Capítulo 2

Elementos básicos de programação

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

Lewis Carroll, *Alice's Adventures in Wonderland*

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

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

O Python é uma linguagem de programação, ou seja, corresponde a um formalismo para escrever programas. Um programa em Python pode ser introduzido

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

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

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

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 13 2013, 13:52:24)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)]
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

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

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

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

[1] Do inglês, "prompt character".
[2] Do inglês, "read-eval-print loop".

definido do seguinte modo:

⟨comando⟩ ::= ⟨expressão⟩ | ⟨instrução⟩ | ⟨definição⟩

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

2.1 Expressões

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

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

⟨expressão⟩ ::= ⟨constante⟩ | ⟨expressão composta⟩ | ⟨nome⟩ | ⟨aplicação de função⟩

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

2.1.1 Constantes

Para os efeitos deste capítulo, consideramos que as constantes em Python podem ser números, valores lógicos ou cadeias de carateres. Sempre que é fornecida uma constante ao Python, este devolve a constante como resultado da avaliação. Ou seja, o valor de uma constante é a própria constante (o Python mostra a representação externa da constante). A *representação externa* de uma entidade corresponde ao modo como nós visualizamos essa entidade, independentemente do modo como esta é representada internamente no computador (a *representação interna*), a qual, como sabemos, é feita recorrendo apenas aos símbolos `0` e `1`. Por exemplo, a representação externa do inteiro 1958 é `1958` ao passo que a sua representação interna é `11110100110`.

A seguinte interação mostra a resposta do Python quando lhe fornecemos algu-

mas constantes:

```
>>> 1958
1958
>>> -1
-1
>>> +2
2
>>> 655484877641416186376754588877162243232221200091999228887333
655484877641416186376754588877162243232221200091999228887333
>>> 3.5
3.5
>>> 65397518517617656541959.888666225224423331
6.539751851761766e+22
>>> 0.00000000000000000000000000000000000001
1e-38
>>> True
True
>>> False
False
>>> 'Bom dia'
'Bom dia'
>>> "Bom dia"
'Bom dia'
```

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

1. *Números inteiros.* Estes correspondem a números sem parte decimal (com ou sem sinal) e podem ser arbitrariamente grandes;

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

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

3. *Valores lógicos*. Os quais são representados por `True` (*verdadeiro*) e `False` (*falso*);

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

2.1.2 Expressões compostas

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

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

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

[3] Uma cadeia de carateres é frequentemente designada pelo seu nome em inglês, "string".
[4] Este caráter corresponde ao espaço em branco.
[5] Do inglês, "built-in" operations.

modo[6]:

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

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

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

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

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

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

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

Para evitar ambiguidade em relação à ordem de aplicação dos operadores numa expressão, o Python utiliza duas regras que especificam a ordem de aplicação dos operadores. A primeira regra, associada a uma *lista de prioridades de*

[6] Iremos ver outros modos de definir expressões compostas.

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

Tabela 2.1: Prioridade dos operadores em Python.

operadores, especifica que os operadores com maior prioridade são aplicados antes dos operadores com menor prioridade; a segunda regra especifica qual a ordem de aplicação dos operadores quando se encontram dois operadores com a mesma prioridade. Na Tabela 2.1 apresentamos a lista de prioridades dos operadores em Python (estas prioridades são, de modo geral, adotadas em todas as linguagens de programação). Quando existem dois (ou mais) operadores com a mesma prioridade, eles são aplicados da esquerda para a direita. A utilização de parêntesis permite alterar a ordem de aplicação dos operadores.

2.2 Tipos elementares de dados

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

A utilização de tipos para caraterizar entidades que correspondem a dados é muito importante em programação. Um *tipo de dados*[7] é caraterizado por um *conjunto de entidades* (valores) e um *conjunto de operações* aplicáveis a essas entidades. Ao conjunto de entidades dá-se nome de *domínio do tipo*. Cada uma das entidades do domínio do tipo é designada por *elemento do tipo*.

Os tipos de dados disponíveis variam de linguagem de programação para lin-

[7] Em inglês "data type".

Operação	*Tipo dos argumentos*	*Valor*
e_1 `+` e_2	Inteiros	O resultado de somar e_1 com e_2.
e_1 `-` e_2	Inteiros	O resultado de subtrair e_2 a e_1.
`-`e	Inteiro	O simétrico de e.
e_1 `*` e_2	Inteiros	O resultado de multiplicar e_1 por e_2.
e_1 `//` e_2	Inteiros	O resultado da divisão inteira de e_1 por e_2.
e_1 `%` e_2	Inteiros	O resto da divisão inteira de e_1 por e_2.
`abs(`e`)`	Inteiro	O valor absoluto de e.

Tabela 2.2: Operações sobre números inteiros.

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

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

2.2.1 O tipo inteiro

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

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

[8]Do inglês, "integer".

[9]Em Python um inteiro não pode começar por zero como o mostra a segunda linha da interação.

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

2.2.2 O tipo real

Os números reais, em Python designados por `float`[10], são números com parte decimal. Em Python, e na maioria das linguagens de programação, existem dois métodos para a representação das constantes do tipo real, a notação decimal e a notação científica.

1. A *notação decimal*, em que se representa o número, com ou sem sinal, por uma parte inteira, um ponto (correspondente à vírgula), e uma parte decimal. São exemplos de números reais em notação decimal, `-7.236`, `7.0` e `0.76752`. Se a parte decimal ou a parte inteira forem zero, estas podem ser omitidas, no entanto a parte decimal e a parte inteira não podem ser omitidas simultaneamente. Assim, `7.` e `.1` correspondem a números reais em Python, respetivamente `7.0` e `0.1`;

2. A *notação científica* em que se representa o número, com ou sem sinal, através de uma *mantissa* (que pode ser inteira ou real) e de uma potência inteira de dez (o *expoente*) que multiplicada pela mantissa produz o número. A mantissa e o expoente são separados pelo símbolo "`e`". São exemplos de números reais utilizando a notação científica, `4.2e5` (=`420000.0`), `-6e-8` (=`-0.00000006`). A notação científica é utilizada principalmente para representar números muito grandes ou muito pequenos.

A seguinte interação mostra algumas constantes reais em Python:

```
>>> 7.7
7.7
```

[10] Designação que está associada à representação de números reais dentro de um computador, a representação em *virgula flutuante* (em inglês, "floating point representation").

Operação	*Tipo dos argumentos*	*Valor*
e_1 `+` e_2	Reais	O resultado de somar e_1 com e_2.
e_1 `-` e_2	Reais	O resultado de subtrair e_2 a e_1.
`-`e	Real	O simétrico de e.
e_1 `*` e_2	Reais	O resultado de multiplicar e_1 por e_2.
e_1 `/` e_2	Reais	O resultado de dividir e_1 por e_2.
`abs(`e`)`	Real	O valor absoluto de e.

Tabela 2.3: Operações sobre números reais.

```
>>> 7.
7.0
>>> .4
0.4
>>> 200000000000000000000000000000000000.
2e+35
>>> .00000000000000000000000000000000000000000000000000001
1e-53
```

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

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

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

resultados que seriam de esperar em termos de operações aritméticas.

O que na realidade se passa dentro do computador é que cada operação sobre números (por exemplo, a operação de adição, `+`) corresponde a duas operações internas, uma para cada um dos tipos numéricos (por exemplo, a operação $+_{\mathbb{Z}}$ que adiciona números inteiros produzindo um número inteiro e a operação $+_{\mathbb{R}}$ que adiciona números reais produzindo um número real). Estas operações estão associadas à mesma representação externa (`+`). Quando isto acontece, ou seja, quando a mesma representação externa de uma operação está associada a mais do que uma operação dentro do computador, diz-se que a operação está *sobrecarregada*[11].

Quando o Python tem de aplicar uma operação sobrecarregada, determina o tipo de cada um dos operandos. Se ambos forem inteiros, aplica a operação $+_{\mathbb{Z}}$, se ambos forem reais, aplica a operação $+_{\mathbb{R}}$, se um for inteiro e o outro real, converte o número inteiro para o real correspondente e aplica a operação $+_{\mathbb{R}}$. Esta conversão tem o nome de *coerção*[12], sendo demonstrada na seguinte interação:

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

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

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

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

[11] Do inglês, "overloaded".
[12] Do inglês, "coercion".

Operação	*Tipo do argumento*	*Tipo do valor*	Operação
`round(`e`)`	Real	Inteiro	O inteiro mais próximo do real e.
`int(`e`)`	Real	Inteiro	A parte inteira do real e.
`float(`e`)`	Inteiro	Real	O número real correspondente a e.

Tabela 2.4: Transformações entre reais e inteiros.

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

2.2.3 O tipo lógico

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

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

- As operações unárias produzem um valor lógico a partir de um valor lógico. Existe uma operação unária em Python, `not`. A operação `not` muda o valor lógico, de um modo semelhante ao papel desempenhado pela palavra "não" em português. Assim, `not(True)` tem o valor `False` e `not(False)` tem o valor `True`.

- As operações binárias aceitam dois argumentos do tipo lógico e produzem um valor do tipo lógico. Entre estas operações encontram-se as operações lógicas tradicionais correspondentes à conjunção e à disjunção. A conjunção, representada por `and`, tem o valor `True` apenas se ambos os seus argumentos têm o valor `True` (Tabela 2.5). A disjunção, representada por `or`, tem o valor `False` apenas se ambos os seus argumentos têm o valor `False` (Tabela 2.5).

[13]Em honra ao matemático inglês George Boole (1815–1864).

e_1	e_2	e_1 `and` e_2	e_1 `or` e_2
`True`	`True`	`True`	`True`
`True`	`False`	`False`	`True`
`False`	`True`	`False`	`True`
`False`	`False`	`False`	`False`

Tabela 2.5: Operações de conjunção e disjunção.

2.3 Nomes e atribuição

'Don't stand there chattering to yourself like that,' Humpty Dumpty said, looking at her for the first time,' but tell me your name and your business.'
'My NAME is Alice, but –'
'It's a stupid name enough!' Humpty Dumpty interrupted impatiently. 'What does it mean?'
'MUST a name mean something?' Alice asked doubtfully.
'Of course it must,' Humpty Dumpty said with a sort laugh: 'MY name means the shape I am – and a good handsome shape it is, too. With a name like yours, you might be any shape, almost.'

Lewis Carroll, *Through the Looking Glass*

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

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

⟨instrução de atribuição⟩ ::= ⟨nome⟩ `=` ⟨expressão⟩ |
⟨nome⟩ `,` ⟨instrução de atribuição⟩ `,` ⟨expressão⟩

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

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

⟨nome⟩ ::= ⟨nome simples⟩ |
⟨nome indexado⟩ |
⟨nome composto⟩

Neste capítulo, apenas consideramos ⟨nome simples⟩, sendo ⟨nome indexado⟩ introduzido na Secção 4.1 e ⟨nome composto⟩ introduzido na Secção 3.5.

Os nomes são utilizados para representar entidades usadas pelos programas. Como estas entidades podem variar durante a execução do programa, os nomes são também conhecidos por *variáveis*.

Em Python, um ⟨nome simples⟩ é uma sequência de carateres que começa por uma letra ou pelo caráter _:

⟨nome simples⟩ ::= ⟨inicial⟩ ⟨subsequente⟩*

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

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

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

Comecemos por discutir a primeira alternativa da instrução de atribuição, a qual corresponde à primeira linha da expressão BNF que define ⟨instrução de

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

Tabela 2.6: Nomes reservados em Python.

atribuição⟩, a que chamamos *atribuição simples*.

Ao encontrar uma instrução da forma ⟨nome⟩ = ⟨expressão⟩, o Python começa por avaliar a ⟨expressão⟩ após o que associa ⟨nome⟩ ao valor da ⟨expressão⟩. A execução de uma instrução de atribuição não devolve nenhum valor, mas sim altera o valor de um nome.

A partir do momento em que associamos um nome a um valor (ou nomeamos o valor), o Python passa a "conhecer" esse nome, mantendo uma memória desse nome e do valor que lhe está associado. Esta memória correspondente à associação de nomes a valores (ou, de um modo mais geral, à associação de nomes a entidades computacionais – de que os valores são um caso particular) tem o nome de *ambiente*. Um ambiente (também conhecido por *espaço de nomes*[14]) contém associações para todos os nomes que o Python conhece. Isto significa que no ambiente existem também associações para os nomes de todas as operações embutidas do Python, ou seja, as operações que fazem parte do Python.

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

Consideremos a seguinte interação com o Python:

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

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

[14]Do inglês, "namespace".

Figura 2.1: Representação de um ambiente.

trução, o Python avalia a expressão `17` (uma constante) e atribui o seu valor à variável `nota`. A partir deste momento, o Python passa a "conhecer" o nome, `nota`, o qual tem o valor `17`. A segunda linha da interação anterior corresponde à avaliação de uma expressão e mostra que se fornecermos ao Python a expressão `nota` (correspondente a um nome), este diz que o seu valor é `17`. Este resultado resulta de uma regra de avaliação de expressões que afirma que o valor de um nome é a entidade associada com o nome no ambiente em questão. Na Figura 2.1 mostramos a representação do ambiente correspondente a esta interação. Um ambiente é representado por um retângulo a cinzento, dentro do qual aparecem associações de nomes a entidades computacionais. Cada associação contém um nome, apresentado no lado esquerdo, ligado por uma seta ao seu valor.

Consideremos agora a seguinte interação com o Python, efetuada depois da interação anterior:

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

Segundo a semântica da instrução de atribuição, para a instrução apresentada na primeira linha da interação anterior, o Python começa por avaliar a expressão `nota + 1`, a qual tem o valor 18, em seguida associa o nome `nota` a este valor, resultando no ambiente apresentado na Figura 2.2. A instrução de atribuição `nota = nota + 1` tem o efeito de atribuir à variável `nota` o valor anterior de `nota` mais um. Este último exemplo mostra o caráter dinâmico da operação de atribuição: em primeiro lugar, a expressão à direita do símbolo `=` é avaliada, e, em segundo lugar, o valor resultante é atribuído à variável à esquerda deste símbolo. Isto mostra que uma operação de atribuição não corresponde a uma equação matemática, mas sim a um processo de atribuir o valor da expressão à direita do operador de atribuição à variável à sua esquerda. Na interação

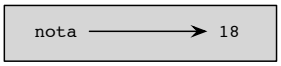

Figura 2.2: Ambiente resultante da execução de `nota = nota + 1`.

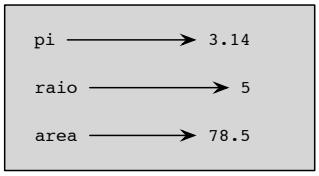

Figura 2.3: Ambiente depois da definição de `pi`, `raio` e `area`.

anterior, mostramos também que se fornecermos ao Python um nome que não existe no ambiente (`soma`), o Python gera um erro, dizendo que não conhece o nome.

Na seguinte interação com o Python, começamos por tentar atribuir o valor `3` ao nome `def`, o qual corresponde a um nome reservado do Python (ver Tabela 2.6). O Python reage com um erro. Seguidamente, definimos valores para as variáveis `pi`, `raio` e `area`, resultando no ambiente apresentado na Figura 2.3.

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

A interação anterior também mostra que se mudarmos o valor da variável `raio` o valor de `area`, embora tenha sido calculado a partir do nome `raio`, não se altera (Figura 2.4).

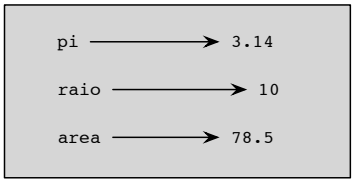

Figura 2.4: Ambiente depois da alteração do valor de `raio`.

Consideremos agora a segunda alternativa da instrução de atribuição, a qual é conhecida por *atribuição múltipla* e que corresponde à segunda linha da expressão BNF que define ⟨instrução de atribuição⟩ apresentada na página 39. Ao encontrar uma instrução da forma

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

o Python começa por avaliar as expressões ⟨exp$_1$⟩, ⟨exp$_2$⟩, ... ⟨exp$_n$⟩ (a ordem da avaliação destas expressões é irrelevante), após o que associa ⟨nome$_1$⟩ ao valor da expressão ⟨exp$_1$⟩, ⟨nome$_2$⟩ ao valor da expressão ⟨exp$_2$⟩, ... ⟨nome$_n$⟩ ao valor da expressão ⟨exp$_n$⟩.

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

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

Consideremos a seguinte interação

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

e analisemos a origem do erro. Dissemos que o Python começa por avaliar as expressões à direita do símbolo `=`, as quais são `17` e `nota_1 + 1`. O valor da

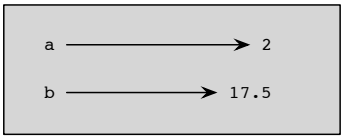

Figura 2.5: Ambiente.

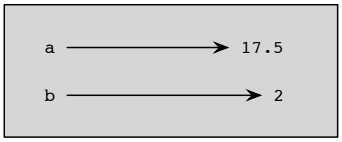

Figura 2.6: Ambiente depois da execução de `a, b = b, a`.

constante `17` é 17. Ao avaliar a expressão `nota_1 + 1`, o Python não encontra o valor de `nota_1` no ambiente, pelo que gera um erro semelhante ao apresentado na página 42.

Consideremos agora a seguinte interação:

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

Ao executar a instrução `a, b = b, a`, o Python começa por avaliar as expressões `b` e `a`, cujos valores são, respetivamente `17.5` e `2` (Figura 2.5). Em seguida, o valor `17.5` é atribuído à variável `a` e o valor `2` é atribuído à variável `b` (Figura 2.6). Ou seja a instrução `a, b = b, a` tem o efeito de trocar os valores das variáveis `a` e `b`.

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

Tabela 2.7: Operadores relacionais.

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

2.4 Predicados e condições

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

Uma expressão cujo valor é do tipo lógico chama-se uma *condição*[15]. As condições podem ser combinadas através de operações lógicas. Entre outros, no Python existem, como operações embutidas, os *operadores relacionais*, que se apresentam na Tabela 2.7.

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

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

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

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

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

Uma alternativa para tornar a notação mais simples, como a que acabámos de apresentar relativa aos operadores relacionais, é vulgarmente designada por *açúcar sintático*[16].

[16] Do inglês, "syntactic sugar".

2.5 Comunicação com o exterior

2.5.1 Leitura de dados

Durante a execução de um programa, é vulgarmente necessário obter valores do exterior para efetuar a manipulação da informação. A obtenção de valores do exterior é feita através das operações de leitura de dados. As *operações de leitura* de dados permitem transmitir informação do exterior para o programa. Por "exterior" entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador. Neste capítulo apenas consideramos operações de leitura de dados em que os dados são fornecidos através do teclado. No Capítulo 9, consideramos operações de leitura de dados localizados em ficheiros.

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

⟨leitura de dados⟩ ::= `input()` |
`input(`⟨informação⟩`)`

⟨informação⟩ ::= ⟨cadeia de carateres⟩

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

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

1. ```
   >>> input('-> ')
   -> 5
   '5'
   ```
   O Python mostra o conteúdo da cadeia de carateres `'-> '`, o qual corresponde a `-> ` , e lê o que é fornecido através do teclado, neste caso, 5 seguido de "Return", sendo devolvida a cadeia de carateres `'5'`.

2. ```
   >>> input()
   ```

Caráter escape	*Significado*
\\	Barra ao contrário (\)
\'	Plica (')
\"	Aspas (")
\b	Retrocesso de um espaço
\f	Salto de página
\n	Salto de linha
\r	"Return"
\t	Tabulação horizontal
\v	Tabulação vertical

Tabela 2.8: Alguns carateres de escape em Python.

```
estou a escrever sem caráter de pronto
'estou a escrever sem caráter de pronto'
```

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

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

Esta interação introduz algo de novo. Na cadeia de carateres que é fornecida à função `input` aparece um caráter que não vimos até agora, a sequência `\n`. A isto chama-se um caráter de escape[17]. Um *caráter de escape* é um caráter não gráfico com um significado especial para um meio de escrita, por exemplo, uma impressora ou o ecrã do computador. Em Python, um caráter de escape corresponde a um caráter precedido por uma barra ao contrário, "\". Na Tabela 2.8 apresentam-se alguns carateres de escape existentes em Python.

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

[17]Do inglês, "escape character".

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

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

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

⟨função de avaliação⟩ ::= `eval(`⟨cadeia de carateres⟩`)`

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

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

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

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

2.5.2 Escrita de dados

Após efetuar a manipulação da informação, é importante que o computador possa comunicar ao exterior os resultados a que chegou. Isto é feito através das operações de escrita de dados. As *operações de escrita de dados* permitem transmitir informação do programa para o exterior. Por "exterior" entenda-se (1) o mundo exterior ao programa, por exemplo, um ser humano, ou (2) o próprio computador, por exemplo, um ficheiro localizado dentro do computador.

Neste capítulo apenas consideramos operações de escrita de dados em que os dados são escritos no ecrã. No Capítulo 9, consideramos operações de escrita de dados para ficheiros.

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

⟨escrita de dados⟩ ::= `print()` |
`print(`⟨expressões⟩`)`

⟨expressões⟩ ::= ⟨expressão⟩ |
⟨expressão⟩`,` ⟨expressões⟩

Ao encontrar a função `print()`, o Python escreve uma linha em branco no ecrã. Ao encontrar a função `print(`⟨exp$_1$⟩`, ...` ⟨exp$_n$⟩`)`, o Python começa por avaliar cada uma das expressões ⟨exp$_1$⟩ ... ⟨exp$_n$⟩, após o que escreve os seus valores, todos na mesma linha do ecrã, separados por um espaço em branco.

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

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

Sendo `print` uma função, estamos à espera que esta devolva um valor. Se imediatamente após a interação anterior atribuirmos à variável `c` o valor devolvido por `print` constatamos o seguinte[18]:

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

Ou seja, `print` é uma função que não devolve nada! Em Python, Existem algumas funções cujo objetivo não é a produção de um valor, mas sim a produção

[18] No capítulo 13 voltamos a considerar a constante `None`.

de um *efeito*, a alteração de qualquer coisa. A função `print` é uma destas funções. Depois da avaliação da função `print`, o conteúdo do ecrã do nosso computador muda, sendo esse efeito a única razão da existência desta função.

2.6 Programas, instruções e sequenciação

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

Um *programa* em Python corresponde a uma sequência de zero ou mais definições seguida por instruções. Em notação BNF, um programa em Python, também conhecido por um *guião*[19], é definido do seguinte modo[20]:

⟨programa em Python⟩ ::= ⟨definição⟩* ⟨instruções⟩

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

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

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

Ao nível da linguagem máquina, existem dois tipos de estruturas de controle: a sequenciação e o salto. A sequenciação especifica que as instruções de um

[19] Do inglês, "script".

[20] Esta não é uma definição completa de um programa em Python, mas corresponde à definição de programa usada neste livro.

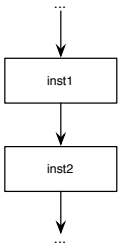

Figura 2.7: Fluxograma para a sequenciação.

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

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

A *sequenciação* é a estrutura de controle mais simples, e consiste na especificação de que as instruções de um programa são executadas sequencialmente, pela ordem em que aparecem no programa. Este é também o princípio utilizado quendo fornecemos diretamente ao interpretador do Python uma sequência de instruções. Sendo ⟨inst1⟩ e ⟨inst2⟩ símbolos não terminais que correspondem a instruções, a indicação de que a instrução ⟨inst2⟩ é executada imediatamente após a execução da instrução ⟨inst1⟩ é especificada, em Python escrevendo a instrução ⟨inst2⟩ numa linha imediatamente a seguir à linha em que aparece a instrução ⟨inst1⟩. Ou seja, o fim de linha representa implicitamente o operador de sequenciação. Na Figura 2.7 apresentamos esquematicamente o conceito de sequenciação. Esta representação diagramática tem o nome de *fluxograma*[21].

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

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

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

⟨instruções⟩ ::= ⟨instrução⟩ CR |
⟨instrução⟩ CR ⟨instruções⟩

Nesta definição, CR é um símbolo terminal que corresponde ao símbolo obtido carregando na tecla "Return" do teclado (ou, em alguns computadores, a tecla "Enter"), ou seja, CR corresponde ao fim de uma linha,

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

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

Consideremos agora o seguinte programa (sequência de instruções):

```
pi = 3.1416

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

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

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

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

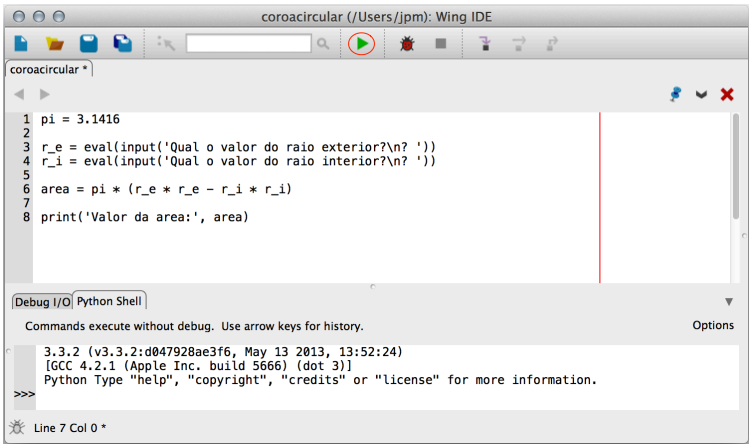

Figura 2.8: Janela do Wing101 antes de executar o programa.

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

⟨instrução vazia⟩ ::=

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

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

2.7 Seleção

Para desenvolver programas complexos, é importante que possamos especificar a execução condicional de instruções: devemos ter a capacidade de decidir se uma instrução ou grupo de instruções deve ou não ser executado, dependendo do valor de uma condição. Esta capacidade é por nós utilizada constantemente.

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

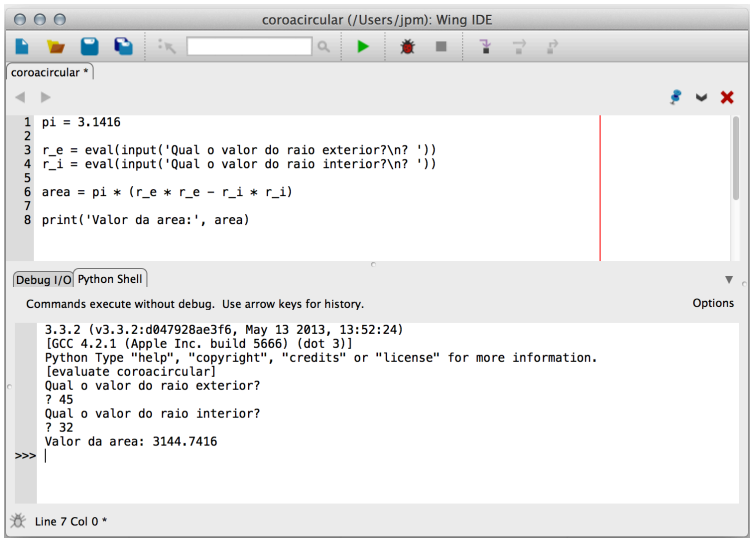

Figura 2.9: Janela do Wing101 depois de executar o programa.

Repare-se, por exemplo, nas instruções para o papagaio voador, apresentadas no Capítulo 1, página 8: "se o vento soprar forte deverá prender na argola inferior. Se o vento soprar fraco deverá prender na argola superior.".

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

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

⟨outras alternativas⟩ ::= `elif` ⟨condição⟩ `:` CR
⟨instrução composta⟩

⟨alternativa final⟩ ::= `else:` CR
⟨instrução composta⟩

[23] A palavra "if" traduz-se, em português, por "se".
[24] Existem outras alternativas para a instrução `if` que não consideramos neste livro.

⟨instrução composta⟩ ::= `TAB+` ⟨instruções⟩ `TAB-`

⟨condição⟩ ::= ⟨expressão⟩

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

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

De acordo com esta definição,

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

corresponde a uma instrução `if`, mas

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

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

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

1. Ao encontrar uma instrução da forma

   `if` ⟨condição⟩`:`
   ⟨instruções⟩

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

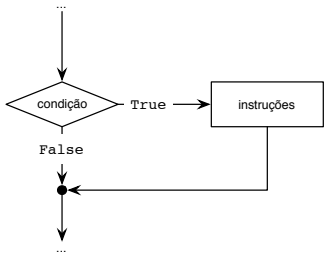

Figura 2.10: Fluxograma para uma forma da instrução `if`.

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

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

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

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

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

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

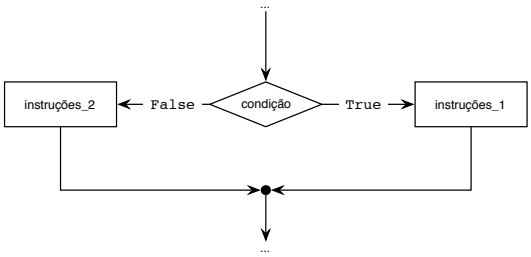

Figura 2.11: Fluxograma para outra forma da instrução `if`.

```
Escreva um número
? 3
Adeus
```

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

2. Ao encontrar uma instrução da forma[25]

   `if` ⟨cond⟩`:`
   ⟨instruções$_1$⟩
   `else:`
   ⟨instruções$_2$⟩

   o Python começa por avaliar a expressão ⟨cond⟩. Se o seu valor for `True`, as instruções correspondentes a ⟨instruções$_1$⟩ são executadas e as instruções correspondentes a ⟨instruções$_2$⟩ não o são; se o seu valor for `False`, as instruções correspondentes a ⟨instruções$_2$⟩ são executadas e as instruções correspondentes a ⟨instruções$_1$⟩ não o são. Esta semântica é mostrada na Figura 2.11 recorrendo a um fluxograma.

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

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

[25] A qual não usa ⟨outras alternativas⟩, utilizando apenas a ⟨alternativa final⟩.

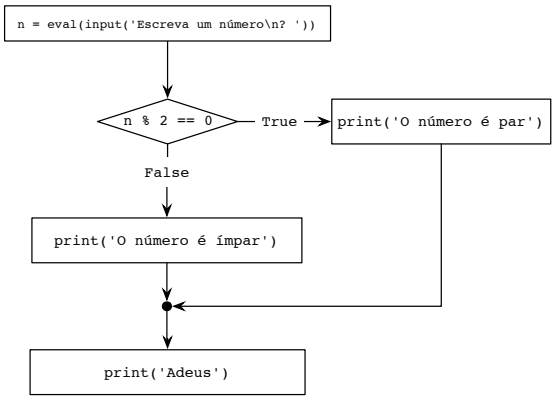

Figura 2.12: Fluxograma para determinar se um número é par ou ímpar.

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

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

```
Escreva um número
? 7
O número é ímpar
Adeus
```

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

```
if ⟨cond1⟩:
   ⟨instruções1⟩
elif ⟨cond2⟩:
   ⟨instruções2⟩
elif ⟨cond3⟩:
   ⟨instruções3⟩
  ⋮
else:
  ⟨instruçõesf⟩
```

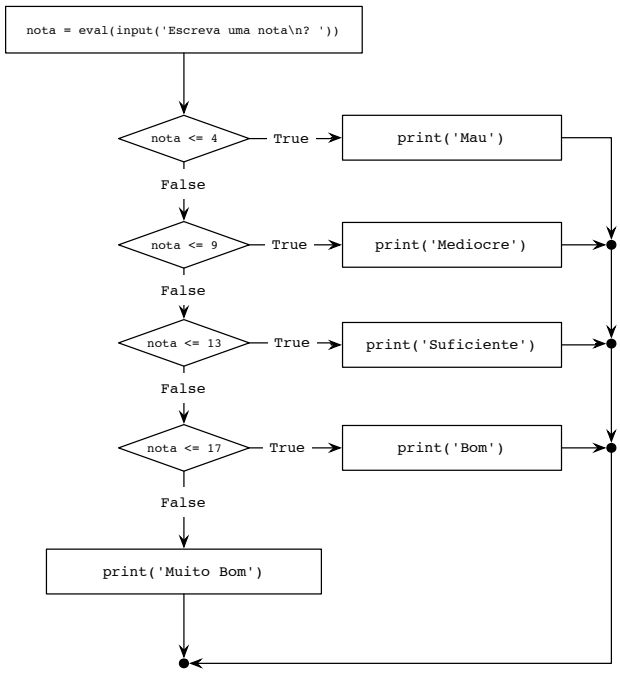

Figura 2.13: Fluxograma para a conversão de notas.

o Python começa por avaliar a expressão ⟨cond$_1$⟩. Se o seu valor for `True`, as instruções correspondentes a ⟨instruções$_1$⟩ são executadas e a execução da instrução `if` termina; se o seu valor for `False`, o Python avalia a expressão ⟨cond$_2$⟩. Se o seu valor for `True`, as instruções correspondentes a ⟨instruções$_2$⟩ são executadas e a execução da instrução `if` termina. Em caso contrário, o Python avalia a expressão ⟨cond$_3$⟩, e assim sucessivamente. Se todas as condições forem falsas, o Python executa as instruções correspondentes a ⟨instruções$_f$⟩.

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

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

```
nota = eval(input('Escreva uma nota\n? '))
if nota <= 4:
    print('Mau')
elif nota <= 9:
    print('Mediocre')
elif nota <= 13:
    print('Suficiente')
elif nota <= 17:
    print('Bom')
else:
    print('Muito bom')
```

2.8 Repetição

Em programação é frequente ser preciso repetir a execução de um grupo de instruções, ou mesmo repetir a execução de todo o programa, para diferentes valores dos dados. Repare-se, por exemplo, na receita para os rebuçados de ovos, apresentada no Capítulo 1, página 8: "Leva-se o açúcar ao lume com um copo de água e deixa-se ferver até fazer ponto de pérola", esta instrução corresponde a dizer que "enquanto não se atingir o ponto de pérola, deve-se deixar o açúcar com água a ferver".

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

Cada vez que as instruções que constituem o corpo do ciclo são executadas, dizemos que se efetuou uma *passagem pelo ciclo*. As instruções que constituem o corpo do ciclo podem ser executadas qualquer número de vezes (eventualmente, nenhuma) mas esse número tem de ser finito. Há erros semânticos que podem provocar a execução interminável do corpo do ciclo, caso em que se diz que

existe um *ciclo infinito*[26]. Em Python, existem duas instruções que permitem a especificação de ciclos, a instrução `while`, que apresentamos nesta secção, e a instrução `for`, que apresentamos no Capítulo 4.

A instrução `while` permite especificar a execução repetitiva de um conjunto de instruções enquanto uma determinada condição tiver o valor verdadeiro. A sintaxe da instrução `while` é definida pela seguinte expressão em notação BNF[27]:

⟨instrução `while`⟩ ::= `while` ⟨condição⟩`:` `CR`
⟨instrução composta⟩

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

Na instrução `while` ⟨condição⟩`:` `CR` ⟨instruções⟩, o símbolo não terminal ⟨instruções⟩ corresponde ao corpo do ciclo e o símbolo ⟨condição⟩, uma expressão do tipo lógico, representa a condição que controla a execução do ciclo.

A semântica da instrução `while` é a seguinte: ao encontrar a instrução `while` ⟨condição⟩`:` `CR` ⟨instruções⟩, o Python calcula o valor de ⟨condição⟩. Se o seu valor for `True`, o Python efetua uma passagem pelo ciclo, executando as instruções correspondentes a ⟨instruções⟩. Em seguida, volta a calcular o valor de ⟨condição⟩ e o processo repete-se enquanto o valor de ⟨condição⟩ for `True`. Quando o valor de ⟨condição⟩ for `False`, a execução do ciclo termina. Na Figura 2.14 apresentamos um fluxograma correspondente à instrução `while`.

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

No corpo do ciclo, pode ser utilizada uma outra instrução existente em Python, a instrução `break`. A instrução `break` apenas pode aparecer dentro do corpo de um ciclo. A sintaxe da instrução `break` é definida pela seguinte expressão em notação BNF[28]:

⟨instrução `break`⟩ ::= `break`

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

[27] A palavra "while" traduz-se em português por "enquanto".

[28] A palavra "break" traduz-se em português por "fuga", "quebra", "pausa", ou "rutura".

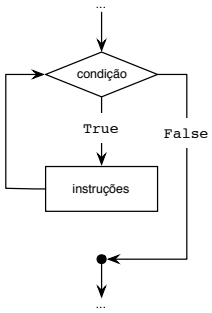

Figura 2.14: Fluxograma correspondente à instrução `while`.

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

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

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

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

print('A soma é:', soma)
```

Estas instruções começam por estabelecer o valor inicial da variável `soma` como sendo zero[29], após o que efetuam a leitura de uma sequência de números inteiros positivos, calculando a sua soma. A quantidade de números a ler é desconhecida à partida. Para assinalar que a sequência de números terminou, fornecemos ao Python um número especial, no nosso exemplo, o número `-1`. Este valor é chamado o *valor sentinela*, porque assinala o fim da sequência a ser lida.

Este programa utiliza uma instrução `while`, a qual é executada enquanto o número fornecido pelo utilizador for diferente de `-1`. O corpo deste ciclo é constituído por uma instrução composta, a qual é constituída por duas instruções

[29]Esta operação é conhecida por *inicialização da variável*.

`soma = soma + num` (que atualiza o valor da soma) e `num = eval(input('Escreva um inteiro\n(-1 para terminar): '))` (que lê o número fornecido). Note-se que este ciclo pressupõe que já foi fornecido um número ao computador, e que o corpo do ciclo inclui uma instrução (`num = eval(input('Escreva um inteiro\n(-1 para terminar): '))`) que modifica o valor da condição que controla o ciclo (`num != -1`).

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

1. De um modo geral, o número de vezes que o corpo do ciclo é executado não pode ser calculado antecipadamente: a condição que especifica o término do ciclo é testada durante a execução do próprio ciclo, sendo impossível saber de antemão como vai prosseguir a avaliação;

2. Pode acontecer que o corpo do ciclo não seja executado nenhuma vez. Com efeito, a semântica da instrução `while` especifica que o valor da expressão que controla a execução do ciclo é calculado antes do início da execução do ciclo. Se o valor inicial desta expressão é `False`, o corpo do ciclo não é executado.

2.9 Exemplos

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

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

O seguinte programa realiza a tarefa desejada:

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

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

print('Soma dos dígitos:', soma)
```

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

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

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

Consideremos agora o programa que corresponde ao algoritmo que acabámos de

[30] Como exercício, o leitor deve convencer-se que este algoritmo apenas produz números primos.

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

Tabela 2.9: Fatores primos de 780.

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

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

O seguinte programa calcula os fatores primos de um número:

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

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

Com este programa obtemos a interação:

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

2.10 Notas finais

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

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

2.11 Exercícios

1. Escreva um programa em Python que pede ao utilizador que lhe forneça um inteiro correspondente a um número de segundos e que calcula o número de dias correspondentes a esse número de segundos. O seu programa deve permitir a interação:

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

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

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

3. Escreva um programa em Python que lê três números e que diz qual o maior dos números lidos.

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

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

5. A conversão de temperatura em graus Farenheit (F) para graus centígrados (C) é dada através da expressão

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

Escreva um programa em Python que produz uma tabela com as temperaturas em graus centígrados, equivalentes às temperaturas em graus Farenheit entre -40° F e 120° F.

6. Escreva um programa em Python que lê uma sequência de dígitos, sendo cada um dos dígitos fornecido numa linha separada, e calcula o número inteiro composto por esses dígitos, pela ordem fornecida. Para terminar a sequência de dígitos é fornecido ao programa o inteiro -1. O seu programa deve permitir a interação:

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

7. Escreva um programa em Python que lê um número inteiro positivo e calcula a soma dos seus dígitos pares. Por exemplo,

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

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

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

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

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

para um dado valor de x e de n. O seu programa deve ter em atenção que o i-ésimo termo da série pode ser obtido do termo na posição $i - 1$, multiplicando-o por x/i. O seu programa deve permitir a interação:

```
Qual o valor de x
? 2
Qual o valor de n
? 3
O valor da soma é 6.333333333333333
```