CAPÍTULO 3

Variáveis e entrada de dados

O capítulo anterior apresentou o conceito de variáveis, mas há mais por descobrir. Já sabemos que variáveis têm nomes que permitem acessar os valores dessas variáveis em outras partes do programa. Neste capítulo, vamos ampliar nosso conhecimento sobre variáveis, estudando novas operações e novos tipos de dados.

3.1 Nomes de variáveis

Em Python, nomes de variáveis devem iniciar obrigatoriamente com uma letra, mas podem conter números e o símbolo sublinha (_). Vejamos exemplos de nomes válidos e inválidos em Python na tabela 3.1.

Tabela 3.1 – Exemplo de nomes válidos e inválidos para variáveis

Nome	Válido	Comentários
`a1`	Sim	Embora contenha um número, o nome a1 inicia com letra.
`velocidade`	Sim	Nome formado por letras.
`velocidade90`	Sim	Nome formado por letras e números, mas iniciado por letra.
`salário_médio`	Sim	O símbolo sublinha (_) é permitido e facilita a leitura de nomes grandes.
`salário médio`	Não	Nomes de variáveis não podem conter espaços em branco.
`_b`	Sim	O sublinha (_) é aceito em nomes de variáveis, mesmo no início.
`1a`	Não	Nomes de variáveis não podem começar com números.

A versão 3 da linguagem Python permite a utilização de acentos em nomes de variáveis, pois, por padrão, os programas são interpretados utilizando-se um conjunto de caracteres chamado UTF-8 (*http://pt.wikipedia.org/wiki/Utf-8*), capaz de representar praticamente todas as letras dos alfabetos conhecidos.

Variáveis têm outras propriedades além de nome e conteúdo. Uma delas é conhecida como tipo e define a natureza dos dados que a variável armazena. Python tem vários tipos de dados, mas os mais comuns são números inteiros, números de ponto flutuante e strings. Além de poder armazenar números e letras, as variáveis em Python também armazenam valores como verdadeiro ou falso. Dizemos que essas variáveis são do tipo lógico. Veremos mais sobre variáveis do tipo lógico na seção 3.3.

TRÍVIA

A maior parte das linguagens de programação foi desenvolvida nos Estados Unidos, ou considerando apenas nomes escritos na língua inglesa como nomes válidos. Por isso, acentos e letras consideradas especiais não são aceitos como nomes válidos na maior parte das linguagens de programação. Essa restrição é um problema não só para falantes do português, mas de muitas outras línguas que utilizam acentos ou mesmo outros alfabetos. O padrão americano é baseado no conjunto de caracteres ASCII (*), desenvolvido na década de 1960. Com a globalização da economia e o advento da internet, as aplicações mais modernas são escritas para trabalhar com um conjunto de caracteres dito universal, chamado Unicode (**).

(*) *http://pt.wikipedia.org/wiki/Ascii*
(**) *http://pt.wikipedia.org/wiki/Unicode*

3.2 Variáveis numéricas

Dizemos que uma variável é numérica quando armazena números inteiros ou de ponto flutuante.

Os números inteiros são aqueles sem parte decimal, como 1, 0, -5, 550, -47, 30000.

Números de ponto flutuante ou decimais são aqueles com parte decimal, como 1.0, 5.478, 10.478, 30000.4. Observe que 1.0, mesmo tendo zero na parte decimal, é um número de ponto flutuante.

Em Python, e na maioria das linguagens de programação, utilizamos o ponto, e não a vírgula, como separador entre a parte inteira e fracionária de um número. Essa é outra herança da língua inglesa. Observe também que não utilizamos nada como separador de milhar. Exemplo: 1.000.000 (um milhão) é escrito 1000000.

Exercício 3.1 Complete a tabela a seguir, marcando inteiro ou ponto flutuante dependendo do número apresentado.

Número	Tipo numérico
5	❍ inteiro ❍ ponto flutuante
5.0	❍ inteiro ❍ ponto flutuante
4.3	❍ inteiro ❍ ponto flutuante
-2	❍ inteiro ❍ ponto flutuante
100	❍ inteiro ❍ ponto flutuante
1.333	❍ inteiro ❍ ponto flutuante

3.2.1 Representação de valores numéricos

Internamente, todos os números são representados utilizando o sistema binário, ou seja, de base 2. Esse sistema permite apenas os dígitos 0 e 1. Para representar números maiores, combinamos vários dígitos, exatamente como fazemos com o sistema decimal, ou de base 10, que utilizamos.

Vejamos primeiro como isso funciona na base 10:

$$\begin{aligned} 531 &= 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \\ &= 5 \times 100 + 3 \times 10 + 1 \times 1 \\ &= 500 + 30 + 1 \\ &= 531 \end{aligned}$$

Multiplicamos cada dígito pela base elevada a um expoente igual ao número de casas à direita do dígito em questão. Como em 531 o 5 tem 2 dígitos à direita, multiplicamos 5×10^2. Para o 3, temos apenas outro dígito à direita, logo, 3×10^1. Finalmente, para o 1, sem dígitos à direita, temos 1×10^0. Somando esses componentes, temos o número 531. Fazemos isso tão rápido que é natural ou automático pensarmos desse jeito.

A mudança para o sistema binário segue o mesmo processo, mas a base agora é 2, e não 10. Assim, 1010 em binário representa:

$$\begin{aligned} 1010 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 8 + 2 \\ &= 10 \end{aligned}$$

A utilização do sistema binário é transparente em Python, ou seja, se você não solicitar explicitamente que esse sistema seja usado, tudo será apresentado na base 10 utilizada no dia a dia. A importância da noção de diferença de base é importante, pois ela explica os limites da representação. O limite de representação é o valor mínimo e máximo que pode ser representado em uma variável numérica. Esse limite é causado pela quantidade de dígitos que foram reservados para armazenar o número em questão. Vejamos como funciona na base 10.

Se você tem apenas 5 dígitos para representar um número, o maior número é 99999. E o menor seria (-99999). O mesmo acontece no sistema binário, sendo que lá reservamos um dígito para registrar os sinais de positivo e negativo.

Para números inteiros, Python utiliza um sistema de precisão ilimitada que permite a representação de números muito grandes. É como se você sempre pudesse escrever novos dígitos à medida que for necessário. Você pode calcular em Python valores como $2^{1000000}$ (`2 ** 1000000`) sem problemas de representação, mesmo quando o resultado é um número de 301030 dígitos.

Em ponto flutuante, temos limite e problemas de representação. Um número decimal é representado em ponto flutuante utilizando-se uma mantissa e um expoente (*sinal* × *mantissa* × *base*$^{\text{expoente}}$). Tanto a mantissa quanto o expoente têm um número de dígitos máximos que limita os números que podem ser representados. Você não precisa se preocupar com isso no momento, pois esses valores são bem grandes e você não terá problemas na maioria de seus programas. Você pode obter mais informações acessando *http://pt.wikipedia.org/wiki/Ponto_flutuante*.

A versão 3.1.2 do Python, rodando em Mac OS X, tem como limites $2.2250738585072014 \times 10^{-308}$ e $1.7976931348623157 \times 10^{308}$, suficientemente grandes e pequenos para quase qualquer tipo de aplicação. É possível encontrar problemas de representação em função de como os números decimais são convertidos em números de ponto flutuante. Esses problemas são bem conhecidos e afetam todas as linguagens de programação, não sendo um problema específico do Python.

Vejamos um exemplo: o número 0.1 não tem nada de especial no sistema decimal, mas é uma dízima periódica no sistema binário. Você não precisa se preocupar com esses detalhes agora, mas pode investigá-los mais tarde quando precisar (normalmente cursos de computação apresentam uma disciplina, chamada cálculo numérico, para abordar esses tipos de problemas). Digite no interpretador `3 * 0.1`

Você deve ter obtido como resultado 0.30000000000000004, e não 0.3, como esperado. Não se assuste: não é um problema de seu computador, mas de representação. Se for necessário, durante seus estudos, cálculos mais precisos, ou se os resultados em ponto flutuante não satisfizerem os requisitos de precisão esperados,

verifique os módulos `decimals` e `fractions`. A documentação do Python traz uma página específica sobre esse tipo de problema: *http://docs.python.org/py3k/tutorial/floatingpoint.html*.

3.3 Variáveis do tipo Lógico

Muitas vezes, queremos armazenar um conteúdo simples: verdadeiro ou falso em uma variável. Nesse caso, utilizaremos um tipo de variável chamado tipo lógico ou booleano. Em Python, escreveremos **`True`** para verdadeiro e **`False`** para falso (Listagem 3.1). Observe que o T e o F são escritos com letras maiúsculas.

▶ Listagem 3.1 – Exemplo de variáveis do tipo lógico

```
resultado = True
aprovado = False
```

3.3.1 Operadores relacionais

Para realizarmos comparações lógicas, utilizaremos operadores relacionais. A lista de operadores relacionais suportados em Python é apresentada na tabela 3.2.

Tabela 3.2 – Operadores relacionais

Operador	Operação	Símbolo matemático
==	igualdade	=
>	maior que	>
<	menor que	<
!=	diferente	≠
>=	maior ou igual	≥
<=	menor ou igual	≤

O resultado de uma comparação é um valor do tipo lógico, ou seja, **`True`** (verdadeiro) ou **`False`** (falso). Utilizaremos o verbo "avaliar" para indicar a resolução de uma expressão.

▶ Listagem 3.2 – Exemplo de uso de operadores relacionais

```
>>> a = 1       # a recebe 1
>>> b = 5       # b recebe 5
```

```
>>> c = 2       # c recebe 2
>>> d = 1       # d recebe 1
>>> a == b      # a é igual a b ?
False
>>> b > a       # b é maior que a?
True
>>> a < b       # a é menor que b?
True
>>> a == d      # a é igual a d?
True
>>> b >= a      # b é maior ou igual a a?
True
>>> c <= b      # c é menor ou igual a b?
True
>>> d != a      # d é diferente de a?
False
>>> d != b      # d é diferente de b?
True
```

Esses operadores são utilizados como na matemática. Especial atenção deve ser dada aos operadores `>=` e `<=`. O resultado desses operadores é realmente maior ou igual e menor ou igual, ou seja, `5 >= 5` é verdadeiro, assim como `5 <= 5`.

Observe que, na listagem 3.2, utilizamos o símbolo de cerquilha (`#`) para escrever comentários na linha de comando. Todo texto à direita do cerquilha é ignorado pelo interpretador Python, ou seja, você pode escrever o que quiser. Leia novamente a listagem 3.2 e veja como é mais fácil entender cada linha quando a comentamos. Comentários não são obrigatórios, mas são muito importantes. Você pode e deve utilizá-los em seus programas para facilitar o entendimento e oferecer uma anotação para você mesmo. É comum lermos um programa alguns meses depois de escrito e termos dificuldade de lembrar o que realmente queríamos fazer. Você também não precisa comentar todas as linhas de seus programas ou escrever o óbvio. Uma dica é identificar os programas com seu nome, a data em que começou a ser escrito e mesmo a listagem ou capítulo do livro onde você o encontrou.

Variáveis de tipo lógico também podem ser utilizadas para armazenar o resultado de expressões e comparações.

► Listagem 3.3 – Exemplo do uso de operadores relacionais com variáveis do tipo lógico

```
nota = 8
média = 7
aprovado = nota > média
print(aprovado)
```

Se uma expressão contém operações aritméticas, estas devem ser calculadas antes que os operadores relacionais sejam avaliados. Quando avaliamos uma expressão, substituímos o nome das variáveis por seu conteúdo e só então verificamos o resultado da comparação.

Exercício 3.2 Complete a tabela abaixo, respondendo True ou False. Considere a = 4, b = 10, c = 5.0, d = 1 e f = 5.

Expressão	Resultado
a == c	❍ True ❍ False
a < b	❍ True ❍ False
d > b	❍ True ❍ False
c! = f	❍ True ❍ False
a == b	❍ True ❍ False
c < d	❍ True ❍ False
b > a	❍ True ❍ False
c >= f	❍ True ❍ False
f >= c	❍ True ❍ False
c <= c	❍ True ❍ False
c <= f	❍ True ❍ False

3.3.2 Operadores lógicos

Para agrupar operações com lógica booleana, utilizaremos operadores lógicos. Python suporta três operadores básicos: **not** (não), **and** (e), **or** (ou). Esses operadores podem ser traduzidos como não (¬ negação), e (∧ conjunção) e ou (∨ disjunção).

Tabela 3.3 – Operadores lógicos

Operador Python	Operação
`not`	não
`and`	e
`or`	ou

Cada operador obedece a um conjunto simples de regras, expresso pela tabela verdade desse operador. A tabela verdade demonstra o resultado de uma operação com um ou dois valores lógicos ou operandos. Quando o operador utiliza apenas um operando, dizemos que é um operador unário. Ao utilizar dois operandos, é chamado operador binário. O operador de negação (`not`) é um operador unário. `or` (ou) e `and` (e) são operadores binários, precisando, assim, de dois operandos.

3.3.2.1 Operador not

O operador `not` (não) é o mais simples, pois precisa apenas de um operador. A operação de negação também é chamada de inversão, pois um valor verdadeiro negado se torna falso e vice-versa. A tabela verdade do operador `not` (não) é apresentada na tabela 3.4.

Tabela 3.4 – Tabela verdade do operador not (não)

V_1	not V_1
V	F
F	V

▶ Listagem 3.4 – Operador not

```
>>> not True
False
>>> not False
True
```

3.3.2.2 Operador and

O operador `and` (e) tem sua tabela verdade representada na tabela 3.5. O operador `and` (e) resulta verdadeiro apenas quando seus dois operadores forem verdadeiros.

Tabela 3.5 – Tabela verdade do operador and (e)

V_1	V_2	V_1 and V_2
V	V	V
V	F	F
F	V	F
F	F	F

▶ Listagem 3.5 – Operador and

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
```

3.3.2.3 Operador or

A tabela verdade do operador **or** (ou) é apresentada na tabela 3.6. A regra fundamental do operador **or** (ou) é que ele resulta em falso apenas se seus dois operadores também forem falsos. Se apenas um de seus operadores for verdadeiro, ou se os dois forem, o resultado da operação será verdadeiro.

Tabela 3.6 – Tabela verdade do operador or(ou)

V_1	V_2	V_1 or V_2
V	V	V
V	F	V
F	V	V
F	F	F

▶ Listagem 3.6 – Operador or

```
>>> True or True
True
```

```
>>> True or False
True
>>> False or True
True
>>> False or False
False
```

Exercício 3.3 Complete a tabela a seguir utilizando a = True, b = False e c = True.

Expressão	Resultado
a and a	❍ True ❍ False
b and b	❍ True ❍ False
not c	❍ True ❍ False
not b	❍ True ❍ False
not a	❍ True ❍ False
a and b	❍ True ❍ False
b and c	❍ True ❍ False
a or c	❍ True ❍ False
b or c	❍ True ❍ False
c or a	❍ True ❍ False
c or b	❍ True ❍ False
c or c	❍ True ❍ False
b or b	❍ True ❍ False

3.3.3 Expressões lógicas

Os operadores lógicos podem ser combinados em expressões lógicas mais complexas. Quando uma expressão tiver mais de um operador lógico, avalia-se o operador **not** (não) primeiramente, seguido do operador **and** (e) e, finalmente, **or** (ou). Vejamos a seguir a ordem de avaliação da expressão, onde a operação sendo avaliada é sublinhada; e o resultado, mostrado na linha seguinte.

True or False and not True

True or False and False

True or False

True

Os operadores relacionais também podem ser utilizados em expressões com operadores lógicos.

salário > 1000 and idade > 18

Nesses casos, os operadores relacionais devem ser avaliados primeiramente. Façamos `salário = 100` e `idade = 20`. Teremos:

salário > 1000 and idade > 18

100 > 1000 and 20 > 18

False and True

False

A grande vantagem de escrever esse tipo de expressão é representar condições que podem ser avaliadas com valores diferentes. Por exemplo: imagine que `salário > 1000 and idade > 18` seja uma condição para um empréstimo de compra de um carro novo. Quando `salário = 100` e `idade = 20`, sabemos que o resultado da expressão é falso, e podemos interpretar que, nesse caso, a pessoa não receberia o empréstimo. Avaliemos a mesma expressão com `salário = 2000` e `idade = 30`.

salário > 1000 and idade > 18

2000 > 1000 and 30 > 18

True and True

True

Agora o resultado é **`True`** (verdadeiro) e poderíamos dizer que a pessoa atende às condições para obter o empréstimo.

Exercício 3.4 Escreva uma expressão para determinar se uma pessoa deve ou não pagar imposto. Considere que pagam imposto pessoas cujo salário é maior que R$ 1.200,00.

Exercício 3.5 Calcule o resultado da expressão A > B and C or D, utilizando os valores da tabela a seguir.

A	B	C	D	Resultado
1	2	True	False	
10	3	False	False	
5	1	True	True	

Exercício 3.6 Escreva uma expressão que será utilizada para decidir se um aluno foi ou não aprovado. Para ser aprovado, todas as médias do aluno devem ser maiores que 7. Considere que o aluno cursa apenas três matérias, e que a nota de cada uma está armazenada nas seguintes variáveis: matéria1, matéria2 e matéria3.

3.4 Variáveis string

Variáveis do tipo string armazenam cadeias de caracteres como nomes e textos em geral. Chamamos cadeia de caracteres uma sequência de símbolos como letras, números, sinais de pontuação etc. Exemplo: João e Maria comem pão. Nesse caso, João é uma sequência com as letras J, o, ã, o. Para simplificar o texto, utilizaremos o nome string para mencionar cadeias de caracteres. Podemos imaginar uma string como uma sequência de blocos, onde cada letra, número ou espaço em branco ocupa uma posição, como mostra a figura 3.1.

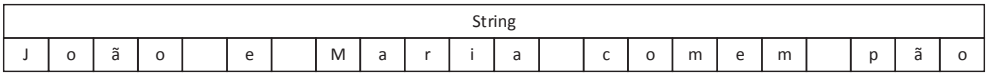

Figura 3.1 – Representação de uma string.

Para possibilitar a separação entre o texto do seu programa e o conteúdo de uma string, utilizaremos aspas (") para delimitar o início e o fim da sequência de caracteres.

Voltando ao exemplo anterior, escreveremos `"João e Maria comem pão"`. Veja que nesse caso não há qualquer problema em utilizarmos espaços. Na verdade, o computador ignora praticamente tudo que escrevemos entre aspas, mas veremos mais tarde que não é bem assim.

As variáveis do tipo string são utilizadas para armazenar sequências de caracteres, normalmente utilizadas em textos ou mensagens. O tipo string é muito útil e bastante utilizado para exibir mensagens ou mesmo para gerar outros arquivos.

Uma string em Python tem um tamanho associado, assim como um conteúdo que pode ser acessado caractere a caractere. O tamanho de uma string pode ser obtido utilizando-se a função `len`. Essa função retorna o número de caracteres na string. Dizemos que uma função retorna um valor quando podemos substituir o texto da função por seu resultado. A função `len` retorna um valor do tipo inteiro, representando a quantidade de caracteres contidos na string. Se a string é vazia (representada simplesmente por `""`, ou seja, duas aspas sem nada entre elas, nem mesmo espaços em branco), seu tamanho é igual a zero. Façamos alguns testes, como mostra a listagem 3.7.

▶ Listagem 3.7 – A função len

```
>>> print (len("A"))
1
>>> print (len("AB"))
2
>>> print (len(""))
0
>>> print (len("O rato roeu a roupa"))
19
```

Como dito anteriormente, outra característica de strings é poder acessar seu conteúdo caractere a caractere. Sabendo que uma string tem um determinado tamanho, podemos acessar seus caracteres utilizando um número inteiro para representar sua posição. Esse número é chamado de índice, e começamos a contar de zero. Isso quer dizer que o primeiro caractere da string é de posição ou índice 0. Observe a figura 3.2.

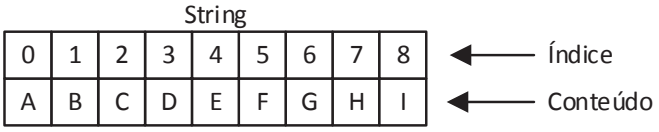

Figura 3.2 – Índices e conteúdo de uma variável string.

Para acessar os caracteres de uma string, devemos informar o índice ou posição do caractere entre colchetes (`[ ]`). Como o primeiro caractere de uma string é o de índice 0, podemos acessar valores de 0 até o tamanho da string menos 1. Logo,

se a string contiver 9 caracteres, poderemos acessar os caracteres de 0 a 8. Veja o resultado de alguns testes com strings na listagem 3.8. Se tentarmos acessar um índice maior que a quantidade de caracteres da string, o interpretador emitirá uma mensagem de erro.

▶ Listagem 3.8 – Manipulação de strings no interpretador

```
>>> a = "ABCDEF"
>>> print(a[0])
A
>>> print(a[1])
B
>>> print(a[5])
F
>>> print(a[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(len(a))
6
```

3.4.1 Operações com strings

As variáveis de tipo string suportam operações como fatiamento, concatenação e composição. Por fatiamento, podemos entender a capacidade de utilizar apenas uma parte de uma string, ou uma fatia. A concatenação nada mais é que poder juntar duas ou mais strings em uma nova string maior. A composição é muito utilizada em mensagens que enviamos à tela e consiste em utilizar strings como modelos onde podemos inserir outras strings. Veremos cada uma dessas operações nas seções a seguir.

3.4.1.1 Concatenação

O conteúdo de variáveis string podem ser somados, ou melhor, concatenados. Para concatenar duas strings, utilizamos o operador de adição (+). Assim, `"AB" + "C"` é igual a `"ABC"`. Um caso especial de concatenação é a repetição de uma string várias vezes. Para isso, utilizamos o operador de multiplicação (*): `"A" * 3` é igual a `"AAA"`. Vejamos alguns exemplos na listagem 3.9.

▶ Listagem 3.9 – Exemplo de concatenação

```
>>> s = "ABC"
>>> print (s + "C")
ABCC
>>> print (s + "D" * 4)
ABCDDDD
>>> print ("X" + "-"*10 + "X")
X----------X
>>> print (s+"x4 = "+s*4)
ABCx4 = ABCABCABCABC
```

3.4.1.2 Composição

Juntar várias strings para construir uma mensagem nem sempre é prático. Por exemplo, exibir que `"João tem X anos"`, onde X é uma variável numérica.

Usando a composição de strings do Python, podemos escrever de forma simples e clara:

```
"João tem %d anos" % X
```

Onde o símbolo de % foi utilizado para indicar a composição da string anterior com o conteúdo da variável X. O `%d` dentro da primeira string é o que chamamos de marcador de posição. O marcador indica que naquela posição estaremos colocando um valor inteiro, daí o `%d`.

Python suporta diversas operações com marcadores. Veremos mais sobre marcadores em outras partes do livro. A tabela 3.7 apresenta os principais tipos de marcadores. Veja que eles são diferentes, de acordo com o tipo de variável que vamos utilizar.

Tabela 3.7 – Marcadores

Marcador	Tipo
%d	Números inteiros
%s	Strings
%f	Números decimais

Imagine que precisamos apresentar um número como 001 ou 002, mas que também pode ser algo como 050 ou 561. Nesse caso, estamos querendo apresentar um número com três posições, completando com zeros à esquerda se o número

for menor. Podemos realizar essa operação utilizando `"%03d" % X`. Observe que adicionamos 03 entre o `%` e o `d`. Se você precisar apenas que o número ocupe três posições, mas não desejar zeros à esquerda, basta retirar o zero e utilizar `"%3d" % X`. Isso é muito importante quando estamos gravando dados em um arquivo ou simplesmente exibindo informações na tela. Vejamos alguns exemplos na listagem 3.10.

▶ Listagem 3.10 – Exemplo de composição com marcadores

```
>>> idade = 22
>>> print("[%d]" % idade)
[22]
>>> print("[%03d]" % idade)
[022]
>>> print("[%3d]" % idade)
[ 22]
>>> print("[%-3d]" % idade)
[22 ]
```

Quando formatamos números decimais, podemos utilizar dois valores entre o símbolo de `%` e a letra `f`. O primeiro indica o tamanho total em caracteres a reservar; e o segundo, o número de casas decimais. Assim, `%5.2f` diz que estaremos imprimindo um número decimal utilizando cinco posições, sendo que duas são para a parte decimal. Isso é muito interessante para exibir o resultado de cálculos ou representar dinheiro. Por exemplo, para exibir R$ 5, você pode utilizar `"R$%f" % 5`, mas o resultado não é bem o que esperamos, pois normalmente utilizamos apenas dois dígitos após a vírgula quando falamos de dinheiro. Vejamos alguns exemplos na listagem 3.11.

▶ Listagem 3.11 – Exemplos de composição com números decimais

```
>>> print("%5f" % 5)
5.000000
>>> print("%5.2f" % 5)
 5.00
>>> print("%10.5f" % 5)
   5.00000
```

O poder da composição realmente aparece quando precisamos combinar vários valores em uma nova string. Imagine que João tem 22 anos e apenas R$ 51,34 no bolso. Para exibir essa mensagem, podemos utilizar:

```
"%s tem %d anos e apenas R$%5.2f no bolso" % ("João", 22, 51.34)
```

Python suporta diversas operações com marcadores. Veremos mais sobre marcadores em outras partes do livro. Quando temos mais de um marcador na string, somos obrigados a escrever os valores a substituir entre parênteses. Agora, vejamos exemplos com outros tipos, e utilizando mais de uma variável na composição, na listagem 3.12.

▶ Listagem 3.12 – Exemplo de composição de string

```
>>> nome = "João"
>>> idade = 22
>>> grana = 51.34
>>> print("%s tem %d anos e R$%f no bolso." % (nome, idade, grana))
João tem 22 anos e R$51.340000 no bolso.
>>> print("%12s tem %3d anos e R$%5.2f no bolso." % (nome, idade, grana))
        João tem  22 anos e R$51.34 no bolso.
>>> print("%12s tem %03d anos e R$%5.2f no bolso." % (nome, idade, grana))
        João tem 022 anos e R$51.34 no bolso.
>>> print("%-12s tem %-3d anos e R$%-5.2f no bolso." % (nome, idade, grana))
João         tem 22  anos e R$51.34 no bolso.
```

3.4.1.3 Fatiamento

O fatiamento em Python é muito poderoso. Imagine nossa string de exemplo da figura 3.2. Podemos fatiá-la de forma a escrever apenas seus dois primeiros caracteres AB utilizando como índice [0:2]. O fatiamento funciona com a utilização de dois pontos no índice da string. O número à esquerda dos dois pontos indica a posição de início da fatia; e o à direita, do fim. No entanto, é preciso ter atenção ao final, pois no exemplo anterior utilizamos 2; e o C, que é o caractere na posição 2; não foi incluído. Dizemos que isso acontece porque o final da fatia não é incluído na mesma, sendo deixado de fora. Entenda [0:2] como a fatia de caracteres da posição 0 até a posição 2, sem incluí-la, ou o intervalo fechado em 0 e aberto em 2.

Vejamos outros exemplos de fatias na listagem 3.13.

▶ Listagem 3.13 – Exemplo de fatiamento

```
>>> s="ABCDEFGHI"
>>> print (s[0:2])
AB
```

```
>>> print (s[1:2])
B
>>> print (s[2:4])
CD
>>> print (s[0:5])
ABCDE
>>> print (s[1:8])
BCDEFGH
```

Podemos também omitir o número da esquerda ou o da direita para representar do início ou até o final. Assim, `[:2]` indica do início até o segundo caractere (sem incluí-lo), e `[1:]` indica do caractere de posição 1 até o final da string. Observe que, nesse caso, nem precisamos saber quantos caracteres a string contém.

Se omitirmos o início e o fim da fatia, estaremos fazendo apenas uma cópia de todos os caracteres da string para uma nova string.

Podemos também utilizar valores negativos para indicar posições a partir da direita. Assim `-1` é o último caractere; `-2`, o penúltimo; e assim por diante. Veja o resultado de testes com índices negativos na listagem 3.14.

▶ Listagem 3.14 – Exemplo de fatiamento com omissão de valores e com índices negativos

```
>>> s="ABCDEFGHI"
>>> print (s[:2])
AB
>>> print (s[1:])
BCDEFGHI
>>> print (s[0:-2])
ABCDEFG
>>> print (s[:])
ABCDEFGHI
>>> print (s[-1:])
I
>>> print (s[-5:7])
EFG
>>> print (s[-2:-1])
H
```

Veremos mais sobre strings em Python no capítulo 7.

3.5 Sequências e tempo

Um programa é executado linha por linha pelo computador, executando as operações descritas no programa uma após a outra. Quando trabalhamos com variáveis, devemos nos lembrar de que o conteúdo de uma variável pode mudar com o tempo. Isso porque a cada vez que alteramos o valor de uma variável, o valor anterior é substituído pelo novo.

Observe o programa da listagem 3.15. A variável `dívida` foi utilizada para registrar quanto alguém estava devendo; e a variável `compra`, o valor de novas despesas dessa pessoa. Como somos justos, a pessoa começou sem dívidas em ❶.

▶ Listagem 3.15 – Exemplo de sequência e tempo

```
dívida = 0 ❶
compra = 100 ❷
dívida = dívida + compra ❸
compra = 200 ❹
dívida = dívida + compra ❺
compra = 300 ❻
dívida = dívida + compra ❼
compra = 0 ❽
print(dívida) ❾
```

Em ❷, temos a primeira compra no valor de R$ 100. No entanto, o valor da dívida continua sendo 0, pois ainda não alteramos seu valor de forma a adicionar a compra realizada. Isso é feito em ❸. Observe que estamos atualizando o valor da dívida com o valor atual mais a compra.

Em ❹, registramos uma nova compra no valor de R$ 200. Nesse ponto, compra tem seu valor substituído por R$ 200, causando a perda do valor anterior R$ 100. Como já somamos o valor anterior na variável dívida, essa perda não representará um problema.

❺ é idêntica a ❸, mas o resultado é bem diferente. Nesse momento, compra vale R$ 200; e dívida, R$ 100.

Em ❻, alteramos o valor de compra novamente. Dessa vez, a compra foi de R$ 300.

❼ é idêntica a ❸ e ❺, mas seu resultado é diferente, pois nesse momento temos compra valendo R$ 300; e dívida igual a R$ 300 (100 + 200), sendo atualizada para R$ 600 (300 + 300).

Em ❽, simplesmente dizemos que a compra foi 0, representando que a pessoa não comprou mais nada.

❾ exibe o conteúdo da variável dívida na tela (600).

A figura 3.3 mostra a evolução do conteúdo de nossas duas variáveis em função do tempo, representado pelo número da linha.

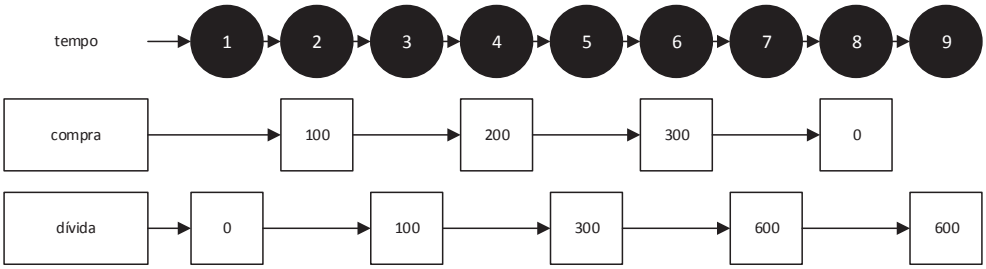

Figura 3.3 – Mudanças no valor de duas variáveis no tempo.

3.6 Rastreamento

Uma das principais diferenças entre ler um texto e um programa é justamente seguir as mudanças de valores de cada variável conforme o programa é executado. Entender que o valor das variáveis pode mudar durante a execução do programa não é tão natural, mas é fundamental para a programação de computadores. Um programa não pode ser lido como um texto, mas cuidadosamente analisado linha a linha. Ao escrever seus programas, verifique linha a linha os efeitos e mudanças causados no valor de cada variável. Para programar corretamente, você deve ser capaz de entender o que cada linha do programa significa e os efeitos que produz. Essa atividade, chamada de rastreamento, é muito importante para entender novos programas e para encontrar erros nos programas que você escreverá.

Para rastrear um programa, utilize lápis, borracha e uma folha de papel. Escreva o nome de suas variáveis na folha de papel, como se fossem títulos de colunas, deixando espaço para ser preenchido embaixo desses nomes. Leia uma linha do programa de cada vez e escreva o valor atribuído a cada variável na outra folha, na mesma coluna em que escreveu o nome da variável. Se o valor da variável mudar, escreva o novo valor e risque o anterior, um embaixo do outro, formando uma coluna. Ao exibir algo na tela, escreva também na outra folha, como se ela fosse a sua tela (você pode desenhar a tela como se fosse uma variável, mas se lembre de deixar um pouco mais de espaço). Um exemplo de como ficaria o resultado do rastreamento do programa da listagem 3.15 é apresentado na figura 3.4. O

rastreamento vai ajudá-lo a entender melhor as mudanças de valores de suas variáveis e a acompanhar a execução do programa, como mais tarde será feito pelo computador. É um processo detalhado que precisa de atenção. Não tente simplificá-lo ou começar a rastrear no meio de um programa. Você deve rastrear linha a linha, do início ao fim do programa. Se você encontrar um erro, pode parar o rastreamento e corrigi-lo, mas lembre-se de recomeçar do início sempre que alterar o programa ou os valores sendo rastreados.

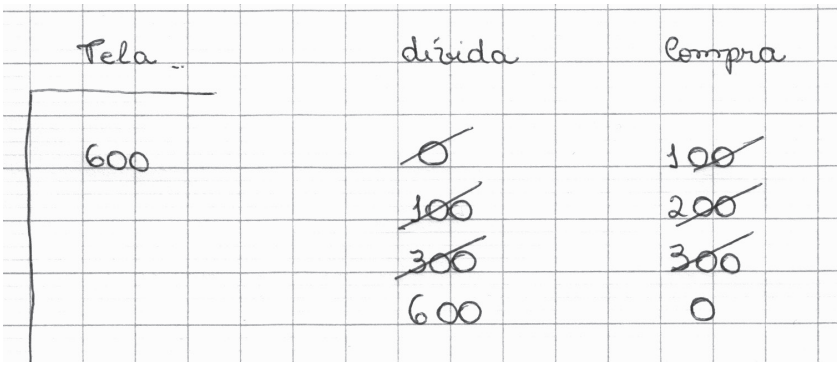

Figura 3.4 – Exemplo de rastreamento no papel.

Dominar o rastreamento de um programa é essencial para programar e ajuda muito a entender como os programas realmente funcionam. Lembre-se de que programar é detalhar, e que simplesmente ler o texto de um programa não é suficiente. Você deve rastreá-lo para entendê-lo. Embora pareça óbvio, esse é um dos erros mais comuns quando se começa a programar. Se um programa não funciona ou se você não entendeu exatamente o que ele faz, o rastreamento é a melhor ferramenta.

3.7 Entrada de dados

Até agora nossos programas trabalharam apenas com valores conhecidos, escritos no próprio programa. No entanto, o melhor da programação é poder escrever a solução de um problema e aplicá-la várias vezes. Para isso, precisamos melhorar nossos programas de forma a permitir que novos valores sejam fornecidos durante sua execução, de modo que poderemos executá-los com valores diferentes sem alterar os programas em si.

Chamamos de entrada de dados o momento em que seu programa recebe dados ou valores por um dispositivo de entrada de dados (como o teclado do computador) ou de um arquivo em disco.

A função **input** é utilizada para solicitar dados do usuário. Ela recebe um parâmetro, que é a mensagem a ser exibida, e retorna o valor digitado pelo usuário. Vejamos um exemplo na listagem 3.16.

▶ **Listagem 3.16 – Entrada de dados**

```
x = input("Digite um número: ")
print(x)
```

▶ **Listagem 3.17 – Saída na tela, tendo o 5 como exemplo de número digitado pelo usuário**

```
Digite um número: 5
5
```

Vejamos outro exemplo na listagem 3.18.

▶ **Listagem 3.18 – Exemplo de entrada de dados**

```
nome = input("Digite seu nome:") ❶
print("Você digitou %s" % nome)
print("Olá, %s!" % nome)
```

Em ❶, solicitamos a entrada de dados, no caso, o nome do usuário. A mensagem "Digite seu nome:" é exibida, e o programa para até que o usuário digite ENTER. Só então o resto do programa é executado. Vejamos a saída de dados quando digitamos João como nome na listagem 3.19.

▶ **Listagem 3.19 – Resultado da entrada de dados**

```
Digite seu nome:João
Você digitou João
Olá, João!
```

Execute o programa outras vezes digitando, por exemplo, 123 como nome. Observe que o programa não se importa com os valores digitados pelo usuário. Essa verificação deve ser feita por seu programa. Veremos como fazer isso em outro capítulo, como validação de dados.

3.7.1 Conversão da entrada de dados

A função **input** sempre retorna valores do tipo string, ou seja, não importa se digitamos apenas números, o resultado sempre é string. Para resolver esse pequeno

problema, vamos utilizar a função `int` para converter o valor retornado em um número inteiro, e a função `float` para convertê-lo em número decimal ou de ponto flutuante. Vejamos outro exemplo usando essas funções, no qual devemos calcular o valor de um bônus por tempo de serviço na listagem 3.20.

▶ Listagem 3.20 – Cálculo de bônus por tempo de serviço

```
anos = int(input("Anos de serviço: "))
valor_por_ano = float(input("Valor por ano: "))
bônus = anos * valor_por_ano
print("Bônus de R$ %5.2f" % bônus)
```

Vejamos o resultado se testarmos 10 anos e R$ 25 por ano na tela da listagem 3.21. Observe que escrevemos apenas 25, e não R$ 25. Isso porque 25 é um número, e R$ 25 é uma string. Por enquanto, não vamos misturar dois tipos de dados na mesma entrada de dados, para simplificar nossos programas.

▶ Listagem 3.21 – Resultado do cálculo para 10 anos e R$ 25 por ano

```
Anos de serviço: 10
Valor por ano: 25
Bônus de R$ 250.00
```

Execute o programa novamente com outros valores. Experimente digitar uma letra em anos de serviço ou em valor por ano. Você receberá uma mensagem de erro, pois a conversão de letras em números não é automática. A próxima seção explica melhor o problema.

Exercício 3.7 Faça um programa que peça dois números inteiros. Imprima a soma desses dois números na tela.

Exercício 3.8 Escreva um programa que leia um valor em metros e o exiba convertido em milímetros.

Exercício 3.9 Escreva um programa que leia a quantidade de dias, horas, minutos e segundos do usuário. Calcule o total em segundos.

Exercício 3.10 Faça um programa que calcule o aumento de um salário. Ele deve solicitar o valor do salário e a porcentagem do aumento. Exiba o valor do aumento e do novo salário.

Exercício 3.11 Faça um programa que solicite o preço de uma mercadoria e o percentual de desconto. Exiba o valor do desconto e o preço a pagar.

Exercício 3.12 Escreva um programa que calcule o tempo de uma viagem de carro. Pergunte a distância a percorrer e a velocidade média esperada para a viagem.

Exercício 3.13 Escreva um programa que converta uma temperatura digitada em °C em °F. A fórmula para essa conversão é:

$$F = \frac{9 \times C}{5} + 32$$

Exercício 3.14 Escreva um programa que pergunte a quantidade de km percorridos por um carro alugado pelo usuário, assim como a quantidade de dias pelos quais o carro foi alugado. Calcule o preço a pagar, sabendo que o carro custa R$ 60 por dia e R$ 0,15 por km rodado.

Exercício 3.15 Escreva um programa para calcular a redução do tempo de vida de um fumante. Pergunte a quantidade de cigarros fumados por dia e quantos anos ele já fumou. Considere que um fumante perde 10 minutos de vida a cada cigarro, calcule quantos dias de vida um fumante perderá. Exiba o total em dias.

3.7.2 Erros comuns

A entrada de dados é um ponto frágil em nossos programas. Como não temos como prever o que o usuário vai digitar, temos que nos preparar para reconhecer os erros mais comuns. Vejamos um programa que lê três valores na listagem 3.22.

▶ Listagem 3.22 – Entrada de dados com conversão de tipos

```
nome = input("Digite seu nome: ")
idade = int(input("Digite sua idade: "))
saldo = float(input("Digite o saldo da sua conta bancária: "))
print(nome)
print(idade)
print(saldo)
```

A listagem 3.23 mostra o resultado da execução do programa quando todos os valores são digitados corretamente. Correto significa uma ausência de erros durante a função **input** ou durante a conversão do valor retornado pela função.

▶ Listagem 3.23 – Exemplo de entrada de dados

```
Digite seu nome: João
Digite sua idade: 42
Digite o saldo da sua conta bancária: 15756.34
João
42
15756.34
```

A listagem 3.24 mostra outro exemplo de entrada de dados bem-sucedida. Observe que, como utilizamos a função `float` para converter o saldo, mesmo inserindo 34 o valor foi convertido para 34.0.

▶ Listagem 3.24 – Exemplo de entrada de dados

```
Digite seu nome: Maria
Digite sua idade: 28
Digite o saldo da sua conta bancária: 34
Maria
28
34.0
```

Já na listagem 3.25, temos um exemplo de erro durante a entrada de dados. No caso, digitamos letras (abc) que não podem ser convertidas em um valor inteiro. Observe que o erro interrompe nosso programa, exibindo a linha em que ocorreu e o nome do erro. Nesse caso, o erro aconteceu na linha 2 e seu nome é `ValueError: invalid literal for int with base 10: 'abc'`.

▶ Listagem 3.25 – Erro de conversão

```
Digite seu nome: Minduim
Digite sua idade: abc
Traceback (most recent call last):
  File "input/input2.py", line 2, in <module>
    idade = int(input("Digite sua idade: "))
ValueError: invalid literal for int() with base 10: 'abc'
```

A listagem 3.26 mostra outro erro de conversão, mas, dessa vez, durante a conversão para número decimal, usando a função `float`. A entrada de dados é um pouco rústica, parando em caso de erro. Mais adiante, aprenderemos sobre exceções em Python e como tratar esse tipo de erro. Por enquanto, basta saber que não estamos validando a entrada e que nossos programas ainda são frágeis.

▶ Listagem 3.26 – Erro de conversão: letras no lugar de números

```
Digite seu nome: Juanito
Digite sua idade: 31
Digite o saldo da sua conta bancária: abc
Traceback (most recent call last):
  File "input/input2.py", line 3, in <module>
    saldo = float(input("Digite o saldo da sua conta bancária: "))
ValueError: could not convert string to float: abc
```

▶ Listagem 3.27 – Erro de conversão: vírgula no lugar de ponto

```
Digite seu nome: Mary
Digite sua idade: 25
Digite o saldo da sua conta bancária: 17,4
Traceback (most recent call last):
  File "input/input2.py", line 3, in <module>
    saldo = float(input("Digite o saldo da sua conta bancária: "))
ValueError: invalid literal for float(): 17,4
```

O erro mostrado na listagem 3.27 é muito comum em países onde se usa a vírgula e não o ponto como separador entre a parte inteira e fracionária de um número. Em Python, você deve sempre digitar valores decimais usando o ponto, e não a vírgula como em português. Assim, 17,4 é um valor inválido, pois deveria ter sido digitado como 17.4. Existem recursos em Python para resolver esse tipo de problema, mas ainda é cedo para abordarmos o assunto.

CAPÍTULO 4

Condições

Executar ou não executar? Eis a questão...

Nem sempre todas as linhas dos programas serão executadas. Muitas vezes, será mais interessante decidir que partes do programa devem ser executadas com base no resultado de uma condição. A base dessas decisões consistirá em expressões lógicas que permitam representar escolhas em programas.

4.1 if

As condições servem para selecionar quando uma parte do programa deve ser ativada e quando deve ser simplesmente ignorada. Em Python, a estrutura de decisão é o **`if`**. Seu formato é apresentado na listagem 4.1.

▶ Listagem 4.1 – Formato da estrutura de condicional if

```
if <condição>:
   bloco verdadeiro
```

O **`if`** nada mais é que o nosso "se". Poderemos então entendê-lo em português da seguinte forma: se a condição for verdadeira, faça alguma coisa.

Vejamos um exemplo: ler dois valores e imprimir o maior deles, apresentado na listagem 4.2.

▶ Listagem 4.2 – Condições

```
a = int(input("Primeiro valor: "))
b = int(input("Segundo valor: "))
```

```
if a > b: ❶
   print ("O primeiro número é o maior!") ❷
if b > a: ❸
   print ("O segundo número é o maior!") ❹
```

Em ❶, temos a condição `a > b`. Essa expressão será avaliada, e, se o seu resultado for verdadeiro, a linha ❷ será executada. Se for falso, a linha ❷ será ignorada. O mesmo acontece para a condição `b > a` da linha ❸. Se o seu resultado for verdadeiro, a linha ❹ será executada. Se for falso, ignorada.

A sequência de execução do programa é alterada de acordo com os valores digitados como o primeiro e segundo valores. Digite o programa da listagem 4.2 e execute-o duas vezes. Na primeira vez, digite um valor maior primeiro e um menor em segundo. Na segunda vez, inverta esses valores e verifique se a mensagem na tela também mudou.

Quando o primeiro valor é maior que o segundo, temos as seguintes linhas sendo executadas: ❶, ❷, ❸. Quando o primeiro valor é menor que o segundo, temos outra sequência: ❶, ❸, ❹. É importante entender que a linha com a condição em si é executada mesmo se o resultado da expressão for falso.

As linhas ❶ e ❸ foram terminadas com o símbolo dois pontos (:). Quando isso acontece, temos o anúncio de um bloco de linhas a seguir. Em Python, um bloco é representado deslocando-se o início da linha para a direita. O bloco continua até a primeira linha com deslocamento diferente.

Observe que começamos a escrever a linha ❷ alguns caracteres mais à direita da linha anterior ❶ que iniciou o bloco. Como a linha ❸ foi escrita mais à esquerda, dizemos que o bloco da linha ❷ foi terminado.

TRÍVIA

Python é uma das poucas linguagens de programação que utiliza o deslocamento do texto à direita (recuo) para marcar o início e o fim de um bloco. Outras linguagens contam com palavras especiais para isso, como `BEGIN` e `END`, em Pascal; ou as famosas chaves (`{` e `}`), em C e Java.

Exercício 4.1 Analise o programa da listagem 4.2. Responda o que acontece se o primeiro e o segundo valores forem iguais? Explique.

Vejamos outro exemplo, onde solicitaremos a idade do carro do usuário e, em seguida, escreveremos novo se o carro tiver menos de três anos; ou velho, em caso contrário.

▶ Listagem 4.3 – Carro novo ou velho, dependendo da idade

```
idade = int(input("Digite a idade do seu carro: "))
if idade <= 3:
   print("Seu carro é novo") ❶
if idade > 3:
   print("Seu carro é velho") ❷
```

Execute o programa da listagem 4.3 e verifique o que aparece na tela. Você pode executá-lo várias vezes com as seguintes idades: 1, 3 e 5. A primeira condição é `idade <= 3`. Essa condição decide se a linha com função **`print`** ❶ será ou não executada. Como é uma condição simples, podemos entender que só exibiremos a mensagem do carro novo para as idades 0, 1, 2 e 3. A segunda condição, `idade > 3`, é o inverso da primeira. Se você observar de perto, não há um só número que torne ambas verdadeiras ao mesmo tempo. A segunda decisão é responsável por decidir a impressão da mensagem do carro velho ❷.

Embora óbvio que um carro não poderia ter valores negativos como idade, o programa não trata desse problema. Vamos alterá-lo mais adiante para verificar valores inválidos.

Exercício 4.2 Escreva um programa que pergunte a velocidade do carro de um usuário. Caso ultrapasse 80 km/h, exiba uma mensagem dizendo que o usuário foi multado. Nesse caso, exiba o valor da multa, cobrando R$ 5 por km acima de 80 km/h.

Um bloco de linhas em Python pode ter mais de uma linha, embora o último exemplo mostre apenas dois blocos com uma linha em cada. Se você precisar de duas ou mais no mesmo bloco, escreva essas linhas na mesma direção ou na mesma coluna da primeira linha do bloco. Isso basta para representá-lo.

Um problema comum é quando temos que pagar Imposto de Renda. Normalmente, pagamos o Imposto de Renda por faixa de salário. Imagine que para salários menores que R$ 1.000,00 não teríamos imposto a pagar, ou seja, alíquota 0%. Para salários entre R$ 1.000,00 e R$ 3.000,00 pagaríamos 20%. Acima desses valores, a alíquota seria de 35%. Esse problema se pareceria muito com o anterior, salvo

se o imposto não fosse cobrado diferentemente para cada faixa, ou seja, quem ganha R$ 4.000,00 tem os primeiros R$ 1.000,00 isentos de imposto; com o montante entre R$ 1.000,00 e R$ 3.000,00 pagando 20%, e o restante pagando os 35%. Vejamos a solução na listagem do programa 4.4.

▶ Listagem 4.4 – Cálculo do Imposto de Renda

```
salário=float(input("Digite o salário para cálculo do imposto: "))
base = salário ❶
imposto = 0
if base > 3000: ❷
   imposto = imposto + ((base - 3000) * 0.35) ❸
   base = 3000 ❹
if base > 1000: ❺
   imposto = imposto + ((base - 1000) * 0.20) ❻
print("Salário: R$%6.2f Imposto a pagar: R$%6.2f" % (salário, imposto))
```

O programa da listagem 4.4 é bem interessante. Tente executá-lo algumas vezes e compare o valor impresso com o valor calculado por você. Rastreie o programa e tente entender o que ele faz antes de ler o parágrafo seguinte. Verifique o que acontece para salários de R$ 500,00, R$ 1.000,00 e R$ 1,500,00.

Em ❶ temos a variável `base` recebendo uma cópia de `salário`. Isso é necessário porque, quando atribuímos um novo valor para uma variável, o valor anterior é substituído (e perdido se não o guardarmos em outro lugar). Como vamos utilizar o valor do salário digitado para exibi-lo na tela, não podemos perdê-lo; por isso, a necessidade de uma variável auxiliar chamada aqui de `base`.

Em ❷ verificamos se a `base` é maior que R$ 3.000,00. Se verdadeiro, executamos as linhas ❸ e ❹. Em ❸, calculamos 35% do valor superior a R$ 3.000,00. O resultado é armazenado na variável `imposto`. Como essa variável contém o valor a pagar para essa quantia, atualizaremos o valor de base para R$ 3.000,00 ❹, pois o que ultrapassa esse valor já foi tarifado.

Em ❺ verificamos se o valor de base é maior que R$ 1.000,00, calculando 20% de imposto em ❻, caso verdadeiro.

Vejamos o rastreamento para um salário de R$ 500,00:

salário	base	imposto
500	500	0

Para um salário de R$ 1.500,00:

salário	base	imposto
1500	1500	~~0~~
		100

Para um salário de R$ 3.000,00:

salário	base	imposto
3000	3000	~~0~~
		400

Para um salário de R$ 5.000,00:

salário	base	imposto
5000	~~5000~~	~~0~~
	3000	~~700~~
		1100

Exercício 4.3 Escreva um programa que leia três números e que imprima o maior e o menor.

Exercício 4.4 Escreva um programa que pergunte o salário do funcionário e calcule o valor do aumento. Para salários superiores a R$ 1.250,00, calcule um aumento de 10%. Para os inferiores ou iguais, de 15%.

4.2 else

Quando há problemas, como a mensagem do carro velho (Listagem 4.3), onde a segunda condição é simplesmente o inverso da primeira, podemos usar outra forma de **`if`** para simplificar os programas. Essa forma é a cláusula **`else`** para especificar o que fazer caso o resultado da avaliação da condição seja falso, sem precisarmos de um novo **`if`**. Vejamos como ficaria o programa reescrito para usar **`else`** na listagem 4.5.

▶ **Listagem 4.5 – Carro novo ou velho, dependendo da idade com else**

```
idade = int(input("Digite a idade de seu carro: "))
if idade <= 3:
   print("Seu carro é novo")
else: ❶
   print("Seu carro é velho") ❷
```

Veja que em ❶ utilizamos ":" após **else**. Isso é necessário porque **else** inicia um bloco, da mesma forma que **if**. É importante notar que devemos escrever **else** na mesma coluna do **if**, ou seja, com o mesmo recuo. Assim, o interpretador reconhece que **else** se refere a um determinado **if**. Você obterá um erro caso não alinhe essas duas estruturas na mesma coluna.

A vantagem de usar **else** é deixar os programas mais claros, uma vez que podemos expressar o que fazer caso a condição especificada em **if** seja falsa. A linha ❷ só é executada se a condição `idade <= 3` for falsa.

Exercício 4.5 Execute o programa (Listagem 4.5) e experimente alguns valores. Verifique se os resultados foram os mesmos do programa anterior (Listagem 4.3).

Exercício 4.6 Escreva um programa que pergunte a distância que um passageiro deseja percorrer em km. Calcule o preço da passagem, cobrando R$ 0,50 por km para viagens de até de 200 km, e R$ 0,45 para viagens mais longas.

4.3 Estruturas aninhadas

Nem sempre nossos programas serão tão simples. Muitas vezes, precisaremos aninhar vários **if** para obter o comportamento desejado do programa. Aninhar, nesse caso, é utilizar um **if** dentro de outro.

Vejamos o exemplo de calcular a conta de um telefone celular da empresa Tchau. Os planos da empresa Tchau são bem interessantes e oferecem preços diferenciados de acordo com a quantidade de minutos usados por mês. Abaixo de 200 minutos, a empresa cobra R$ 0,20 por minuto. Entre 200 e 400 minutos, o preço é de R$ 0,18. Acima de 400 minutos, o preço por minuto é de R$ 0,15. O programa da listagem 4.6 resolve esse problema.

▶ Listagem 4.6 – Conta de telefone com três faixas de preço

```
minutos=int(input("Quantos minutos você utilizou este mês:"))
if minutos < 200: ❶
   preço = 0.20 ❷
else:
   if minutos < 400: ❸
      preço = 0.18 ❹
   else: ❺
      preço = 0.15 ❻
print("Você vai pagar este mês: R$%6.2f" % (minutos * preço))
```

Em ❶, temos a primeira condição: `minutos < 200`. Se a quantidade de minutos for menor que 200, atribuímos 0,20 ao preço em ❷. Até aqui, nada de novo. Observe que **`if`** de ❸ está dentro de **`else`** da linha anterior: dizemos que está aninhado dentro de **`else`**. A condição de ❸, `minutos < 400`, decide se vamos executar a linha de ❹ ou a de ❻. Observe que **`else`** de ❺ está alinhado com **`if`** de ❸. No final, calculamos e imprimimos o preço na tela. Lembre-se de que o alinhamento do texto é muito importante em Python.

Vejamos, por exemplo a situação em que cinco categorias são necessárias. Façamos um programa que leia a categoria de um produto e determine o preço pela tabela 4.1.

Tabela 4.1 – Categorias de produto e preço

Categoria	Preço
1	10,00
2	18,00
3	23,00
4	26,00
5	31,00

À esquerda da listagem 4.7, você encontrará os números de linha do programa, numeradas de 1 a 19. Esses números servem apenas para ajudar o entendimento da explicação a seguir: lembre-se de não digitá-los.

▶ Listagem 4.7 – Categoria x preço

```
1  categoria = int(input("Digite a categoria do produto:"))
2  if categoria == 1:
3     preço = 10
4  else:
5     if categoria == 2:
6        preço = 18
7     else:
8        if categoria == 3:
9           preço = 23
10       else:
11          if categoria == 4:
12             preço = 26
13          else:
14             if categoria == 5:
15                preço = 31
16             else:
17                print("Categoria inválida, digite um valor entre 1 e 5!")
18                preço = 0
19 print("O preço do produto é: R$%6.2f" % preço)
```

Observe que o alinhamento se tornou um grande problema, uma vez que tivemos que deslocar à direita a cada **else**.

No programa da listagem 4.7, introduzimos o conceito de validação da entrada. Dessa vez, se o usuário digitar um valor inválido, receberá uma mensagem de erro na tela. Nada muito prático ou bonito.

Vejamos a execução das linhas dependendo da categoria digitada na tabela 4.2.

Tabela 4.2 – Linhas executadas

Categoria	Linhas executadas
1	1,2,3,19
2	1,2,4,5,6,19
3	1,2,4,5,7,8,9,19
4	1,2,4,5,7,8,10,11,12,19
5	1,2,4,5,7,8,10,11,13,14,15,19
outras	1,2,4,5,7,8,10,11,13,14,16,17,18,19

Quando lermos um programa com estruturas aninhadas, devemos prestar muita atenção para visualizar corretamente os blocos. Observe como o alinhamento é importante.

Exercício 4.7 Rastreie o programa da listagem 4.7. Compare seu resultado ao apresentado na tabela 4.2.

4.4 elif

Python apresenta uma solução muito interessante ao problema de múltiplos **if**s aninhados. A cláusula **elif** substitui um par **else if**, mas sem criar outro nível de estrutura, evitando problemas de deslocamentos desnecessários à direita.

Vamos revisitar o problema da listagem 4.7, dessa vez usando **elif**. Veja o resultado no programa da listagem 4.8.

▶ Listagem 4.8 – Categoria x preço, usando elif

```
categoria = int(input("Digite a categoria do produto:"))
if categoria == 1:
   preço = 10
elif categoria == 2:
   preço = 18
elif categoria == 3:
   preço = 23
elif categoria == 4:
   preço = 26
elif categoria == 5:
   preço = 31
else:
   print("Categoria inválida, digite um valor entre 1 e 5!")
   preço = 0
print("O preço do produto é: R$%6.2f" % preço)
```

Exercício 4.8 Escreva um programa que leia dois números e que pergunte qual operação você deseja realizar. Você deve poder calcular a soma (+), subtração (-), multiplicação (*) e divisão (/). Exiba o resultado da operação solicitada.

Exercício 4.9 Escreva um programa para aprovar o empréstimo bancário para compra de uma casa. O programa deve perguntar o valor da casa a comprar, o salário e a quantidade de anos a pagar. O valor da prestação mensal não pode ser superior a 30% do salário. Calcule o valor da prestação como sendo o valor da casa a comprar dividido pelo número de meses a pagar.

Exercício 4.10 Escreva um programa que calcule o preço a pagar pelo fornecimento de energia elétrica. Pergunte a quantidade de kWh consumida e o tipo de instalação: R para residências, I para indústrias e C para comércios. Calcule o preço a pagar de acordo com a tabela a seguir.

Preço por tipo e faixa de consumo		
Tipo	Faixa (kWh)	Preço
Residencial	Até 500	R$ 0,40
	Acima de 500	R$ 0,65
Comercial	Até 1000	R$ 0,55
	Acima de 1000	R$ 0,60
Industrial	Até 5000	R$ 0,55
	Acima de 5000	R$ 0,60

CAPÍTULO 5

Repetições

Repetições representam a base de vários programas. São utilizadas para executar a mesma parte de um programa várias vezes, normalmente dependendo de uma condição. Por exemplo, para imprimir três números na tela, poderíamos escrever um programa como o apresentado na listagem 5.1.

▶ Listagem 5.1 – Imprimindo de 1 a 3

```
print(1)
print(2)
print(3)
```

Podemos imaginar que para imprimir três números, começando de 1 até o 3, devemos variar `print(x)`, onde `x` varia de 1 a 3. Vejamos outra solução para o problema na listagem 5.2.

▶ Listagem 5.2 – Imprimindo de 1 a 3 usando uma variável

```
x=1
print(x)
x=2
print(x)
x=3
print(x)
```

Outra solução seria incrementar o valor de `x` após cada `print`. Vejamos essa solução na listagem 5.3.

▶ **Listagem 5.3 – Imprimindo de 1 a 3 incrementando**

```
x=1
print(x)
x=x+1
print(x)
x=x+1
print(x)
```

Porém, se o objetivo fosse escrever 100 números, a solução não seria tão agradável, pois teríamos que escrever pelo menos 200 linhas! A estrutura de repetição aparece para nos auxiliar a resolver esse tipo de problema.

Uma das estruturas de repetição do Python é o **`while`**, que repete um bloco enquanto a condição for verdadeira. Seu formato é apresentado na listagem 5.4, onde `condição` é uma expressão lógica, e `bloco` representa as linhas de programa a repetir enquanto o resultado da condição for verdadeiro.

▶ **Listagem 5.4 – Formato da estrutura de repetição com while**

```
while <condição>:
   bloco
```

Para resolver o problema de escrever três números utilizando o **`while`**, escreveríamos um programa como o da listagem 5.5.

▶ **Listagem 5.5 – Imprimindo de 1 a 3 com while**

```
x=1 ❶
while x<=3: ❷
   print(x) ❸
   x = x + 1 ❹
```

A execução desse programa seria um pouco diferente do que vimos até agora. Primeiramente, ❶ seria executada inicializando a variável `x` com o valor 1. A linha ❷ seria uma combinação de estrutura condicional com estrutura de repetição. Podemos entender a condição do **`while`** da mesma forma que a condição de **`if`**. A diferença é que, se a condição for verdadeira, repetiremos as linhas ❸ e ❹ (bloco) enquanto a avaliação da condição for verdadeira.

Em ❸ teremos a impressão na tela propriamente dita, onde `x` é 1. Em ❹ temos que o valor de `x` é acrescentado de 1. Como `x` vale 1, `x + 1` valerá 2. Esse novo valor é então atribuído a `x`. A parte nova é que a execução não termina após ❹ que é o

fim do bloco, mas retorna para ❷. É esse retorno que faz a estrutura de repetição especial.

Agora, `x` vale 2 e `x <= 3` continua verdadeiro (`True`), logo, o bloco será executado outra vez. ❸ realizará a impressão do valor 2, e ❹ atualizará o valor de `x` para `x + 1`; nesse caso, 2 + 1 = 3. A execução volta novamente para a linha ❷.

A condição em ❷ é avaliada, e como `x` vale 3, e `x <= 3` continua verdadeira, fazendo com que as linhas ❸ e ❹ sejam executadas, exibindo 3 e atualizando o valor de `x` para 4 (3 + 1).

Nesse ponto, ❷, temos que `x` vale 4 e que a condição `x <= 3` resulta em Falso (`False`), terminando, assim, a repetição do bloco.

Exercício 5.1 Modifique o programa para exibir os números de 1 a 100.

Exercício 5.2 Modifique o programa para exibir os números de 50 a 100.

Exercício 5.3 Faça um programa para escrever a contagem regressiva do lançamento de um foguete. O programa deve imprimir 10, 9, 8, ..., 1, 0 e Fogo! na tela.

5.1 Contadores

O poder das estruturas de repetições é muito interessante, principalmente quando utilizamos condições com mais de uma variável. Imagine um problema onde deveríamos imprimir os números inteiros entre 1 e um valor digitado pelo usuário. Vamos modificar o programa da listagem 5.5 de forma que o último número a imprimir seja informado pelo usuário. O programa já modificado é apresentado na listagem 5.6.

▶ Listagem 5.6 – Impressão de 1 até um número digitado pelo usuário

```
fim=int(input("Digite o último número a imprimir:")) ❶
x = 1
while x <= fim: ❷
    print(x) ❸
    x = x + 1 ❹
```

Nesse caso, o programa imprimirá de 1 até o valor digitado em ❶. Em ❷ utilizamos a variável `fim` para representar o limite de nossa repetição.

Agora vamos analisar o que realizamos com a variável `x` dentro da repetição. Em ❸, o valor de `x` é simplesmente impresso. Em ❹ atualizamos o valor de `x` com `x + 1`, ou seja, com o próximo valor inteiro. Quando realizamos esse tipo de operação dentro de uma repetição estamos contando. Logo diremos que `x` é um contador. Um contador é uma variável utilizada para contar o número de ocorrências de um determinado evento; nesse caso, o número de repetições do **`while`**, que satisfaz às necessidades de nosso problema.

Experimente esse programa com vários valores, primeiro digitando 5, depois 500 e, por fim, 0 (zero). Provavelmente, 5 e 500 produzirão os resultados esperados, ou seja, a impressão de 1 até 5, ou de 1 até 500. Porém, quando digitamos zero, nada acontece, e o programa termina logo a seguir, sem impressão.

Analisando nosso programa quando a variável `fim` vale 0, ou seja, quando digitamos 0 em ❶, temos que a condição em ❷ é `x <= fim`. Como `x` é 1 e `fim` é 0, temos que `1 <= 0` é falso desde a primeira execução, fazendo com que o bloco a repetir não seja executado, uma vez que sua condição de entrada é falsa. O mesmo aconteceria na inserção de valores negativos.

Imagine que o problema agora seja um pouco diferente: imprimir apenas os números pares entre 0 e um número digitado pelo usuário, de forma bem similar ao problema anterior. Poderíamos resolver o problema com um **`if`** para testar se `x` é par ou ímpar antes de imprimir. Vale lembrar que um número é par quando é 0 ou múltiplo de 2. Quando é múltiplo de 2, temos que o resto da divisão desse número por 2 é 0, ou seja, o resultado é uma divisão exata, sem resto. Em Python, podemos escrever esse teste com `x % 2 == 0` (resto da divisão de x por 2 é igual a zero), alterando o programa anterior para o da listagem 5.7.

▶ Listagem 5.7 – Impressão de números pares de 0 até um número digitado pelo usuário

```
fim=int(input("Digite o último número a imprimir:"))
x = 0 ❶
while x <= fim:
   if x % 2 == 0: ❷
      print(x) ❸
   x = x + 1
```

Veja que, para começar a imprimir do 0, e não de 1, modificamos ❶. Um detalhe importante é que ❸ é um bloco dentro de **`if`** ❷, sendo para isso deslocado à direita. Execute o programa e verifique seu resultado.

Agora, finalmente, estamos resolvendo o problema, mas poderíamos resolvê-lo de forma ainda mais simples se adicionássemos 2 a x a cada repetição. Isso garantiria que x sempre fosse par. Vejamos o programa da listagem 5.8.

▶ Listagem 5.8 – Impressão de números pares de 0 até um número digitado pelo usuário, sem if

```
fim=int(input("Digite o último número a imprimir:"))
x = 0
while x <= fim:
    print(x)
    x = x + 2
```

Esses dois exemplos mostram que existe mais de uma solução para o problema, que podemos escrever programas diferentes e obter a mesma solução. Essas soluções podem ser às vezes mais complicadas, às vezes mais simples, mas ainda assim corretas.

Exercício 5.4 Modifique o programa anterior para imprimir de 1 até o número digitado pelo usuário, mas, dessa vez, apenas os números ímpares.

Exercício 5.5 Reescreva o programa anterior para escrever os 10 primeiros múltiplos de 3.

Vejamos outro tipo de problema. Imagine ter que imprimir a tabuada de adição de um número digitado pelo usuário. Essa tabuada deve ser impressa de 1 a 10, sendo n o número digitado pelo usuário. Teríamos, assim, n+1, n+2, ... n+10. Confira a solução na listagem 5.9.

▶ Listagem 5.9 – Tabuada simples

```
n = int(input("Tabuada de:"))
x = 1
while x <= 10:
    print(n+x)
    x=x+1
```

Execute o programa anterior e experimente diversos valores.

Exercício 5.6 Altere o programa anterior para exibir os resultados no mesmo formato de uma tabuada: 2x1 = 2, 2x2=4, ...

Exercício 5.7 Modifique o programa anterior de forma que o usuário também digite o início e o fim da tabuada, em vez de começar com 1 e 10.

Exercício 5.8 Escreva um programa que leia dois números. Imprima o resultado da multiplicação do primeiro pelo segundo. Utilize apenas os operadores de soma e subtração para calcular o resultado. Lembre-se de que podemos entender a multiplicação de dois números como somas sucessivas de um deles. Assim, 4 × 5 = 5 + 5 + 5 + 5 = 4 + 4 + 4 + 4 + 4.

Exercício 5.9 Escreva um programa que leia dois números. Imprima a divisão inteira do primeiro pelo segundo, assim como o resto da divisão. Utilize apenas os operadores de soma e subtração para calcular o resultado. Lembre-se de que podemos entender o quociente da divisão de dois números como a quantidade de vezes que podemos retirar o divisor do dividendo. Logo, 20 ÷ 4 = 5, uma vez que podemos subtrair 4 cinco vezes de 20.

Contadores também podem ser úteis quando usados com condições dentro dos programas. Vejamos um programa para corrigir um teste de múltipla escolha com três questões. A resposta da primeira é "b"; da segunda, "a"; e da terceira, "d". O programa da listagem 5.10 conta um ponto a cada resposta correta.

▶ Listagem 5.10 – Contagem de questões corretas

```
pontos = 0
questão = 1
while questão <= 3:
    resposta = input("Resposta da questão %d: " % questão)
    if questão == 1 and resposta == "b":
        pontos = pontos + 1
    if questão == 2 and resposta == "a":
        pontos = pontos + 1
```

```
    if questão == 3 and resposta == "d":
        pontos = pontos + 1
    questão +=1
print("O aluno fez %d ponto(s)" % pontos)
```

Execute o programa e digite todas as respostas corretas, depois tente com respostas diferentes. Veja que estamos verificando apenas respostas simples de uma só letra e que consideramos apenas letras minúsculas. Em Python, uma letra minúscula é diferente de uma maiúscula. Se você digitar "A" na segunda questão em vez de "a", o programa não considerará essa resposta correta. Uma solução para esse tipo de problema é utilizar o operador lógico **or** e verificar a resposta maiúscula e minúscula. Por exemplo, `questão == 1 and (resposta == "b" or resposta == "B")`.

Exercício 5.10 Modifique o programa da listagem 5.10 para que aceite respostas com letras maiúsculas e minúsculas em todas as questões.

Embora essa verificação resolva o problema, veremos que, se digitarmos um espaço em branco antes ou depois da resposta, ela também será considerada errada. Sempre que trabalharmos com strings, esse tipo de problema deve ser controlado. Veremos mais sobre o assunto no capítulo 7.

5.2 Acumuladores

Nem só de contadores precisamos. Em programas para calcular o total de uma soma, por exemplo, precisaremos de acumuladores. A diferença entre um contador e um acumulador é que nos contadores o valor adicionado é constante e, nos acumuladores, variável. Vejamos um programa que calcule a soma de 10 números na listagem 5.11. Nesse caso, `soma` ❶ é um acumulador e `n` ❷ é um contador.

▶ Listagem 5.11 – Soma de 10 números

```
n = 1
soma = 0
while n <= 10:
    x = int(input("Digite o %d número:"%n))
    soma = soma + x ❶
    n = n + 1 ❷
print("Soma: %d"%soma)
```

Podemos definir a média aritmética como a soma de vários números divididos pela quantidade de números somados. Assim, se somarmos três números, 4, 5 e 6, teríamos a média aritmética como (4+5+6) / 3, onde 3 é a quantidade de números. Se chamarmos o primeiro número de n1, o segundo de n2, e o terceiro de n3, teremos (n1 + n2 + n3) / 3.

Vejamos um programa que calcula a média de cinco números digitados pelo usuário na listagem 5.12. Se chamarmos o primeiro valor digitado de n1, o segundo de n2, e assim sucessivamente, teremos que:

$$média = (n1+n2+n3+n4+n5)/5 = \frac{n1 + n2 + n3 + n4 + n5}{5}$$

Em vez de utilizarmos cinco variáveis, vamos acumular os valores à medida que são lidos.

▶ Listagem 5.12 – Cálculo de média com acumulador

```
x = 1
soma = 0 ❶
while x <= 5:
    n = int(input("%d Digite o número:" % x))
    soma = soma + n ❷
    x = x + 1
print("Média: %5.2f" % (soma/5)) ❸
```

Nesse caso, temos `x` sendo um contador e `n` o valor digitado pelo usuário. A variável `soma` é criada em ❶ e inicializada com 0. Diferentemente de `x`, que recebe 1 a cada passagem, a variável `soma`, em ❷, é adicionada do valor digitado pelo usuário. Podemos dizer que o incremento de `soma` não é um valor constante, pois varia com o valor digitado pelo usuário. Podemos também dizer que `soma` acumula os valores de `n` a cada repetição. Logo, diremos que a variável `soma` é um acumulador.

Acumuladores são muito interessantes quando não sabemos ou não conseguimos obter o total da soma pela simples multiplicação de dois números. No caso do cálculo da média, o valor de `n` pode ser diferente cada vez que o usuário digitar um valor.

Exercício 5.11 Escreva um programa que pergunte o depósito inicial e a taxa de juros de uma poupança. Exiba os valores mês a mês para os 24 primeiros meses. Escreva o total ganho com juros no período.

Exercício 5.12 Altere o programa anterior de forma a perguntar também o valor depositado mensalmente. Esse valor será depositado no início de cada mês, e você deve considerá-lo para o cálculo de juros do mês seguinte.

Exercício 5.13 Escreva um programa que pergunte o valor inicial de uma dívida e o juro mensal. Pergunte também o valor mensal que será pago. Imprima o número de meses para que a dívida seja paga, o total pago e o total de juros pago.

5.3 Interrompendo a repetição

Embora muito útil, a estrutura **`while`** só verifica sua condição de parada no início de cada repetição. Dependendo do problema, a habilidade de terminar **`while`** dentro do bloco a repetir pode ser interessante.

A instrução **`break`** é utilizada para interromper a execução de **`while`** independentemente do valor atual de sua condição. Vejamos o exemplo da leitura de valores até que digitemos 0 (zero) no programa da listagem 5.13.

▶ Listagem 5.13 – Interrompendo a repetição

```
s=0
while True: ❶
   v=int(input("Digite um número a somar ou 0 para sair:"))
   if v==0:
      break ❷
   s = s+v ❸
print(s) ❹
```

Nesse exemplo, substituímos a condição do **`while`** por **`True`** em ❶. Dessa forma, o **`while`** executará para sempre, pois o valor de sua condição de parada (**`True`**) é constante. Em ❷ temos a instrução **`break`** sendo ativada dentro de um **`if`**, especificamente quando `v` é zero. Porém, enquanto `v` for diferente de zero, a repetição continuará a somar `v` a `s` em ❸. Quando `v` for igual a zero (0), teremos ❷ sendo executada, terminando a repetição e transferindo a execução para ❹, que, então, exibe o valor de `s` na tela.

Exercício 5.14 Escreva um programa que leia números inteiros do teclado. O programa deve ler os números até que o usuário digite 0 (zero). No final da execução, exiba a quantidade de números digitados, assim como a soma e a média aritmética.

Exercício 5.15 Escreva um programa para controlar uma pequena máquina registradora. Você deve solicitar ao usuário que digite o código do produto e a quantidade comprada. Utilize a tabela de códigos abaixo para obter o preço de cada produto:

Código	Preço
1	0,50
2	1,00
3	4,00
5	7,00
9	8,00

Seu programa deve exibir o total das compras depois que o usuário digitar 0. Qualquer outro código deve gerar a mensagem de erro "Código inválido".

Vejamos como exemplo um programa que leia um valor e que imprima a quantidade de cédulas necessárias para pagar esse mesmo valor, apresentado na listagem 5.14. Para simplificar, vamos trabalhar apenas com valores inteiros e com cédulas de R$ 50, R$ 20, R$ 10, R$ 5 e R$ 1.

Exercício 5.16 Execute o programa (Listagem 5.14) para os seguintes valores: 501, 745, 384, 2, 7 e 1.

Exercício 5.17 O que acontece se digitarmos 0 (zero) no valor a pagar?

Exercício 5.18 Modifique o programa para também trabalhar com notas de R$ 100.

Exercício 5.19 Modifique o programa para aceitar valores decimais, ou seja, também contar moedas de 0,01, 0,02, 0,05, 0,10 e 0,50.

Exercício 5.20 O que acontece se digitarmos 0,001 no programa anterior? Caso ele não funcione, altere-o de forma a corrigir o problema.

▶ **Listagem 5.14 – Contagem de cédulas**

```
valor=int(input("Digite o valor a pagar:"))
cédulas=0
atual=50
apagar=valor
while True:
    if atual<=apagar:
        apagar-=atual
        cédulas+=1
    else:
        print("%d cédula(s) de R$%d" % (cédulas, atual))
        if apagar == 0:
            break
        if atual == 50:
            atual = 20
        elif atual == 20:
            atual = 10
        elif atual == 10:
            atual = 5
        elif atual == 5:
            atual = 1
        cédulas = 0
```

5.4 Repetições aninhadas

Podemos combinar vários **while** de forma a obter resultados mais interessantes, como a repetição com incremento de duas variáveis. Imagine imprimir as tabuadas de multiplicação de 1 a 10. Vejamos como fazer isso, lendo a listagem do programa 5.15.

▶ **Listagem 5.15 – Impressão de tabuadas**

```
tabuada=1
while tabuada <= 10: ❶
   número = 1 ❷
   while número <= 10: ❸
      print("%d x %d = %d" % (tabuada, número, tabuada * número))
      número+=1 ❹
   tabuada+=1 ❺
```

Em ❶ temos nosso primeiro **while**, criado para repetir seu bloco enquanto o valor de `tabuada` for menor ou igual a 10. Em ❷ temos a inicialização da variável `número` dentro do primeiro **while**. Isso é importante porque precisamos voltar a multiplicar por 1 a cada novo valor da variável `tabuada`. Finalmente, em ❸ temos o segundo **while** com a condição de parada `número <= 10`. Esse **while** executará suas repetições dentro do primeiro, ou seja, o ponto de execução passa de ❹ para ❸ enquanto a condição for verdadeira. Veja que em ❹ utilizamos o operador `+=` para representar `número = número + 1`. Quando `número` valer 11, a condição em ❸ resultará falsa, e a execução do programa continuará a partir da linha ❺. Em ❺ incrementamos o valor de `tabuada` e voltamos a ❶, onde será verificada a condição do primeiro **while**. Como resulta verdadeiro, voltaremos a executar ❷, reinicializando a variável `número` com o valor 1. ❷ é muito importante para que possamos novamente executar o segundo **while**, responsável por imprimir a tabuada na tela.

Vejamos o mesmo problema, mas sem utilizar repetições aninhadas, como apresenta a listagem 5.16.

▶ **Listagem 5.16 – Impressão de tabuadas sem repetições aninhadas**

```
tabuada=1
número=1
while tabuada <= 10:
   print("%d x %d = %d" % (tabuada, número, tabuada * número))
   número+=1
   if número == 11:
      número = 1
      tabuada+=1
```

Exercício 5.21 Reescreva o programa da listagem 5.14 de forma a continuar executando até que o valor digitado seja 0. Utilize repetições aninhadas.

Exercício 5.22 Escreva um programa que exiba uma lista de opções (menu): adição, subtração, divisão, multiplicação e sair. Imprima a tabuada da operação escolhida. Repita até que a opção saída seja escolhida.

Exercício 5.23 Escreva um programa que leia um número e verifique se é ou não um número primo. Para fazer essa verificação, calcule o resto da divisão do número por 2 e depois por todos os números ímpares até o número lido. Se o resto de uma dessas divisões for igual a zero, o número não é primo. Observe que 0 e 1 não são primos e que 2 é o único número primo que é par.

Exercício 5.24 Modifique o programa anterior de forma a ler um número `n`. Imprima os `n` primeiros números primos.

Exercício 5.25 Escreva um programa que calcule a raiz quadrada de um número. Utilize o método de Newton para obter um resultado aproximado. Sendo n o número a obter a raiz quadrada, considere a base b=2. Calcule p usando a fórmula p=(b+(n/b))/2. Agora, calcule o quadrado de p. A cada passo, faça b=p e recalcule p usando a fórmula apresentada. Pare quando a diferença absoluta entre n e o quadrado de p for menor que 0,0001.

Exercício 5.26 Escreva um programa que calcule o resto da divisão inteira entre dois números. Utilize apenas as operações de soma e subtração para calcular o resultado.

Exercício 5.27 Escreva um programa que verifique se um número é palíndromo. Um número é palíndromo se continua o mesmo caso seus dígitos sejam invertidos. Exemplos: 454, 10501

CAPÍTULO 6

Listas

Listas são um tipo de variável que permite o armazenamento de vários valores, acessados por um índice. Uma lista pode conter zero ou mais elementos de um mesmo tipo ou de tipos diversos, podendo inclusive conter outras listas. O tamanho de uma lista é igual à quantidade de elementos que ela contém.

Podemos imaginar uma lista como um edifício de apartamentos, onde o térreo é o andar zero, o primeiro andar é o andar 1 e assim por diante. O índice é utilizado para especificarmos o "apartamento" onde guardaremos nossos dados.

Em um prédio de seis andares, teremos números de andar variando entre 0 e 5. Se chamarmos nosso prédio de P, teremos P[0] como o endereço do térreo, P[1] como endereço do primeiro andar, continuando assim até P[5]. Em Python, P seria o nome da lista; e o número entre colchetes, o índice.

Listas são mais flexíveis que prédios e podem crescer ou diminuir com o tempo. Vejamos como criar uma lista em Python na listagem 6.1.

▶ Listagem 6.1 – Uma lista vazia

```
L=[]
```

Essa linha cria uma lista chamada `L` com zero elemento, ou seja, uma lista vazia. Os colchetes ([ ]) após o símbolo de igualdade servem para indicar que `L` é uma lista. Vejamos agora como criar uma lista `Z`, com 3 elementos na listagem 6.2.

▶ Listagem 6.2 – Uma lista com três elementos

```
Z=[ 15, 8, 9 ]
```

A lista `Z` foi criada com três elementos: 15, 8 e 9. Dizemos que o tamanho da lista `Z` é 3. Como o primeiro elemento tem índice 0, temos que o último elemento é `Z[2]`. Veja o resultado de testes com `Z` na listagem 6.3.

▶ **Listagem 6.3 – Acesso a uma lista**

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[1]
8
>>> Z[2]
9
```

Utilizando o nome da lista e um índice, podemos mudar o conteúdo de um elemento. Observe os testes no interpretador, apresentados na listagem 6.4.

▶ **Listagem 6.4 – Modificação de uma lista**

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[0]=7
>>> Z[0]
7
>>> Z
[7, 8, 9]
```

Quando criamos a lista `Z`, o primeiro elemento era o número 15. Por isso, `Z[0]` era 15. Quando executamos `Z[0]=7`, alteramos o conteúdo do primeiro elemento para 7. Isso pode ser verificado quando pedimos para exibir `Z`, agora com 7, 8 e 9 como elementos.

Vejamos um exemplo onde um aluno tem cinco notas e no qual desejamos calcular sua média aritmética. Veja o programa na listagem 6.5.

▶ **Listagem 6.5 – Cálculo da média**

```
notas=[6,7,5,8,9] ❶
soma=0
x=0
while x<5: ❷
   soma += notas[x] ❸
   x+=1 ❹
print("Média: %5.2f" % (soma/x))
```

Criamos a lista de notas em ❶. Em ❷, criamos a estrutura de repetição para variar o valor de `x` e continuar enquanto este for menor que 5. Lembre-se de que uma lista de cinco elementos contém índices de 0 a 4. Por isso inicializamos `x=0` na linha anterior. Em ❸, adicionamos o valor de `notas[0]` à soma e depois `notas[1]`, `notas[2]`, `notas[3]` e `notas[4]`, um elemento a cada repetição. Para isso, utilizamos o valor de `x` como índice e o incrementamos de 1 em ❹. Uma grande vantagem desse programa foi que não precisamos declarar cinco variáveis para guardar as cinco notas. Todas as notas foram armazenadas na lista, utilizando um índice para identificar ou acessar cada valor.

Vejamos uma modificação desse exemplo, mas, dessa vez, vamos ler as notas uma a uma. O programa modificado é apresentado na listagem 6.6.

▶ Listagem 6.6 – Cálculo da média com notas digitadas

```
notas=[0,0,0,0,0] ❶
soma=0
x=0
while x<5:
    notas[x]=float(input("Nota %d:" % x)) ❷
    soma += notas[x]
    x+=1
x=0 ❸
while x<5: ❹
    print("Nota %d: %6.2f" % (x, notas[x]))
    x+=1
print("Média: %5.2f" % (soma/x))
```

Em ❶ criamos a lista de notas com cinco elementos, todos zero. Em ❷, utilizamos a repetição para ler as notas do aluno e armazená-las na lista de notas. Veja que adicionamos `nota[x]` à `soma` já na linha seguinte. Terminada a primeira repetição, teremos a lista de notas preenchidas. Para imprimir a lista de notas, reinicializamos o valor da variável `x` para 0 ❸ e criamos outra estrutura de repetição ❹.

Exercício 6.1 Modifique o programa da listagem 6.6 para ler 7 notas em vez de 5.

6.1 Trabalhando com índices

Vejamos outro exemplo: um programa que lê cinco números, armazena-os em uma lista e depois solicita que o usuário escolha um número a mostrar. O objetivo é, por exemplo, ler 15, 12, 5, 7 e 9 e armazená-los na lista. Depois, se o usuário digitar 2, ele imprimirá o segundo número digitado, 3, o terceiro, e assim sucessivamente. Observe que o índice do primeiro número é 0, e não 1: essa pequena conversão será feita no programa da listagem 6.7.

▶ Listagem 6.7 – Apresentação de números

```
números=[0,0,0,0,0]
x=0
while x<5:
   números[x]=int(input("Número %d:" % (x+1))) ❶
   x+=1
while True:
   escolhido=int(input("Que posição você quer imprimir (0 para sair): "))
   if escolhido == 0:
      break
   print("Você escolheu o número: %d" % (números[escolhido-1])) ❷
```

Execute o programa da listagem 6.7 e experimente alguns valores. Observe que em ❶ adicionamos 1 a `x` para que possamos imprimir Número 1...5 e não a partir de 0. Isso é importante porque começar a contar de 0 não é natural para a maioria das pessoas. Veja que mesmo imprimindo `x+1` para o usuário, a atribuição é feita para `números[x]` porque nossas listas começam em 0. Em ❷, fizemos a operação inversa. Quando o usuário escolhe o número a imprimir, ele faz uma escolha entre 1 e 5. Como 1 é o elemento 0, 2, o elemento 1, e assim por diante. Diminuímos o valor da escolha de um para obtermos o índice de notas.

6.2 Cópia e fatiamento de listas

Embora listas em Python sejam um recurso muito poderoso, todo poder traz responsabilidades. Um dos efeitos colaterais de listas aparece quando tentamos fazer cópias. Vejamos um teste no interpretador na listagem 6.8.

▶ **Listagem 6.8 – Tentativa de copiar listas**

```
>>> L=[1,2,3,4,5]
>>> V=L
>>> L
[1, 2, 3, 4, 5]
>>> V
[1, 2, 3, 4, 5]
>>> V[0]=6
>>> V
[6, 2, 3, 4, 5]
>>> L
[6, 2, 3, 4, 5]
```

Veja que, ao modificarmos `V`, modificamos também o conteúdo de `L`. Isso porque uma lista em Python é um objeto e, quando atribuímos um objeto a outro, estamos apenas copiando a mesma referência da lista, e não seus dados em si. Nesse caso, `V` funciona como um apelido de `L`, ou seja, `V` e `L` são a mesma lista. Vejamos o que acontece no gráfico da figura 6.1.

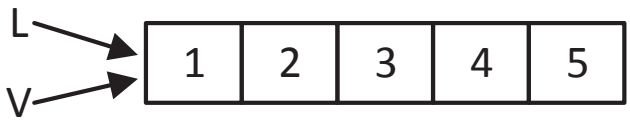

Figura 6.1 – Duas variáveis referenciando a mesma lista.

Quando modificamos `V[0]`, estamos modificando o mesmo valor de `L[0]`, pois ambos são referências, ou apelidos para a mesma lista na memória.

Dependendo da aplicação, esse efeito pode ser desejado ou não. Para criar uma cópia independente de uma lista, utilizaremos outra sintaxe. Vejamos o resultado das operações da listagem 6.9.

▶ **Listagem 6.9 – Cópia de listas**

```
>>> L=[1,2,3,4,5]
>>> V=L[:]
>>> V[0]=6
>>> L
[1, 2, 3, 4, 5]
>>> V
[6, 2, 3, 4, 5]
```

Ao escrevermos `L[:]`, estamos nos referindo a uma nova cópia de `L`. Assim `L` e `V` se referem a áreas diferentes na memória, permitindo alterá-las de forma independente, como na figura 6.2.

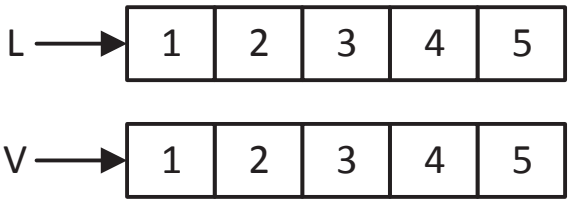

Figura 6.2 – Duas variáveis referenciando duas listas.

Podemos também fatiar uma lista, da mesma forma que fizemos com strings no capítulo 3. Vejamos alguns exemplos no interpretador na listagem 6.10.

▶ Listagem 6.10 – Fatiamento de listas

```
>>> L=[1,2,3,4,5]
>>> L[0:5]
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[:-1]
[1, 2, 3, 4]
>>> L[1:3]
[2, 3]
>>> L[1:4]
[2, 3, 4]
>>> L[3:]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[-1]
5
>>> L[-2]
4
```

Veja que índices negativos também funcionam. Um índice negativo começa a contar do último elemento, mas observe que começamos de `-1`. Assim `L[0]` representa o primeiro elemento; `L[-1]`, o último; `L[-2]`, o penúltimo, e assim por diante.

6.3 Tamanho de listas

Podemos usar a função `len` com listas. O valor retornado é igual ao número de elementos da lista. Veja alguns testes na listagem 6.11.

▶ Listagem 6.11 – Tamanho de listas

```
>>> L=[12,9,5]
>>> len(L)
3
>>> V=[]
>>> len(V)
0
```

A função `len` pode ser utilizada em repetições para controlar o limite dos índices. Veja o exemplo na listagem 6.12.

▶ Listagem 6.12 – Repetição com tamanho fixo da lista

```
L=[1,2,3]
x=0
while x < 3:
   print(L[x])
   x+=1
```

Isso pode ser reescrito como na listagem 6.13.

▶ Listagem 6.13 – Repetição com tamanho da lista usando len

```
L=[1,2,3]
x=0
while x < len(L):
   print(L[x])
   x+=1
```

A vantagem é que se trocarmos `L` para:

```
   L=[7,8,9,10,11,12]
```

o resto do programa continuaria funcionando, pois utilizamos a função `len` para calcular o tamanho da lista. Observe que o valor retornado pela função `len` é um número que não pode ser utilizado como índice, mas que é perfeito para testarmos os

limites de uma lista, como fizemos na listagem 6.13. Isso acontece porque `len` retorna a quantidade de elementos na lista e nossos índices começam a ser numerados de 0 (zero). Assim, os índices válidos de uma lista (`L`) variam de 0 até o valor de `len(L)-1`.

6.4 Adição de elementos

Uma das principais vantagens de trabalharmos com listas é poder adicionar novos elementos durante a execução do programa. Vejamos um teste no interpretador (Listagem 6.14).

Para adicionar um elemento ao fim da lista, utilizaremos o método `append`. Em Python, chamamos um método escrevendo o nome dele após o nome do objeto. Como listas são objetos, sendo `L` a lista, teremos `L.append(valor)`. Métodos são recursos de orientação a objetos, suportados e muito usados em Python. Você pode imaginar um método como uma função do objeto. Quando invocado, ele já sabe a que objeto estamos nos referindo, pois o informamos à esquerda do ponto. Falaremos mais sobre métodos no capítulo 10; por enquanto, observe como os utilizamos e saiba que são diferentes de funções.

▶ Listagem 6.14 – Adição de elementos à lista

```
>>> L=[]
>>> L.append("a")
>>> L
['a']
>>> L.append("b")
>>> L
['a', 'b']
>>> L.append("c")
>>> L
['a', 'b', 'c']
>>> len(L)
3
```

Vejamos um programa que lê números até que 0 seja digitado. Esse programa depois os imprimirá na mesma ordem em que foram digitados (Listagem 6.15).

▶ **Listagem 6.15 – Adição de elementos à lista**

```
L=[]
while True:
   n=int(input("Digite um número (0 sai):"))
   if n == 0:
      break
   L.append(n)
x=0
while x < len(L):
   print(L[x])
   x=x+1
```

Esse simples programa é capaz de ler e imprimir um número inicialmente indeterminado de valores. Isso é possível porque adicionamos elementos à lista `L`, conforme necessário.

Outra forma de adicionarmos elementos a uma lista é com adição de listas (Listagem 6.16).

Quando adicionamos apenas um elemento, tanto `L.append(1)` quanto `L+[1]` produzem o mesmo resultado. Perceba que em `L+[1]` escrevemos o elemento a adicionar dentro de uma lista (`[1]`), e que, em `append`, apenas 1. Isso porque, quando adicionamos uma lista a outra, o interpretador executa um método chamado `extend` que adiciona os elementos de uma lista a outra. Vejamos alguns exemplos na listagem 6.17.

▶ **Listagem 6.16 – Adição de listas**

```
>>> L=[]
>>> L=L+[1]
>>> L
[1]
>>> L+=[2]
>>> L
[1, 2]
>>> L+=[3,4,5]
>>> L
[1, 2, 3, 4, 5]
```

▶ Listagem 6.17 – Adição de elementos e listas

```
>>> L=["a"]
>>> L.append("b")
>>> L
['a', 'b']
>>> L.extend(["c"])
>>> L
['a', 'b', 'c']
>>> L.append(["d","e"])
>>> L
['a', 'b', 'c', ['d', 'e']]
>>> L.extend(["f","g","h"])
>>> L
['a', 'b', 'c', ['d', 'e'], 'f', 'g', 'h']
```

O método `extend` sequer aceita parâmetros que não sejam listas. Se você utilizar o método `append` com uma lista como parâmetro, em vez de adicionar os elementos no fim da lista, `append` adicionará a lista inteira, mas como apenas um novo elemento. Teremos então listas dentro de listas (Listagem 6.18).

▶ Listagem 6.18 – Adição de elementos e listas com append

```
>>> L=["a"]
>>> L.append(["b"])
>>> L.append(["c","d"])
>>> len(L)
3
>>> L[1]
['b']
>>> L[2]
['c', 'd']
>>> len(L[2])
2
>>> L[2][1]
'd'
```

Esse conceito é interessante, pois permite a utilização de estruturas de dados mais complexas, como matrizes, árvores e registros. Por enquanto, vamos utilizar esse recurso para armazenar múltiplos valores por elemento.

Exercício 6.2 Faça um programa que leia duas listas e que gere uma terceira com os elementos das duas primeiras.

Exercício 6.3 Faça um programa que percorra duas listas e gere uma terceira sem elementos repetidos.

6.5 Remoção de elementos da lista

Como o tamanho da lista pode variar, permitindo a adição de novos elementos, podemos também retirar alguns elementos da lista, ou mesmo todos eles. Para isso, utilizaremos a instrução **`del`** (Listagem 6.19).

▶ Listagem 6.19 – Remoção de elementos

```
>>> L=["a","b","c"]
>>> del L[1]
>>> L
['a', 'c']
>>> del L[0]
>>> L
['c']
```

É importante notar que o elemento excluído não ocupa mais lugar na lista, fazendo com que os índices sejam reorganizados, ou melhor, que passem a ser calculados sem esse elemento.

Podemos também apagar fatias inteiras de uma só vez (Listagem 6.20).

▶ Listagem 6.20 – Remoção de fatias

```
>>> L=list(range(101))
>>> del L[1:99]
>>> L
[0, 99, 100]
```

6.6 Usando listas como filas

Uma lista pode ser utilizada como fila se obedecermos a certas regras de inclusão e eliminação de elementos. Em uma fila, a inclusão é sempre realizada no fim, e as remoções são feitas no início. Dizemos que o primeiro a chegar é o primeiro a sair (*FIFO - First In First Out*).

É mais simples de entender se imaginarmos uma fila de banco. Quando a agência abre pela manhã, a fila está vazia. Quando os clientes começam a chegar, eles vão diretamente para o fim da fila. Os caixas então começam a atender esses clientes por ordem de chegada, ou seja, o cliente que chegou primeiro será atendido primeiro. Uma vez que o cliente é atendido, ele sai da fila. Então, um novo cliente passa a ser o primeiro da fila e o próximo a ser atendido.

Para escrevermos algo similar em Python, imaginaremos uma lista de clientes, representando a fila, onde o valor de cada elemento é igual à ordem de chegada do cliente. Vamos imaginar uma lista inicial com 10 clientes. Se outro cliente chegar, realizaremos um `append` para que ele seja inserido no fim da fila (`fila.append(último)`). Para retirarmos um cliente da fila e atendê-lo, poderíamos fazer `del fila[0]`, porém, isso apagaria o cliente da fila. Se quisermos retirá-lo da fila e, ao mesmo tempo, obter o elemento retirado, podemos utilizar o método pop `fila.pop(0)`. O método `pop` retorna o valor do elemento e o exclui da fila. Passamos 0 como parâmetro para indicar que queremos excluir o primeiro elemento. Veja o programa completo na listagem 6.21.

Exercício 6.4 O que acontece quando não verificamos se a lista está vazia antes de chamarmos o método `pop`?

Exercício 6.5 Altere o programa da listagem 6.21 de forma a poder trabalhar com vários comandos digitados de uma só vez. Atualmente, apenas um comando pode ser inserido por vez. Altere-o de forma a considerar operação como uma string.

Exemplo: FFFAAAS significaria três chegadas de novos clientes, três atendimentos e, finalmente, a saída do programa.

Exercício 6.6 Modifique o programa para trabalhar com duas filas. Para facilitar seu trabalho, considere o comando A para atendimento da fila 1; e B, para atendimento da fila 2. O mesmo para a chegada de clientes: F para fila 1; e G, para fila 2.

▶ **Listagem 6.21 – Simulação de uma fila de banco**

```
último = 10
fila = list(range(1,último+1))
while True:
   print("\nExistem %d clientes na fila" % len(fila))
   print("Fila atual:", fila)
   print("Digite F para adicionar um cliente ao fim da fila,")
   print("ou A para realizar o atendimento. S para sair.")
   operação = input("Operação (F, A ou S):")
   if operação == "A":
      if(len(fila))>0:
         atendido = fila.pop(0)
         print("Cliente %d atendido" % atendido)
      else:
         print("Fila vazia! Ninguém para atender.")
   elif operação == "F":
      último+=1 # Incrementa o ticket do novo cliente
      fila.append(último)
   elif operação == "S":
      break
   else:
      print("Operação inválida! Digite apenas F, A ou S!")
```

6.7 Uso de listas como pilhas

Uma pilha tem uma política de acesso bem definida: novos elementos são adicionados ao topo. A retirada de elementos também é feita pelo topo.

Imagine uma pilha de pratos para lavar. Retiramos o prato que está no topo da pilha para lavar e, se mais alguns pratos chegarem, serão também adicionados ou empilhados ao topo, ou seja, um sobre o outro. Na pilha, o último elemento a chegar é o primeiro a sair (*LIFO - Last In First Out*). Por enquanto, vamos nos concentrar nas pilhas de prato. Veja o programa da listagem 6.22, que simula uma pia de cozinha cheia de pratos.

Você deve ter percebido que o exemplo da listagem 6.22 é muito parecido com o programa da listagem 6.21. Isso porque a grande diferença entre pilhas e filas é o

elemento que escolhemos para retirar. Em uma fila, o primeiro elemento é retirado primeiro. Já em pilhas, retira-se a partir do último elemento. A única mudança em Python é o valor que passamos para o método `pop`. No caso de uma pilha, como retiramos o último elemento, passamos `-1` a `pop`.

▶ Listagem 6.22 – Pilha de pratos

```
prato = 5
pilha = list(range(1,prato+1))
while True:
   print("\nExistem %d pratos na pilha" % len(pilha))
   print("Pilha atual:", pilha)
   print("Digite E para empilhar um novo prato,")
   print("ou D para desempilhar. S para sair.")
   operação = input("Operação (E, D ou S):")
   if operação == "D":
      if(len(pilha))>0:
         lavado = pilha.pop(-1)
         print("Prato %d lavado" % lavado)
      else:
         print("Pilha vazia! Nada para lavar.")
   elif operação == "E":
      prato+=1 # Novo prato
      pilha.append(prato)
   elif operação == "S":
      break
   else:
      print("Operação inválida! Digite apenas E, D ou S!")
```

Você pode também observar o uso de pilhas com seu browser de internet. Abra algumas páginas, clicando em seus links. Observe como o histórico de navegação é modificado. Cada novo link adiciona uma página a seu histórico de navegação. Se você clicar em voltar uma página, o browser utilizará a última página que entrou no histórico. Nesse caso, o histórico do browser funciona como uma pilha. Na realidade, ele é um pouco mais complexo, pois permite que voltemos várias vezes e que depois possamos voltar no outro sentido se necessário (avançar). Agora faça outro teste: volte duas ou três vezes e depois visite um novo link. Dessa vez, o browser deve ter desativado a função de avanço, uma vez que o histórico mudou

(você mudou de direção). Tente entender esse processo como duas pilhas, uma à esquerda e outra à direta. Quando você escolhe voltar, desempilhamos um elemento da pilha à esquerda. Ao visitar um novo link, adicionamos um novo elemento a essa mesma pilha. Para simular a opção de avançar, imagine que, ao retirarmos um elemento da pilha à esquerda, devemos adicioná-lo à da direita. Se escolhermos um endereço novo, apagamos a pilha da direita.

Essas mesmas operações podem ser simuladas com outras estruturas, como uma lista simples e uma variável contendo onde estamos no histórico de navegação. A cada novo endereço, adicionaríamos um novo elemento à lista e atualizaríamos nossa posição. Ao voltarmos, decrementaríamos a posição e a incrementaríamos no caso de avanço. Se mudássemos de direção, bastaria apagar os elementos entre a posição atual e o fim da lista antes de adicionar o novo elemento.

A importância de aprender a manipular listas como filas ou pilhas é entender algoritmos mais complexos no futuro.

Exercício 6.7 Faça um programa que leia uma expressão com parênteses. Usando pilhas, verifique se os parênteses foram abertos e fechados na ordem correta. Exemplo:

```
(())        OK
()(()())    OK
())         Erro
```

Você pode adicionar elementos à pilha sempre que encontrar abre parênteses e desempilhá-la a cada fecha parênteses. Ao desempilhar, verifique se o topo da pilha é um abre parênteses. Se a expressão estiver correta, sua pilha estará vazia no final.

6.8 Pesquisa

Podemos pesquisar se um elemento está ou não em uma lista, verificando do primeiro ao último elemento se o valor procurado estiver presente. Vejamos a listagem 6.23.

▶ Listagem 6.23 – Pesquisa sequencial

```
L=[15,7,27,39]
p=int(input("Digite o valor a procurar:"))
achou=False ❶
```

```
x=0
while x<len(L):
   if L[x]==p:
      achou=True ❷
      break ❸
   x+=1
if achou: ❹
   print("%d achado na posição %d" % (p,x))
else:
   print("%d não encontrado" % p)
```

A pesquisa simplesmente compara todos os elementos da lista com o valor procurado, interrompendo a repetição ao encontrar o primeiro elemento cujo valor é igual ao procurado. É importante saber que podemos ou não encontrar o que procuramos, daí a utilização da variável `achou` em ❶. Essa variável do tipo lógico (booleano) será utilizada para verificar se saímos da repetição por termos achado o que procurávamos ou simplesmente porque visitamos todos os elementos sem encontrar o valor procurado. Veja que `achou` é marcada como `True` em ❷, mas dentro de `if` com a condição de pesquisa, e antes do `break` em ❸. Dessa forma, `achou` só será `True` se algum elemento for igual ao valor procurado. Em ❹, verificamos o valor de achou para decidir o que vamos imprimir.

Exercício 6.8 Modifique o primeiro exemplo (Listagem 6.23) de forma a realizar a mesma tarefa, mas sem utilizar a variável `achou`. Dica: observe a condição de saída do `while`.

Exercício 6.9 Modifique o exemplo para pesquisar dois valores. Em vez de apenas `p`, leia outro valor `v` que também será procurado. Na impressão, indique qual dos dois valores foi achado primeiro.

Exercício 6.10 Modifique o programa do exercício 6.9 de forma a pesquisar `p` e `v` em toda a lista e informando o usuário a posição onde `p` e a posição onde `v` foram encontrados.

6.9 Usando for

Python apresenta uma estrutura de repetição especialmente projetada para percorrer listas. A instrução **for** funciona de forma parecida a **while**, mas a cada repetição utiliza um elemento diferente da lista.

A cada repetição, o próximo elemento da lista é utilizado, o que se repete até o fim da lista. Vamos escrever um programa que utilize **for** para imprimir todos os elementos de uma lista (Listagem 6.24).

▶ Listagem 6.24 – Impressão de todos os elementos da lista com for

```
L=[8,9,15]
for e in L: ❶
   print(e) ❷
```

Quando começamos a executar o **for** em ❶, temos `e` igual ao primeiro elemento da lista, no caso, 8, ou seja, `L[0]`. Em ❷ imprimimos 8, e a execução do programa volta para ❶, onde `e` passa a valer 9, ou seja, `L[1]`. Na próxima repetição `e` valerá 15, ou seja, `L[2]`. Depois de imprimir o último número, a repetição é concluída, pois não temos mais elementos a substituir. Se tivéssemos que fazer a mesma tarefa com **while**, teríamos que escrever um programa como o da listagem 6.25.

▶ Listagem 6.25 – Impressão de todos os elementos da lista com while

```
L=[8,9,15]
x=0
while x<len(L):
   e=L[x]
   print(e)
   x+=1
```

Embora a instrução **for** facilite nosso trabalho, ela não substitui completamente **while**. Dependendo do problema, utilizaremos **for** ou **while**. Normalmente utilizaremos **for** quando quisermos processar os elementos de uma lista, um a um. **while** é indicado para repetições nas quais não sabemos ainda quantas vezes vamos repetir ou onde manipulamos os índices de forma não sequencial.

Vale lembrar que a instrução **break** também interrompe o **for**. Vejamos a pesquisa, escrita usando **for**, na listagem 6.26.

▶ **Listagem 6.26 – Pesquisa usando for**

```
L=[7,9,10,12]
p=int(input("Digite um número a pesquisar:"))
for e in L:
   if e == p:
      print("Elemento encontrado!")
      break ❶
else: ❷
   print("Elemento não encontrado.")
```

Utilizamos a instrução **break** para interromper a busca depois de encontrarmos o primeiro elemento em ❶. Em ❷ utilizamos um **else**, parecido com o da instrução **if**, para imprimir a mensagem informando que o elemento não foi encontrado. O **else** deve ser escrito na mesma coluna de **for** e só será executado se todos os elementos da lista forem visitados, ou seja, se não utilizarmos a instrução **break**, deixando **for** terminar normalmente.

Exercício 6.11 Modifique o programa da listagem 6.15 usando **for**. Explique por que nem todos os **while** podem ser transformados em **for**.

6.10 Range

Podemos utilizar a função `range` para gerar listas simples. A função `range` não retorna uma lista propriamente dita, mas um gerador ou *generator*. Por enquanto, basta entender como podemos usá-la. Imagine um programa simples que imprime de 0 a 9 na tela (Listagem 6.27).

▶ **Listagem 6.27 – Uso da função range**

```
for v in range(10):
   print(v)
```

Execute o programa e veja o resultado. A função `range` gerou números de 0 a 9 porque passamos 10 como parâmetro. Ela normalmente gera valores a partir de 0, logo, ao especificarmos apenas 10, estamos apenas informando onde parar. Experimente mudar de 10 para 20. O programa deve imprimir de 0 a 19. Agora

experimente 20000. A vantagem de utilizarmos a função `range` é gerar listas eficientemente, como mostrado no exemplo, sem precisar escrever os 20.000 valores no programa.

Com a mesma função `range`, podemos também indicar qual é o primeiro número a gerar. Para isso, utilizaremos dois parâmetros: início e fim (Listagem 6.28).

▶ Listagem 6.28 – Uso da função range com intervalos

```
for v in range(5, 8):
   print(v)
```

Usando 5 como início e 8 como fim, vamos imprimir os números 5, 6 e 7. A notação aqui para o fim é a mesma utilizada com fatias, ou seja, o fim é um intervalo aberto, isto é, não incluso na faixa de valores.

Se acrescentarmos um terceiro parâmetro à função `range`, teremos como saltar entre os valores gerados, por exemplo, `range(0,10,2)` gera os pares entre 0 e 10, pois começa de 0 e adiciona 2 a cada elemento. Vejamos um exemplo onde geramos os 10 primeiros múltiplos de 3 (Listagem 6.29).

▶ Listagem 6.29 – Uso da função range com saltos

```
for t in range(3,33,3):
   print(t, end=" ")
print()
```

Observe que um gerador como o retornado pela função `range` não é exatamente uma lista. Embora seja usado de forma parecida, é, na realidade, um objeto de outro tipo. Para transformar um gerador em lista, utilize a função `list` (Listagem 6.30).

▶ Listagem 6.30 – Transformação do resultado de range em uma lista

```
L=list(range(100,1100,50))
print(L)
```

Voltando à listagem 6.29, observe que utilizamos uma construção especial com a função **`print`**, onde `t` é o valor que queremos imprimir, mas com `end=" "`, que indica a função para não pular de linha após a impressão. `end` é, na realidade, um parâmetro opcional da função **`print`**. Veremos mais sobre isso quando estudarmos funções no capítulo 8. Veja também que, para saltar a linha no final do programa, fizemos uma chamada a **`print()`** sem qualquer parâmetro.

6.11 Enumerate

Com a função `enumerate` podemos ampliar as funcionalidades de **`for`** facilmente. Vejamos como imprimir uma lista, onde teremos o índice entre colchetes e o valor à sua direita (Listagem 6.31).

▶ Listagem 6.31 – Impressão de índices sem usar a função enumerate

```
L=[5,9,13]
x=0
for e in L:
   print("[%d] %d" % (x,e))
   x+=1
```

Veja o mesmo programa, mas utilizando a função `enumerate` na listagem (Listagem 6.32).

▶ Listagem 6.32 – Impressão de índices usando a função enumerate

```
L=[5,9,13]
for x, e in enumerate(L):
   print("[%d] %d" % (x,e))
```

A função `enumerate` gera uma tupla em que o primeiro valor é o índice e o segundo é o elemento da lista sendo enumerada. Ao utilizarmos `x, e` em **`for`**, indicamos que o primeiro valor da tupla deve ser colocado em `x`, e o segundo, em `e`. Assim, na primeira iteração teremos a tupla (0,5), onde `x=0` e `e=5`. Isso é possível porque Python permite o desempacotamento de valores de uma tupla, atribuindo um elemento da tupla a cada variável em **`for`**. O que temos a cada iteração de **`for`** é equivalente a `x,e = (0,5)`, em que o gerador `enumerate` retorna cada vez uma nova tupla. Os próximos valores retornados são (1,9) e (2,13), respectivamente. Experimente substituir `x,e` na listagem 6.30 por `z`. Antes de **`print`**, faça `x,e = z`. Adicione mais um **`print`** para exibir também o valor de `z`.

6.12 Operações com listas

Podemos percorrer uma lista de forma a verificar o menor e o maior valor (Listagem 6.33).

▶ **Listagem 6.33 – Verificação do maior valor**

```
L=[1,7,2,4]
máximo=L[0] ❶
for e in L:
   if e > máximo:
      máximo = e
print(máximo)
```

Em ❶, utilizamos um pequeno truque, inicializando o máximo com o valor do primeiro elemento. Precisamos de um valor para máximo antes de utilizá-lo na comparação com **if**. Se usássemos 0, não teríamos problema, desde que nossa lista não tenha valores negativos.

Exercício 6.12 Altere o programa da listagem 6.33 de forma a imprimir o menor elemento da lista.

Exercício 6.13 A lista de temperaturas de Mons, na Bélgica, foi armazenada na lista T = [-10, -8, 0, 1, 2, 5, -2, -4]. Faça um programa que imprima a menor e a maior temperatura, assim como a temperatura média.

6.13 Aplicações

Vejamos uma situação na qual temos que selecionar os elementos de uma lista de forma a copiá-los para outras duas listas. Para simplificar o problema, imagine que os valores estejam inicialmente na lista `V`, mas que devam ser copiados para a `P`, se forem pares; ou para a `I`, se forem ímpares. Veja o programa que resolve esse problema, na listagem 6.34.

▶ **Listagem 6.34 – Cópia de elementos para outras listas**

```
V=[9,8,7,12,0,13,21]
P=[]
I=[]
for e in V:
   if e % 2 == 0:
      P.append(e)
```

```
    else:
        I.append(e)
print("Pares: ", P)
print("Impares: ", I)
```

Vejamos agora um programa que controla a utilização das salas de um cinema. Imagine que a lista `Salas = [10,2,1,3,0]` contenha o número de lugares vagos nas salas 1, 2, 3, 4 e 5, respectivamente. Esse programa lerá o número da sala e a quantidade de lugares solicitados. Ele deve informar se é possível vender o número de lugares solicitados, ou seja, se ainda há lugares livres. Caso seja possível vender os bilhetes, atualizará o número de lugares livres. A saída ocorre quando se digita 0 no número da sala (Listagem 6.35).

▶ Listagem 6.35 – Controle da utilização de salas de um cinema

```
lugares_vagos=[10,2,1,3,0]
while True:
    sala=int(input("Sala (0 sai): "))
    if sala == 0:
        print("Fim")
        break
    if sala>len(lugares_vagos) or sala<1:
        print("Sala inválida")
    elif lugares_vagos[sala-1]==0:
        print("Desculpe, sala lotada!")
    else:
        lugares =int(input("Quantos lugares você deseja (%d vagos):"
            % lugares_vagos[sala-1]))
        if lugares > lugares_vagos[sala-1]:
            print("Esse número de lugares não está disponível.")
        elif lugares < 0:
            print("Número inválido")
        else:
            lugares_vagos[sala-1]-=lugares
            print("%d lugares vendidos" % lugares)
print("Utilização das salas")
for x,l in enumerate(lugares_vagos):
    print("Sala %d - %d lugar(es) vazio(s)" % (x+1, l))
```

6.14 Listas com strings

No capítulo 3 vimos que strings podem ser indexadas ou acessadas letra por letra. Listas em Python funcionam da mesma forma, permitindo o acesso a vários valores e se tornando uma das principais estruturas de programação da linguagem.

Vejamos agora outro exemplo de listas, mas utilizando strings, na listagem 6.36. Nesse caso, `S` é uma lista, e cada elemento, uma string.

▶ Listagem 6.36 – Listas com strings

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(len(S))
3
>>> print(S[0])
maçãs
>>> print(S[1])
peras
>>> print(S[2])
kiwis
```

O programa da listagem 6.37 lê e imprime uma lista de compras até que seja digitado fim.

▶ Listagem 6.37 – Lendo e imprimindo uma lista de compras

```
compras=[]
while True:
   produto=input("Produto:")
   if produto == "fim":
      break
   compras.append(produto)
for p in compras:
   print(p)
```

6.15 Listas dentro de listas

Um fator interessante é que podemos acessar as strings dentro da lista, letra por letra, usando um segundo índice. Vejamos as listagens 6.38 e 6.39.

▶ **Listagem 6.38 – Listas com strings, acessando letras**

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(S[0][0])
m
>>> print(S[0][1])
a
>>> print(S[1][1])
e
>>> print(S[2][2])
w
```

▶ **Listagem 6.39 – Impressão de uma lista de strings, letra a letra**

```
L=["maçãs", "peras", "kiwis"]
for s in L:
   for letra in s:
      print(letra)
```

Isso nos leva a outra vantagem das listas em Python: listas dentro de listas. Como bônus, temos também que os elementos de uma lista não precisam ser do mesmo tipo. Vejamos um exemplo onde teríamos uma lista de compras na listagem 6.40. O primeiro elemento seria o nome do produto; o segundo, a quantidade; e o terceiro, seu preço.

▶ **Listagem 6.40 – Listas com elementos de tipos diferentes**

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera",  5, 0.75]
produto3 = [ "kiwi",  4, 0.98]
```

Assim, `produto1`, `produto2`, `produto3` seriam três listas com três elementos cada uma. Observe que o primeiro elemento é do tipo string; o segundo, do tipo inteiro; e o terceiro, do tipo ponto flutuante (*float*)!

Vejamos agora outra lista na listagem 6.41.

▶ **Listagem 6.41 – Listas de listas**

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera",  5, 0.75]
produto3 = [ "kiwi",  4, 0.98]
```

```
compras = [ produto1, produto2, produto3]
print(compras)
```

Agora temos uma lista chamada `compras`, também com três elementos, mas cada elemento é uma lista à parte. Para imprimir essa lista, teríamos o programa da listagem 6.42.

▶ Listagem 6.42 – Impressão das compras

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera",  5, 0.75]
produto3 = [ "kiwi",  4, 0.98]
compras = [ produto1, produto2, produto3]
for e in compras:
   print("Produto: %s" % e[0])
   print("Quantidade: %d" % e[1])
   print("Preço: %5.2f" % e[2])
```

Da mesma forma, poderíamos ter acessado o preço do segundo elemento com `compras[1][2]`. Vejamos agora um programa completo, capaz de perguntar nome do produto, quantidade e preço e, no final, imprimir uma lista de compras completa.

▶ Listagem 6.43 – Criação e impressão da lista de compras

```
compras = [  ]
while True:
   produto = input("Produto: ")
   if produto == "fim":
      break
   quantidade = int(input("Quantidade: "))
   preço = float(input("Preço: "))
   compras.append([produto, quantidade, preço])
soma = 0.0
for e in compras:
   print("%20s x%5d %5.2f %6.2f" % (e[0],
                e[1],e[2],
                e[1] * e[2]))
   soma += e[1] * e[2]
print("Total: %7.2f" % soma)
```

6.16 Ordenação

Até agora, os elementos de nossas listas apresentam a mesma ordem em que foram digitados, sem qualquer ordenação. Para ordenar uma lista, realizaremos uma operação semelhante à da pesquisa, mas trocando a ordem dos elementos quando necessário. Um algoritmo muito simples de ordenação é o "*Bubble Sort*", ou método de bolhas, fácil de entender e aprender. Por ser lento, você não deve utilizá-lo com listas grandes.

A ordenação pelo método de bolhas consiste em comparar dois elementos a cada vez. Se o valor do primeiro elemento for maior que o do segundo, eles trocarão de posição. Essa operação é então repetida para o próximo elemento até o fim da lista. O método de bolhas exige que percorramos a lista várias vezes. Por isso, utilizaremos um marcador para saber se chegamos ao fim da lista trocando ou não algum elemento. Esse método tem outra propriedade, que é posicionar o maior elemento na última posição da lista, ou seja, sua posição correta. Isso permite eliminar um elemento do fim da lista a cada passagem completa. Vejamos o programa da listagem 6.44.

▶ Listagem 6.44 – Ordenação pelo método de bolhas

```
L=[7,4,3,12,8]
fim=5 ❶
while fim > 1: ❷
   trocou=False ❸
   x=0 ❹
   while x<(fim-1): ❺
      if L[x] > L[x+1]: ❻
         trocou=True ❼
         temp=L[x] ❽
         L[x]=L[x+1]
         L[x+1]=temp
      x+=1
   if not trocou: ❾
      break
   fim-=1 ❿
for e in L:
   print(e)
```

Execute o programa e tente entender como ele funciona. Em ❶, utilizamos a variável `fim` para marcar a quantidade de elementos da lista. Precisamos marcar o fim ou a posição do último elemento porque no "*Bubble Sort*" não precisamos verificar o último elemento após uma passagem completa. Em ❷, verificamos se `fim > 1`, pois como comparamos o elemento atual(`L[x]`) com o seguinte(`L[x+1]`), precisamos de, no mínimo, dois elementos. Utilizamos a variável `trocou` em ❸ para indicar que não realizamos nenhuma troca. O valor de `trocou` será utilizado mais tarde para verificar se já temos a lista ordenada ou não. A variável `x` ❹ será utilizada como índice, começando da posição 0. Nossa condição do segundo **`while`** ❺ é especial, pois temos que garantir um próximo elemento para comparar com o atual. Isso faz com que a condição de saída desse **`while`** seja `x<(fim-1)`, ou melhor, que `x` seja anterior ao último elemento. Em ❻ temos a comparação em si, onde `x` é nossa posição atual, e o próximo elemento é o de índice `x+1`. Como estamos ordenando em ordem crescente, desejamos que o próximo elemento sempre seja maior que o atual, dessa forma garantindo a ordenação da lista. Como comparamos apenas dois elementos de cada vez, temos que visitar a lista várias vezes, até que todos os elementos estejam em suas posições. Se a condição de ❻ for verdadeira, esses elementos estão fora de ordem. Nesse caso, devemos inverter ou trocar a posição dos dois elementos. Em ❼ marcamos que efetuamos uma troca e, em ❽, utilizamos uma variável auxiliar ou temporária para trocar os dois valores de posição.

A troca de valores entre duas variáveis é mostrada nas figuras 6.3, 6.4 e 6.5. Como cada variável só guarda um valor de cada vez, utilizaremos uma terceira variável temporária (`temp`) para armazenar o valor de uma delas durante a troca.

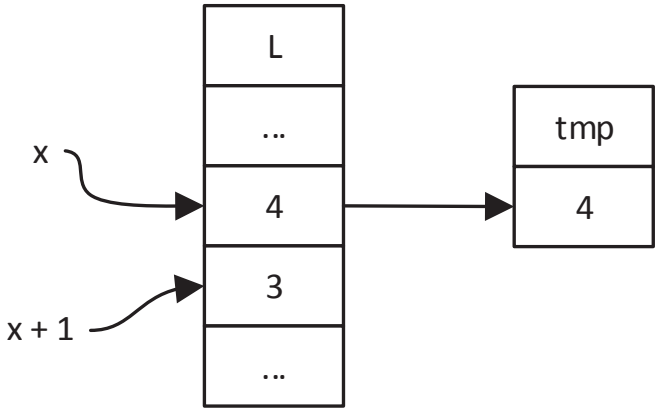

Figura 6.3 – Troca passo a passo. Primeira etapa.

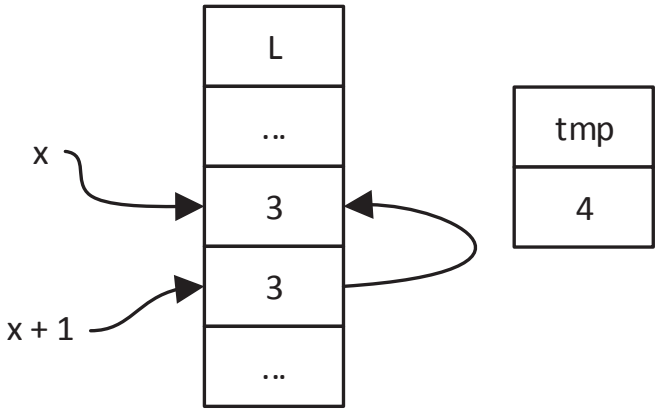

Figura 6.4 – Troca passo a passo. Segunda etapa.

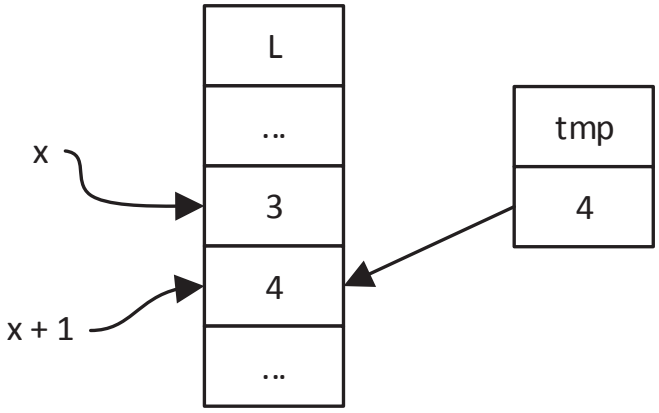

Figura 6.5 – Troca passo a passo. Terceira etapa.

Você pode entender melhor o processo de troca de valores utilizando um problema do dia a dia: imagine que você tenha duas xícaras, uma com leite e outra com café. O problema consiste em passar o leite para a xícara que contém o café, e o café para a xícara que contém o leite, ou seja, você deve trocar o conteúdo das xícaras. Nesse caso, resolveríamos o problema utilizando uma terceira xícara para facilitar a operação. Fizemos o mesmo durante a ordenação, onde chamamos nossa terceira variável de `temp`.

Quando a repetição iniciada em ❺ termina, o maior elemento está posicionado na última posição da lista, ou seja, em sua posição correta. Em ❾, verificamos se algo foi trocado na repetição anterior. Se não trocamos nada de lugar, nossa lista está ordenada, e não precisamos executar a repetição outra vez, por isso o **`break`**. Caso contrário, como a última posição já está com o elemento correto, diminuiremos o valor de `fim` para que não precisemos mais verificá-lo.

Vamos rastrear o algoritmo e observar como tudo isso funciona. Observe o rastreamento completo na tabela 6.1. A coluna linha indica a linha do programa sendo executada: não confunda com os números dentro dos círculos brancos da listagem.

Exercício 6.14 O que acontece quando a lista já está ordenada? Rastreie o programa da listagem 6.44, mas com a lista `L=[1,2,3,4,5]`.

Exercício 6.15 O que acontece quando dois valores são iguais? Rastreie o programa da listagem 6.44, mas com a lista `L=[3,3,1,5,4]`.

Exercício 6.16 Modifique o programa da listagem 6.44 para ordenar a lista em ordem decrescente. `L=[1,2,3,4,5]` deve ser ordenada como `L=[5,4,3,2,1]`.

Tabela 6.1 – Rastreamento da ordenação com Bubble Sort

Linha	L [0]	L[1]	L [2]	L [3]	L[4]	fim	trocou	x	temp
1	7	4	3	12	8				
2						5			
4							False		
5								0	
8							True		
9									7
10	4								
11		7							
12								1	
8							True		
9									7
10		3							
11			7						
12								2	
12								3	

Linha	L [0]	L[1]	L [2]	L [3]	L[4]	fim	trocou	x	temp
8							True		
9									12
10				8					
11					12				
12								4	
15						4			
4							False		
5								0	
8							True		
9									4
10	3								
11		4							
12								1	
12								2	
12								3	
15						3			
4							False		
5								0	
12								1	
12								2	

6.17 Dicionários

Dicionários consistem em uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes. Um dicionário é composto por um conjunto de chaves e valores. O dicionário em si consiste em relacionar uma chave a um valor específico.

Em Python, criamos dicionários utilizando chaves (`{}`). Cada elemento do dicionário é uma combinação de chave e valor. Vejamos um exemplo onde os preços de mercadorias sejam como os da tabela 6.2.

Tabela 6.2 – Preços de mercadorias

Produto	Preço
Alface	R$ 0,45
Batata	R$ 1,20
Tomate	R$ 2,30
Feijão	R$ 1,50

A tabela 6.2 pode ser vista como um dicionário, onde chave seria o produto; e valor, seu preço. Vejamos como criar esse dicionário em Python na listagem 6.45.

▶ Listagem 6.45 – Criação de um dicionário

```
tabela = { "Alface": 0.45,
           "Batata": 1.20,
           "Tomate": 2.30,
           "Feijão": 1.50 }
```

Um dicionário é acessado por suas chaves. Para obter o preço da alface, digite no interpretador, depois de ter criado a tabela, `tabela["Alface"]`, onde `tabela` é o nome da variável do tipo dicionário, e "Alface" é nossa chave. O valor retornado é o mesmo que associamos na tabela, ou seja, 0,45.

Diferentemente de listas, onde o índice é um número, dicionários utilizam suas chaves como índice. Quando atribuímos um valor a uma chave, duas coisas podem ocorrer:

1. Se a chave já existe: o valor associado é alterado para o novo valor.
2. Se a chave não existe: a nova chave será adicionada ao dicionário.

Observe a listagem 6.46. Em ❶, acessamos o valor associado à chave "Tomate". Em ❷, alteramos o valor associado à chave "Tomate" para um novo valor. Observe que o valor anterior foi perdido. Em ❸, criamos uma nova chave, "Cebola", que é adicionada ao dicionário. Veja também como Python imprime o dicionário. Outra diferença entre dicionários e listas é que, ao utilizarmos dicionários, perdemos a noção de ordem. Observe que, durante a manipulação do dicionário, a ordem das chaves foi alterada.

▶ **Listagem 6.46 – Funcionamento do dicionário**

```
>>> tabela = { "Alface": 0.45,
...            "Batata": 1.20,
...            "Tomate": 2.30,
...            "Feijão": 1.50 }
>>> print(tabela["Tomate"]) ❶
2.3
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.3, 'Feijão': 1.5}
>>> tabela["Tomate"] = 2.50 ❷
>>> print(tabela["Tomate"])
2.5
>>> tabela["Cebola"] = 1.20 ❸
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.5, 'Cebola': 1.2, 'Feijão': 1.5}
```

Quanto ao acesso aos dados, temos que verificar se uma chave existe, antes de acessá-la (Listagem 6.47).

▶ **Listagem 6.47 – Acesso a uma chave inexistente**

```
>>> tabela = { "Alface": 0.45,
...            "Batata": 1.20,
...            "Tomate": 2.30,
...            "Feijão": 1.50 }
>>> print(tabela["Manga"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Manga'
```

Se a chave não existir, uma exceção do tipo `KeyError` será ativada. Para verificar se uma chave pertence ao dicionário, podemos usar o operador **`in`** (Listagem 6.48).

▶ **Listagem 6.48 – Verificação da existência de uma chave**

```
>>> tabela = { "Alface": 0.45,
...            "Batata": 1.20,
...            "Tomate": 2.30,
...            "Feijão": 1.50 }
```

```
>>> print("Manga" in tabela)
False
>>> print("Batata" in tabela)
True
```

Podemos também obter uma lista com as chaves do dicionário, ou mesmo uma lista dos valores associados (Listagem 6.49).

▶ Listagem 6.49 – Obtenção de uma lista de chaves e valores

```
>>> tabela = { "Alface": 0.45,
...            "Batata": 1.20,
...            "Tomate": 2.30,
...            "Feijão": 1.50 }
>>> print(tabela.keys())
dict_keys(['Batata', 'Alface', 'Tomate', 'Feijão'])
>>> print(tabela.values())
dict_values([1.2, 0.45, 2.3, 1.5])
```

Observe que os métodos `keys()` e `values()` retornam geradores. Você pode utilizá-los diretamente dentro de um **`for`** ou transformá-los em lista usando a função `list`.

Vejamos um programa que utiliza dicionários para exibir o preço de um produto na listagem 6.50.

▶ Listagem 6.50 – Obtenção do preço com um dicionário

```
tabela = { "Alface": 0.45,
           "Batata": 1.20,
           "Tomate": 2.30,
           "Feijão": 1.50 }
while True:
   produto=input("Digite o nome do produto, fim para terminar:")
   if produto == "fim":
      break
   if produto in tabela: ❶
      print("Preço %5.2f" % tabela[produto]) ❷
   else:
      print("Produto não encontrado!")
```

Em ❶ verificamos se o dicionário contém a chave procurada. Em caso afirmativo, imprimimos o preço associado à chave, ou haverá uma mensagem de erro.

Para apagar uma chave, utilizaremos a instrução **del** (Listagem 6.51).

▶ Listagem 6.51 – Exclusão de uma associação do dicionário

```
>>> tabela = { "Alface": 0.45,
...            "Batata": 1.20,
...            "Tomate": 2.30,
...            "Feijão": 1.50 }
>>> del tabela["Tomate"]
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Feijão': 1.5}
```

Você pode estar-se perguntando quando utilizar listas e quando utilizar dicionários. Tudo depende do que você deseja realizar. Se seus dados são facilmente acessados por suas chaves, quase nunca você precisa acessá-los de uma só vez: um dicionário é mais interessante. Além disso, você pode acessar os valores associados a uma chave rapidamente sem pesquisar. A implementação interna de dicionários também garante uma boa velocidade de acesso quando temos muitas chaves. Porém, um dicionário não organiza suas chaves, ou seja, as primeiras chaves inseridas nem sempre serão as primeiras na lista de chaves. Se seus dados precisam preservar a ordem de inserção (como em filas ou pilhas, continue a usar listas), dicionários não serão uma opção.

6.18 Dicionários com listas

Em Python, podemos ter dicionários nos quais as chaves são associadas a listas ou mesmo a outros dicionários. Imagine uma relação de estoque de mercadorias onde teríamos, além do preço, a quantidade em estoque (Listagem 6.52).

▶ Listagem 6.52 – Dicionário com listas

```
estoque = { "tomate": [ 1000, 2.30],
            "alface": [  500, 0.45],
            "batata": [ 2001, 1.20],
            "feijão": [  100, 1.50] }
```

Nesse caso, o nome do produto é a chave, e a lista consiste nos valores associados, uma lista por chave. O primeiro elemento da lista é a quantidade disponível; e o segundo, o preço do produto.

Uma aplicação seria processarmos uma lista de operações e calcular o preço total de venda, atualizando também a quantidade em estoque.

▶ **Listagem 6.53 – Exemplo de dicionário com estoque e operações de venda**

```
estoque={ "tomate": [ 1000, 2.30],
          "alface": [  500, 0.45],
          "batata": [ 2001, 1.20],
          "feijão": [  100, 1.50] }
venda = [ ["tomate", 5], ["batata", 10], ["alface",5] ]
total = 0
print("Vendas:\n")
for operação in venda:
   produto, quantidade = operação ❶
   preço = estoque[produto][1] ❷
   custo = preço * quantidade
   print("%12s: %3d x %6.2f = %6.2f" %
      (produto, quantidade,preço,custo))
   estoque[produto][0] -= quantidade ❸
   total+=custo
print(" Custo total: %21.2f\n" % total)
print("Estoque:\n")
for chave, dados in estoque.items(): ❹
   print("Descrição: ", chave)
   print("Quantidade: ", dados[0])
   print("Preço: %6.2f\n" % dados[1])
```

Em ❶ utilizamos uma operação de desempacotamento, como já fizemos com `for` e `enumerate`. Como `operação` é uma lista com dois elementos, ao escrevermos `produto, quantidade` temos o primeiro elemento de `operação` atribuído a `produto`; e o segundo, a `quantidade`. Dessa forma, a construção é equivalente a:

```
produto = operação[0]
quantidade = operação[1]
```

Em ❷, utilizamos o conteúdo de `produto` como chave no dicionário `estoque`. Como nossos dados são uma lista, escolhemos o segundo elemento, que armazena o preço do referido produto. Observe que atribuir nomes a cada um desses componentes facilita a leitura do programa.

Imagine escrever:

```
preço = estoque[operação[0]][1]
```

Em ❸, atualizamos a quantidade em estoque subtraindo a quantidade vendida do estoque atual.

Já em ❹ utilizamos o método `items` do objeto dicionário. O método `items` retorna uma tupla contendo a chave e o valor de cada item armazenado no dicionário. Usando um **`for`** com duas variáveis, `chave` e `dados`, efetuamos o desempacotamento desses valores em uma só passagem. Para entender melhor como isso acontece, experimente alterar o programa para exibir o valor de chave e dados a cada iteração.

Exercício 6.17 Altere o programa da listagem 6.53 de forma a solicitar ao usuário o produto e a quantidade vendida. Verifique se o nome do produto digitado existe no dicionário, e só então efetue a baixa em estoque.

Exercício 6.18 Escreva um programa que gere um dicionário, onde cada chave seja um caractere, e seu valor seja o número desse caractere encontrado em uma frase lida.

Exemplo: O rato -> { “O”:1, “r”:1, “a”:1, “t”:1, “o”:1}

6.19 Tuplas

Tuplas podem ser vistas como listas em Python, com a grande diferença de serem imutáveis. Tuplas são ideais para representar listas de valores constantes e também para realizar operações de empacotamento e desempacotamento de valores. Primeiramente, vejamos como criar uma tupla.

Tuplas são criadas de forma semelhante às listas, mas utilizamos parênteses em vez de colchetes. Por exemplo:

```
>>> tupla = ("a", "b", "c")
>>> tupla
('a', 'b', 'c')
```

Tuplas suportam a maior parte das operações de lista, como fatiamento e indexação.

```
>>> tupla[0]
'a'
>>> tupla[2]
```

```
'c'
>>> tupla[1:]
('b', 'c')
>>> tupla * 2
('a', 'b', 'c', 'a', 'b', 'c')
>>> len(tupla)
3
```

Mas tuplas não podem ter seus elementos alterados. Veja o que acontece se tentarmos alterar uma tupla:

```
>>> tupla[0]="A"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Várias funções utilizam ou geram tuplas em Python. Tuplas podem ser utilizadas com **for**:

```
>>> for elemento in tupla:
...     print(elemento)
...
a
b
c
```

Python também permite criar tuplas usando valores separados por vírgula, independente de usarmos parênteses:

```
>>> tupla = 100, 200, 300
>>> tupla
(100, 200, 300)
```

No caso, 100, 200 e 300 foram convertidos em uma tupla com três elementos. Esse tipo de operação é chamado de empacotamento.

Tuplas também podem ser utilizadas para desempacotar valores, por exemplo:

```
>>> a, b = 10, 20
>>> a
10
>>> b
20
```

Onde o primeiro valor, 10, foi atribuído à primeira variável `a` e 20 a segunda, `b`. Esse tipo de construção é interessante para distribuirmos o valor de uma tupla em várias variáveis.

Também podemos trocar rapidamente os valores de variáveis com construções do tipo:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Onde a tupla da esquerda foi usada para atribuir os valores à direita. Nesse caso, as atribuições: `a=b` e `b = a` foram realizadas imediatamente, sem precisar utilizarmos uma variável intermediária para a troca.

A sintaxe do Python é um tanto especial quando precisamos criar tuplas com apenas um elemento. Como os valores são escritos entre parênteses, quando apenas um valor estiver presente, devemos acrescentar uma vírgula para indicar que o valor é uma tupla com apenas um elemento. Veja o que acontece, usando e não usando a vírgula:

```
>>> t1 = (1)
>>> t1
1
>>> t2 = (1,)
>>> t2
(1,)
>>> t3 = 1,
>>> t3
(1,)
```

Veja que em `t1` não utilizamos a vírgula, e o código foi interpretado como um número inteiro entre parênteses. Já em `t2`, utilizamos a vírgula, e nossa tupla foi corretamente construída. Em `t3`, criamos outra tupla, mas nesse caso nem precisamos usar parênteses.

Podemos também criar tuplas vazias, escrevendo apenas os parênteses:

```
>>> t4=()
>>> t4
()
>>> len(t4)
0
```

Tuplas também podem ser criadas a partir de listas, utilizando-se a função `tuple`:

```
>>> L=[1,2,3]
>>> T=tuple(L)
>>> T
(1, 2, 3)
```

Embora não possamos alterar uma tupla depois de sua criação, podemos concatená-las, gerando novas tuplas:

```
>>> t1=(1,2,3)
>>> t2=(4,5,6)
>>> t1+t2
(1, 2, 3, 4, 5, 6)
```

Observe que se uma tupla contiver uma lista ou outro objeto que pode ser alterado, este continuará a funcionar normalmente. Veja o exemplo de uma tupla que contém uma lista:

```
>>> tupla=("a", ["b", "c", "d"])
>>> tupla
('a', ['b', 'c', 'd'])
>>> len(tupla)
2
>>> tupla[1]
['b', 'c', 'd']
>>> tupla[1].append("e")
>>> tupla
('a', ['b', 'c', 'd', 'e'])
```

Neste caso, nada mudou na tupla em si, mas na lista que é seu segundo elemento. Ou seja, a tupla não foi alterada, mas a lista que ela continha, sim.