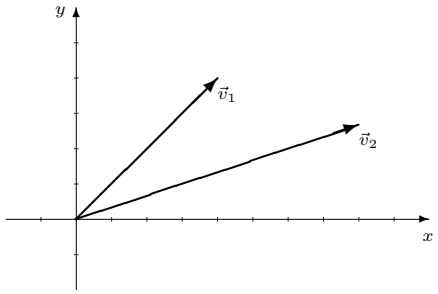

Figura 9.2: Exemplo de vetores.

- *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.

   (a) Especifique as operações básicas do tipo `ponto`.

   (b) Implemente o tipo `ponto`.

   (c) Escreva uma função em Python que recebe duas variáveis do tipo `ponto` e que determina a distância entre esses pontos. A distância entre os pontos (x_1, y_1) e (x_2, y_2) é dada por $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

   (d) Escreva uma função em Python que recebe como argumento um `ponto` e que determina o quadrante em que este se encontra. A sua função deve devolver um inteiro entre 1 e 4.

4. Suponha que quer representar o tempo, dividindo-o em horas e minutos.

   (a) 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.

   (b) Com base no tipo `tempo`, escreva as seguintes funções:

   - $depois : tempo \times tempo \mapsto lógico$
     $depois(t_1, t_2)$ tem o valor *verdadeiro*, se t_1 corresponder a um instante de tempo posterior a t_2.
   - $num\_minutos : tempo \mapsto inteiro$
     $num\_minutos(t)$ tem como valor o número de minutos entre o momento 0 horas e 0 minutos e o tempo t.

5. (a) Especifique as operações básicas do tipo abstrato `carta` o qual é caraterizado 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).

   (b) Implemente o tipo `carta`.

   (c) Usando o tipo `carta`, defina uma função em Python que devolve uma lista em que cada elemento corresponde a uma carta de um baralho.

   (d) 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 caraterizado 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 operacões, 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`.