Capítulo 3

Funções

'Can you do Addition?' the White Queen asked. 'What's one and one and one and one and one and one and one and one and 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 $\{(1,1),(2,4),(3,9),\ldots\}$.

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 compreende 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[1]:

⟨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;

[1] A ⟨definição de função⟩ corresponde em Python a uma ⟨definição⟩ (ver página 28).

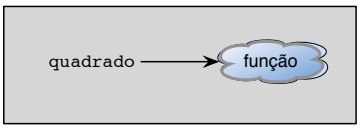

Figura 3.1: Ambiente com o nome `quadrado`.

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)
49
```

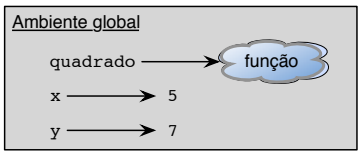

Figura 3.2: Ambiente global com os nomes `quadrado`, `x` e `y`.

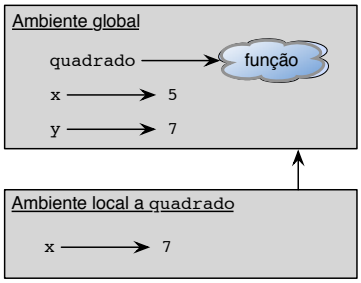

Figura 3.3: Ambiente criado durante a avaliação de `quadrado(y)`.

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 procedimental

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 procedimental* 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 procedimental* 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 procedimental 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 procedimental consideramos 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 procedimental.

A abstração procedimental 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 minima?
? 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 Potência

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

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

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

Com esta função, podemos gerar a seguinte interação:

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

3.4.3 Fatorial

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

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

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

Com esta função obtemos a interação:

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

[2]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 2 300 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?

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

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.)[3]. 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:

$$p_{i+1} = \frac{p_i + \frac{x}{p_i}}{2}. \tag{3.1}$$

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[4], 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, palpite):
    while not bom_palpite(x, palpite):
```

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

[4]Na realidade, para o nosso exemplo, este é um número irracional.

```
        palpite = novo_palpite(x, palpite)
    return palpite
```

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*[5]. 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, palpite):
    return (palpite + x / palpite) / 2
```

Resta-nos decidir quando estamos perante um bom palpite. É evidente que a satisfação com dado palpite vai depender do 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.

[5]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[6]. Utilizando esta medida, diremos que p_i é uma boa aproximação de $\sqrt{x}$ se $\mid (p_i)^2 - x \mid < \delta$, em que δ é um valor suficientemente pequeno. Supondo que δ corresponde ao nome `delta`, o qual define o limiar de aproximação exigido pela nossa aplicação, podemos escrever a função:

```
def bom_palpite(x, 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
```

[6]Uma outra alternativa corresponde a considerar o *erro relativo*, o quociente entre o erro absoluto e o valor correto, $\mid (p_i)^2 - x \mid /x$.

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

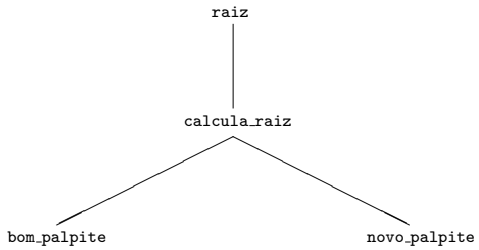

Figura 3.5: Funções associadas ao cálculo da raiz.

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 procedimental.

Sob esta perspetiva, 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 α, define-se $sin(\alpha)$ como sendo a razão entre o cateto oposto e a hipotenusa deste triângulo. Para calcular o valor de *seno* podemos utilizar o desenvolvimento em série de Taylor o qual fornece o valor de $sin(x)$ para um valor de x em radianos:

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

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.[7] Na segunda linha da função `calc_termo` usamos o símbolo "\", o *símbolo de continuação*, que indica ao Python que a linha que estamos as 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):
```

[7] 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
`pi`	π	Uma aproximação de π
`e`	e	Uma aproximação de e
`sin(x)`	$sen(x)$	O seno de x (x em radianos)
`cos(x)`	$cos(x)$	O coseno de x (x em radianos)
`tan(x)`	$tg(x)$	A tangente de x (x em radianos)
`log(x)`	$ln(x)$	O logaritmo natural de x
`exp(x)`	e^x	A função inversa de ln
`pow(x, y)`	x^y	O valor de x levantado a y
`sqrt(x)`	$\sqrt{x}$	A raiz quadrada de x
`ceil(x)`	$\lceil x \rceil$	O maior inteiro superior ou igual a x
`floor(x)`	$\lfloor x \rfloor$	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

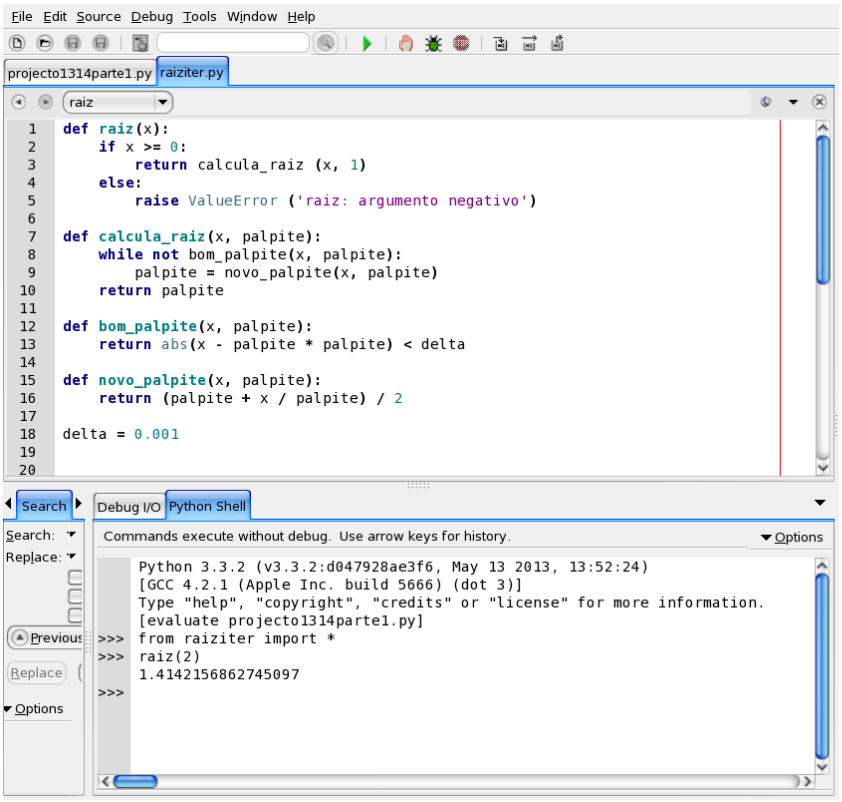

Figura 3.6: Utilização do módulo `raiziter` no Wing 101.

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 procedimental, 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 `bissexto` 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*[8] 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 comos 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

[8] [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 $n = 1 + 2 + \ldots + (m-1) + m$. 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

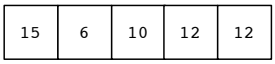

Figura 4.1: Representação gráfica de um tuplo.

referirmos aos elementos de um vetor, utilizamos índices que caraterizam univocamente estes elementos. Por exemplo, se $\vec{x}$ representa um vetor com três elementos, estes são caraterizados, 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[1] é definida sintaticamente pelas seguintes expressões em notação BNF[2, 3]:

⟨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-

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

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

[3] As definições de ⟨lista⟩ e ⟨dicionário⟩ são apresentadas, respetivamente, nos capítulos 5 e 7.

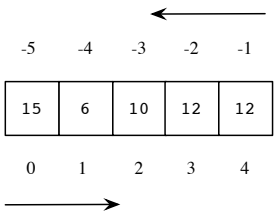

Figura 4.2: Valores dos índices de um tuplo.

plo 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:

⟨nome indexado⟩ ::= ⟨nome⟩ `[`⟨expressão⟩`]`

em que ⟨nome⟩ corresponde ao nome do tuplo e ⟨expressão⟩ (que é do tipo inteiro[4]) 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)
```

[4]Este aspeto 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 sobrecarregadas, pois também são aplicáveis a inteiros e a reais.

Operação	*Tipo dos argumentos*	*Valor*
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 `in` t	Universal e tuplo	`True` se o elemento e pertence ao tuplo t; `False` em caso contrário.
e `not in` t	Universal e tuplo	A negação do resultado da operação e `in` t.
`tuple(`a`)`	Lista ou dicionário ou cadeia de carateres	Transforma o seu argumento num tuplo. Se não forem fornecidos argumentos, devolve o tuplo vazio.
`len(`t`)`	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 $\overrightarrow{v} = c_1\overrightarrow{e_x} + c_2\overrightarrow{e_y} + c_3\overrightarrow{e_z}$ pode ser representado pelo tuplo (`c1, c2, c3`). 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*[5] é um ciclo cuja execução é controlada por uma variável, designada por *variável de controle*. Para a variável de controle é especificado o seu valor inicial, a forma de 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[6]:

⟨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

[5] Em inglês, "counted loop".
[6] 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, ...,` $e_1 - 1$`)`. 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`,` $e_1 + 1$`, ...,` $e_2 - 1$`)`. 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`,` $e_1 + e_3$`,` $e_1 + 2 \cdot e_3$`, ...)`. 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 caraterísticas 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[7]. 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`.

[7] 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`[8], 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[9] e os carateres de uma cadeia de carateres delimitadas por aspas não podem conter aspas[10]. As cadeias de carateres `''` e `""` são chamadas *cadeias de carateres vazias*. Ao longo do livro utilizamos as plicas para delimitar as cadeias de carateres.

[8] Da sua designação em inglês, "string".
[9] Exceto se recorrermos ao caráter de escape apresentado na Tabela 2.8.
[10] *Ibid.*

As cadeias de carateres delimitadas por três aspas, chamadas *cadeias de carateres de documentação*[11], 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.

[11] Do inglês, "docstring".

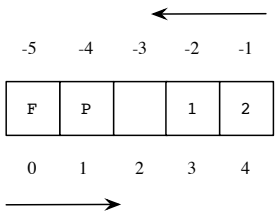

Figura 4.3: Valores dos índices de uma cadeia de carateres.

Os elementos das cadeias de carateres 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 carateres `'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 carateres com apenas um elemento. É importante notar que `'2'` não é o mesmo que o inteiro `2`, é o caráter "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 carateres podemos efetuar as operações indicadas na Tabela 4.3[12] (note-se que estas operações são semelhantes às apresentadas na Tabela 4.1 e, consequentemente, todas estas operações são sobrecarregadas). A seguinte interação mostra a utilização de algumas destas operações:

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

[12]Nesta tabela, "Universal", designa qualquer tipo.

Operação	*Tipo dos argumentos*	*Valor*
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 `in` s	Cadeias de carateres	`True` se e pertence à cadeia de carateres s; `False` em caso contrário.
e `not in` s	Cadeias de carateres	A negação do resultado da operação e `in` s.
`len(`s`)`	Cadeia de carateres	O número de elementos da cadeia de carateres s.
`eval(`s`)`	Cadeia de carateres	Avalia a cadeia de carateres s como se fosse uma expressão.
`str(`a`)`	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,[13] 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 digitos. 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 $165 \ mod \ 11 = 0$. 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

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

Phyton 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 carateres `s1` contiver carateres repetidos e estes carateres existirem em `s2`, a função `simbolos_comum` apresenta um resultado com carateres repetidos como o mostra a interação anterior. Podemos modificar a nossa função de modo a que esta não apresente carateres 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 carateres são representados dentro do computador associados a um código numérico. Embora tenham sido concebidos vários códigos para a representação de carateres, os computadores modernos são baseados no código ASCII ("American Standard Code for Information Interchange")[14], o qual foi concebido para representar os carateres 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 carateres, 33 dos quais correspondem a carateres de controle (muitos dos quais são atualmente obsoletos) e 95 carateres 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 carateres existentes na língua inglesa. Para lidar com a representação de outros carateres,

[14]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
/	47	H	72	`	96	w	119
0	48	I	73	a	97	x	120
1	49	J	74	b	98	y	121
2	50	K	75	c	99	z	122
3	51	L	76	d	100	{	123
4	52	M	77	e	101	|	124
5	53	N	78	f	102	}	125
6	54	O	79	g	103	~	126
7	55						

Tabela 4.4: Carateres ASCII visíveis.

por exemplo carateres acentuados, foi desenvolvida uma representação, conhecida por *Unicode*,[15] 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,

[15] 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[16]. 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
```

[16] 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 agore 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,

$$r(n) = \begin{cases} 0 & \text{se } n = 0 \\ r(n-1) - n & \text{se } r(n-1) > n \ \wedge \ (r(n-1) - n) \notin \{r(i) : 1 < n\} \\ r(n-1) + n & \text{em caso contrário} \end{cases}$$

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 carateres 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 espressão

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

Escreva um programa para validar um ISBN com 13 dígitos.