

Python - Guia do Programador

Peter Jandl Junior

2021-07-14

Contents

Prefácio	5
Convenções	5
Código fonte	6
1 Introdução	7
1.1 Breve Histórico	7
1.2 Características	8
1.3 Habilidades da Programação	9
2 Computação	11
2.1 Valores Literais	11
2.2 Tipos de Dados	12
2.3 Variáveis	14
2.4 Operadores	18
2.5 Expressões	27
2.6 Prioridade dos operadores	28
2.7 Atribuição	30
2.8 Funções	32
3 Entrada e Saída	37
3.1 Saída Simples	38
3.2 Entrada	43
3.3 Saída Formatada	46

4	Sequenciação	55
4.1	Programa Mínimo	56
4.2	Estratégia E-P-S	56
5	Repetição	69
5.1	Repetição Condicional	70
5.2	Repetição Automática	81
6	Decisão	89
6.1	Decisão Simples	90
6.2	Decisão Completa	93
6.3	Decisões Aninhadas	97
6.4	Decisões Encadeadas	97
6.5	Desvio Incondicional	101
7	Funções	109
7.1	Tipos de funções	110
7.2	Retorno de Valor	110
7.3	Passagem de Parâmetros	110
7.4	Parâmetros <i>Default</i>	110
7.5	Parâmetros Variáveis	110
8	Módulos	111
8.1	Criação de módulos	111
8.2	Importação geral	111
8.3	Programa Principal	112
8.4	Importação seletiva	112
8.5	Pacotes	112
A	Instalação básica	113
B	Instalação do Anaconda	115

Prefácio

Este é um livro experimental, organizado com o propósito de incentivar o uso da linguagem de programação Python e, sendo assim, procura cobrir os aspectos mais básicos de sua utilização, bem como alguns conceitos fundamentais e boas práticas da programação. Para tanto, apresenta os elementos básicos da linguagem, ao mesmo tempo que trata dos aspectos básicos da programação, prosseguindo para conceitos e técnicas mais avançados.

Será utilizada a versão 3 do Python, que é substancialmente diferente da versão 2 e, também, considerada mais correta semanticamente falando, além de suportar um conjunto de novas e interessantes características.

Este material foi escrito com a ferramenta *RStudio*, *Markdown*, *R Markdown* e o pacote *bookdown*¹ (XIE, 2021, 2015), portanto emprega o suporte do *Pandoc*, como, por exemplo, para indicar expressões matemáticas, tal como $f(x) = a * x^2 + b * x + c$.

Convenções

Para dar destaque a termos técnicos, nomes, marcas ou palavras em língua estrangeira, será usada a formatação em *itálico*. O **negrito** será reservado para elementos realmente importantes. Além disso, valores, variáveis, funções e classes figurarão em texto monoespaçado, como 2021, `x`, `print()` ou `complex`.

O muitos fragmentos de código e exemplos contidos neste material aparecem como segue. Quando incluída, a saída produzida pelos fragmentos aparecerá intercalada ou imediatamente a seguir do código apresentado, sempre precedida pelos caracteres `>>>`.

```
# Uma mensagem de boas vindas
print('Bem vindo ao "Python - Guia do Programador"!')
```

```
>>> Bem vindo ao "Python - Guia do Programador"!
```

¹Bookdown no GitHub: <https://github.com/rstudio/bookdown>.

Além dos fragmentos de código e exemplos completos, dicas e avisos aparecerão em destaque no texto, como segue.



Para fazer recomendações e indicar boas práticas de programação.



Para chamar a atenção para detalhes e outras situações pouco usuais.



Para destacar erros comuns e problemas de difícil solução.



Para complementar o texto e fornecer informações adicionais.

Código fonte

Este material será futuramente armazenado no **GitHub**, no repositório: <https://github.com/pjandl/pygp>.

Chapter 1

Introdução

Python é uma linguagem de programação bastante popular, moderna, utilizada tanto no ambiente corporativo, como no meio acadêmico, e que todo programador devia conhecer. Segundo Kopec (KOPEC, 2019, pág.1):

“Python é usada em atividades tão diversas como ciência de dados, produção de filmes, educação em ciência da computação, gerenciamento de tecnologia da informação e muito mais. Realmente não há um ramo da computação que Python não tenha tocado (exceto, talvez, desenvolvimento de *kernel* de sistemas operacionais). Python é amada por sua flexibilidade, sintaxe bela e sucinta, orientação a objetos pura, e uma comunidade movimentada.”

Então, vamos falar um pouco sobre ela!

1.1 Breve Histórico

Observando as dificuldades de muitas pessoas em aprender a programar e trabalhar com programação, Guido Van Rossum, que então trabalhava no *Stichting Mathematisch Centrum* (CWI, Holanda), deu início ao desenvolvimento de uma nova linguagem de programação em 1990. Sua proposta era oferecer um linguagem de *script* simples, fácil de aprender e de programar. O nome **Python** foi inspirado pelo grupo humorístico britânico *Monty Python*, criador do programa de televisão *Monty Python's Flying Circus*.

Em 1995, já no *Corporation for National Research Initiatives* (EUA), Rossum, deu continuidade ao seu projeto de tornar o Python ainda melhor, neste ponto com as opiniões e ajuda de várias outras pessoas contribuíram no desenvolvimento da linguagem e de suas bibliotecas.

Para garantir a evolução contínua da linguagem e, ao mesmo tempo, desvincular o Python da pessoa de Guido Van Rossum, foi criada em 2001 a *Python Software Foundation* (PSF), uma organização sem fins lucrativos, que detém os direitos de propriedade intelectual do Python, destinada a manter, desenvolver e divulgar a linguagem com base em um modelo de desenvolvimento comunitário, aberto, com participação de membros individuais e corporativos.

Todas as versões do Python são de *código aberto*¹.

1.2 Características

Python é uma linguagem de programação de alto nível, orientada a objetos, interpretada, interativa, de semântica dinâmica, com tipagem forte. Sua sintaxe é bastante compacta e direta, o que possibilita aos seus programas serem mais curtos do que os construídos em C, C++, Java e outras linguagens.

As razões para isso são:

- dispõe de tipos de dados de alto nível que permitem a expressão direta de operações complexas;
- não requer a declaração prévia de variáveis ou argumentos, cujos tipos são inferidos durante a execução do programa;
- a criação de blocos de diretivas nas suas construções requer apenas a indentação simples, sem necessidade de elementos extras, como chaves ou palavras reservadas, para identificação de início e fim de tais blocos.

Foi projetada para ser simples e elegante, ao mesmo que é sofisticada e completa, pois:

- é orientada a objetos, oferece herança simples, herança múltipla e tratamento polimórfico de objetos;
- oferece exceções como mecanismo mais moderno para o tratamento de erros;
- possui coleta automática de lixo, que efetua a reciclagem de memória de objetos descartados, simplificando o desenvolvimento;
- inclui recursos avançados de manipulação de texto, listas e outras estruturas de dados; e
- seus programas podem ser organizados em módulos e pacotes, reusáveis em diferentes programas.

Pode ser usada em múltiplas plataformas (Mac OS, Microsoft Windows, distribuições Linux e Unix), o que evidencia sua portabilidade². Além de ser

¹Veja uma definição de *código aberto* ou *open source* em <http://www.opensource.org/>.

²*Portabilidade* é a capacidade dos programas gerados por uma linguagem de programação de serem executados em diferentes plataformas operacionais, idealmente sem alterações, ou com um mínimo de modificações.

extensível e dotada de uma ampla e versátil biblioteca.

Como característica de seu projeto, a linguagem Python possui um interpretador que pode funcionar em modo interativo. Isto possibilita que programas escritos em Python, ou mesmo trechos de código, possam ser testados antes de serem compilados ou inclusos em outros programas. Com isso, o Python encoraja a programação de maneira simples, sem requisitar código burocrático, o que a torna muito conveniente para criação rápida de programas.

Por tudo isso é muito utilizada para tratamento, processamento de visualização de dados em aplicações de Análise de Dados (*Data Analysis*) e de Ciência de Dados (*Data Science*). Mas também é empregada na construção de aplicações complexas e de grande porte em empresas icônicas como DropBox, Google, IBM, Instagram, Nasa, Reddit, Spotify, Uber, YouTube e outras.

1.3 Habilidades da Programação

*Programar*³ significa *fazer o programa de, planejar, incluir em programação*, além de ter como sinônimos *planejar, projetar, delinear, designar e coordenar*.

A programação de computadores exige o domínio de seis habilidades distintas:

1. computação (Capítulo 2),
2. entrada e saída (Capítulo 3),
3. repetição (Capítulo 5),
4. decisão (Capítulo 6),
5. modularização (Capítulo 8), e
6. sequenciação (Capítulo 4).

A *computação* é habilidade necessária para expressar e realizar cálculos, ou seja, a capacidade de combinar valores, variáveis, operadores e funções para obter os resultados desejados. Esta habilidade explora as capacidades dos computadores em realizar cálculos.

Os dados necessários para realizar os cálculos desejados, isto é, requeridos pela computação, precisam ser obtidos dos usuários ou outras fontes de dados. Da mesma maneira, os resultados produzidos por um programa precisam ser exibidos para seus usuários. Estas são as operações de *entrada e saída*, simples mas essenciais.

A *repetição* consiste da identificação de uma instrução ou de um conjunto de instruções que deve ser executado mais de uma vez. Isto inclui determinar o número de vezes que tais instruções serão realizadas ou a condição que exige sua repetição. Além disso, a repetição permite reduzir a quantidade de instruções

³Dicio (Dicionário On-Line de Português): <https://www.dicio.com.br/programar/>.

que devem ser escritas para resolver um problema, o que confere maior flexibilidade às soluções criadas.

Outra habilidade importante é a *decisão*, pois possibilita escolher quais instruções serão executadas, isto é, permite que, durante a execução das instruções, sejam selecionadas aquelas que serão executadas. Esta decisão é realizada mediante a avaliação de uma condição que o programador estabelece como critério de escolha ou de seleção.

A *modularização*, que é a habilidade que trata da divisão das instruções necessárias para resolução de um problema em partes menores, cada uma com responsabilidades distintas. Essa estratégia oferece várias vantagens, entre elas, simplificar a compreensão e solução do problema, além de possibilitar o reuso.

Finalmente temos a *sequenciação*, que se refere a habilidade de organizar as instruções de um programa de maneira tal que seja resolvido um problema específico. Esta habilidade trata de estabelecer uma sequência adequada de instruções para solução de um problema, ou seja, ela representa a *lógica* com a qual resolvemos um problema. A sequenciação determina como os dados de um problema serão processados, portanto combina todas as habilidades anteriores: a entrada e saída, a computação, a repetição, a decisão e a modularização necessárias para a construção de um programa.

Chapter 2

Computação

A palavra *computar*¹ significa *fazer o cômputo de, calcular, orçar*, assim, a *computação* é a habilidade da programação voltada para a realização de cálculos, o que permite explorar uma das capacidades centrais dos computadores.

A realização de cálculos envolve a construção de *expressões*, as quais podem conter vários elementos:

- valores literais,
- variáveis,
- operadores, e
- funções.

2.1 Valores Literais

Um *valor literal*, ou apenas *literal*, é uma quantidade, um número, uma palavra, um nome ou um texto que podemos ler diretamente numa instrução, isto é, um elemento que não depende da execução da instrução ou de qualquer outra parte do programa para que possa ser compreendido. Alguns exemplos podem facilitar no entendimento do que são os literais.

O número 2021 é um literal, assim como 15, 3.14, -3 ou 4294967296 também são literais numéricos, que são escritos diretamente no texto do programa Python, tal como em todas as linguagens de programação. Vale notar que o *separador* decimal é o caractere ponto (.).

Já a inclusão de literais de texto no Python, seja uma palavra, frase ou um caractere individual, requer que sejam dispostos entre aspas simples (') ou aspas

¹Dicio (Dicionário On-Line de Português): <https://www.dicio.com.br/computar>.

duplas ("). Estes *delimitadores* de texto podem ser usados para indicar palavras, como 'computador' ou "programação", e caracteres individuais, como 'A', "x", '!', "@" ou '+'. Pequenas frases, que não podem exceder uma linha, são indicadas da mesma maneira, ou seja, com uso destes delimitadores, como no fragmento que segue.

```
'Python é uma linguagem de programação moderna.'  
"A área de 'data science' utiliza Python."  
'A linguagem Python é "interpretada" e "dinâmica".'
```

Os delimitadores são exigidos para que seja possível distinguir o texto literal fornecidos pelo programador dos demais elementos da linguagem. O delimitador não pode fazer parte do texto delimitado, no entanto, é interessante observar que as aspas simples podem usadas quando delimitadas por aspas duplas e vice-versa.

Também é possível definir texto literal com múltiplas linhas, com o uso triplo de aspas simples ou aspas duplas, o que pode ser conveniente em algumas situações, como por exemplo indicar comandos SQL² usados pelo programa. Seguem exemplos diretos.

```
"""Python is used in pursuits as diverse as data science,  
film-making, computer science education, IT management,  
and much more."""  
  
'''There really is no computing field that Python has not  
touched (except maybe kernel development). Python is loved  
for its flexibility, beautiful and succinct syntax,  
object-oriented purity, and bustling community.  
--- Kopec (2019)'''
```

Além de literais numéricos e de texto, o Python também dispõe de dois literais de tipo lógico (ou booleanos), que são **False** e **True**, que respectivamente representam os estados *falso* e *verdadeiro*.

2.2 Tipos de Dados

A linguagem Python é capaz de lidar com vários tipos de dados, ou seja, com categorias distintas de valores, cada uma oferecendo um conjunto próprio de

² *Structured Query Language*, linguagem padronizada para consulta e manipulação de dados em *Sistemas Gerenciadores de Bancos de Dados Relacionais* (SGBDR).

possibilidades. Os tipos de dados básicos disponíveis na linguagem são considerados tipos *built-in*³, ou *nativos*, e estão listados na Tabela 2.1.

Tabela 2.1: Tipos de dados *built-in*

Tipo	Descrição
<code>int</code>	Inteiro (ou integral), valor numérico sem parte fracionária.
<code>float</code>	Real, valor numérico com parte fracionária em ponto flutuante.
<code>bool</code>	Lógico (ou booleano).
<code>string</code>	<i>String</i> ou cadeia de caracteres.
<code>complex</code>	Número complexo, com a parte imaginária identificada pelo sufixo <code>j</code> .

O Python dispõe de três tipos de dados numéricos *built-in*: `int`, `float` e `complex`. O tipo `int` possibilita a representação de valores numéricos inteiros, ou seja, números, contagens e quantidades, positivos ou negativos, mas sem uma parte fracionária. A função *built-in* `type()` permite determinar o tipo de quaisquer valores literais (na verdade, de qualquer coisa no Python). Observe o uso de `type()` para o valor inteiro `15`.

```
type(15)
```

```
>>> <class 'int'>
```

Qualquer valor inteiro, quando avaliado por `type()`, produz como retorno a classe `int`, que representa este tipo de dados.

Analogamente, o tipo `float` possibilita a representação de valores numéricos reais, ou seja, números positivos ou negativos dotados de uma parte fracionária. Como antes, pode ser usada a função *built-in* `type()` para determinar o tipo de literais reais, como segue.

```
type(3.14)
```

```
>>> <class 'float'>
```

Valores reais avaliados por `type()` produzem como retorno a classe `float`, que representa este tipo de dados.

Diferente da grande maioria das linguagens de programação, Python permite a representação nativa de números complexos, ou seja, valores dotados de uma

³O termo *built-in* é utilizado para designar um elemento que faz parte da própria definição da linguagem, ou seja, está disponível em todos os programas, sem necessidade de importação de módulos ou pacotes.

parte real e uma parte imaginária, que utiliza o sufixo `j` para diferenciá-la da parte real. No fragmento que segue, a função `type()` é utilizada para determinar o tipo do valor literal `1.5 - 4.9j`, um número complexo cuja parte real tem valor 1.5 e a parte imaginária vale 4.9j.

```
type(1.5 - 4.9j)
```

```
>>> <class 'complex'>
```

A verificação de tipo com `type()` retorna a classe `complex` quando recebe números complexos como argumento.

Outro importante tipo *built-in* é `bool` que representa o tipo lógico ou booleano, que possui apenas dois valores possíveis: `False`, para valores falsos; e `True`, para valores verdadeiros. O uso de `type()` para o literal lógico `True` é mostrado no fragmento que segue.

```
type(True)
```

```
>>> <class 'bool'>
```

Como esperado, é retornada a classe `bool`.

Finalmente, a representação de texto, sejam caracteres, palavras ou frases, é feita por *strings*, ou seja, elementos do tipo `str`, à despeito do delimitador usado (aspas simples, duplas ou triplas). A função `type()` também permite determinar o tipo de literais ou valores de texto, como segue.

```
type('Python: Guia do programador')
```

```
>>> <class 'str'>
```

Observamos que a classe `str` é retornada quando `type()` verifica o tipo de uma *string*, ou seja, quando se fornece um argumento de texto.

2.3 Variáveis

Um programa de computador requer o uso de alguns ou de muitos dados para que possa produzir os resultados desejados. Durante a execução do programa, os dados necessários são armazenados na memória do computador. Para evitar que o programador tenha que lidar com os endereços de memória, isto é, com as posições onde os dados ficam efetivamente armazenados, são utilizadas *variáveis*.

Uma *variável* é um espaço em memória, reservado para guardar um valor, ao qual se associa um *identificador*, ou seja, um nome por meio do qual se define e se recupera o valor armazenado. O uso de variáveis simplifica a programação, pois o programador não precisa se preocupar com os endereços de memória utilizados, nem com o espaço necessário (número de bytes) para armazenar tais valores, tão pouco com a organização dos dados e das instruções do programa.

Por meio do uso das variáveis, o programador pode armazenar valores literais ou o resultado de cálculos diversos, que podem ser utilizados em etapas posteriores do programa, evitando sua repetição e o processamento destes cálculos.

Além disso, o uso de variáveis constitui um importante mecanismo de *abstração*⁴, pois o uso de nomes significativos melhora a legibilidade do programa e permite que suas ações sejam compreendidas mais facilmente.

A criação de variáveis em Python é bastante simples e direta, empregando a sintaxe que segue:

```
identificador = valor_inicial
```

O *identificador* é o *nome* que o programador escolhe para uma variável, o símbolo `=` é o operador de atribuição e `valor_inicial` é o valor que será inicialmente armazenado por esta variável. Por exemplo, a criação da variável de nome `x` com valor inicial definido pelo literal `15`:

```
# Variável x com conteúdo 15  
x = 15
```

Esta construção é lida como *variável `x` recebe o valor 15* ou, resumidamente, *`x` recebe 15*.

A linha iniciada pelo caractere `#` é um *comentário*, ou seja, um texto, com qualquer conteúdo desejado pelo programador, que serve para registrar alguma explicação ou observação sobre o código. Os comentários não são considerados na execução dos programas e, portanto, sua utilização não interfere em seu funcionamento, nem em seu desempenho.

⁴Segundo o Dicio (Dicionário On-Line de Português) *abstrair* é a ação de analisar isoladamente um aspecto, contido num todo, sem ter em consideração sua relação com a realidade. Fazer a abstração de uma coisa permite simplificar, pois observamos seu aspecto principal, sem levar em conta seus detalhes (<https://www.dicio.com.br/abstracao>).



O uso de comentários é considerado uma boa prática de programação e, assim, é incentivado. Sempre que possível, o programador deve considerar acrescentar comentários aos seus programas, clarificando as decisões sobre sua construção.

Desta forma, para criar uma nova variável em um programa Python basta atribuir um valor para um novo identificador. A criação de uma variável desta maneira é chamada *inicialização*. A partir de sua inicialização, a variável criada se torna disponível no escopo onde foi declarada.

Para utilizar uma variável, em Python e outras linguagens de programação, basta utilizar seu nome, de maneira que este é automaticamente substituído pelo valor atual (ou corrente) da variável. Ou seja, apenas escrever seu nome.

```
x
```

```
>>> 15
```

Como esperado, o uso do nome de variável `x` recupera seu valor, no caso `15`.

Observe que o Python não requer que o tipo da variável seja declarado, pois este é inferido conforme o tipo do valor atribuído, assim a variável `x` será do tipo `int`, como mostrado pelo uso da função *built-in* `type()`:

```
type(x) # obtenção do tipo de x
```

```
>>> <class 'int'>
```

Neste fragmento também se observa a existência de um comentário. Aqui, o texto existente *antes* do caractere `#` é considerado como código e, depois dele, até o final da linha, como comentário.

Cada vez que a variável recebe um valor, o tipo da variável é novamente inferido, de maneira que, se atribuído um valor de tipo diferente do previamente armazenado na variável, seu tipo é *alterado dinamicamente*, sem produzir qualquer tipo de erro. Assim, a variável `x`, do tipo `int`, pode receber um valor real como segue:

```
x = 7.45 # atribuição de valor para x
```

A alteração do tipo da variável pode ser visto por meio da função `type()`:


```
type(x)
```

```
>>> <class 'float'>
```

A valor da variável `a` pode ser recuperado com uso de seu nome, permitindo verificar a alteração em seu conteúdo.

```
x
```

```
>>> 7.45
```

Em conjunto, tudo isto confere grande simplificação e flexibilidade ao Python em relação a criação e utilização de variáveis.

2.3.1 Denominação de Variáveis

Os nomes de variáveis em Python podem ser compostos de uma ou mais letras, números e também símbolos `_` (sublinhado ou *underscore*), desde que iniciados por uma letra ou sublinhado. É recomendado que usem apenas letras minúsculas e, caso sejam compostos de mais de uma palavra, estas sejam separadas por um sublinhado. Esta convenção é conhecida como *snake case*.

São exemplo válidos: `x`, `s3`, `total`, `quadra03`, `posicao_absoluta`, `_media_parcial`.

Desde que seguida esta regra de formação, os nomes podem quaisquer, exceto das *palavras reservadas* da linguagem (seção 2.3.2), e arbitrariamente longos, assim sugere-se o uso de denominações representativas do propósito das variáveis, melhorando a legibilidade dos programas. Caracteres acentuados podem ser usados, embora desaconselhado. Em hipótese alguma os nomes podem conter espaços em branco, tabulações ou quaisquer operadores.

2.3.2 Palavras Reservadas

O Python possui um conjunto de *palavras reservadas* que tem significado pré-definido, pois indicam as diretivas da linguagem e outros elementos de sua sintaxe. As *palavras reservadas*, ou as *keywords*, listadas na Tabela 2.2 não podem ser utilizadas como identificadores ou para qualquer outro fim, exceto o determinado pela linguagem.

Tabela 2.2: Palavras reservadas (*keywords*)

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

A maioria das palavras reservadas do Python é comum à outras linguagens de programação. Por exemplo, dentre as 35 *keywords*, 12 são comuns ao Java e ao C#.

2.4 Operadores

A utilidade dos computadores se deve, em grande parte, às suas capacidades de realizar cálculos. Então, as linguagens de programação devem suportar essas capacidades e, para isso, deve oferecer operadores que permitam combinar valores e variáveis (seção 2.3) para expressar as sequências de cálculos adequadas à obtenção dos resultados desejados.

Como na matemática, um operador é um símbolo convencionado para representar uma operação específica entre seus operandos, isto é, os valores participantes desta operação. Existem cinco grupos principais de operadores, indicados na Tabela 2.3.

Tabela 2.3: Grupos de operadores

Grupo	Descrição
Aritméticos	Destinados às operações algébricas comuns, como adição, subtração e outras.
Relacionais	Possibilitam a comparação entre valores numéricos e não numéricos.
Lógicos	Permitem a combinação de predicados lógicos.
Atribuição	São usados para definir o valor de variáveis e parâmetros de funções.
Bit-a-bit	Permitem a manipulação dos bits de valores inteiros ⁵ .

Com o uso destes operadores, é possível realizar cálculos, comparar valores, avaliar condições e atribuir valores para variáveis, como será tratados nas seções que seguem.

2.4.1 Operadores Aritméticos

Os operadores aritméticos são destinados à realização das operações algébricas de adição, subtração, multiplicação, divisão e potenciação, como relacionado na Tabela 2.4, onde podemos observar que a maior parte dos operadores aritméticos são idênticos aos usados na matemática, exatamente para facilitar sua identificação e emprego.

Tabela 2.4: Operadores aritméticos

Operador	Operação	Aridade ⁶	Associatividade ⁷
+	Adição (soma).	2	Esquerda
-	Subtração (diferença).	2	Esquerda
*	Multiplicação (produto).	2	Esquerda
/	Divisão (quociente).	2	Esquerda
//	Divisão inteira (quociente).	2	Esquerda
%	Resto da divisão inteira (módulo).	2	Esquerda
**	Potenciação (exponenciação).	2	Esquerda
+	Sinal positivo.	1	Direita
-	Sinal negativo.	1	Direita

Nas próximas seções, as variáveis A, B e C serão empregadas para armazenar valores que serão utilizados em expressões simples. Alguns dos resultados obtidos serão atribuídos à outras variáveis, como R, S e T.

2.4.1.1 Adição

Utilizamos o operador + para indicar a adição ou a soma, que requer dois operandos (sua aridade), ou seja, os dois valores que serão adicionados. No fragmento que segue é possível ver que o uso deste operador é simples.

```
A = 123 # Valores arbitrários, podem ser outros
A + 456
```

```
>>> 579
```

O operador + pode ser usado para somar qualquer combinação de valores in-

⁶Na matemática a *aridade* de uma função ou operação é o número de argumentos ou operandos tomados.

⁷Na matemática a *associatividade* de um operador determina qual de seus operandos é avaliado e tomado primeiro.

teiros e reais, além de obedecer as propriedades *comutativa*⁸, *associativa*⁹, *distributiva*¹⁰ e do *elemento nêutro*¹¹ da adição. A soma de valores inteiros produz resultados de tipo `int`, mas se combinados valores inteiros e reais, o resultado será de tipo `float`, como segue.

```
B = 0.456 # Valores arbitrários, podem ser outros
A + B
```

```
>>> 123.456
```

Existe outro uso para o operador `+`, que é como sinal positivo, tal como `+5` ou `+19.12`, mas cujo uso é pouco frequente, pois por padrão, valores sem sinal são considerados positivos.

2.4.1.2 Subtração

O operador `-` permite realizar a subtração ou a diferença entre dois valores, ou seja, requer dois operandos (sua aridade). Seu uso também é simples.

```
A = 123 # Valores arbitrários, podem ser outros
A - 100
```

```
>>> 23
```

O operador `-` pode efetuar a diferença de qualquer combinação de valores inteiros e reais, além de obedecer as propriedades *distributiva*¹² e do *elemento nêutro*¹³ da subtração. A subtração de valores inteiros produz resultados de tipo `int`, mas se combinados valores inteiros e reais, o resultado será de tipo `float`.

```
B = 0.456 # Valores arbitrários, podem ser outros
A - B
```

⁸Propriedade *comutativa*: a ordem dos operandos não altera o resultado, pois na adição temos que

$$A + B = B + A.$$

⁹Propriedade *associativa*: a associação dos operandos não modifica o resultado, pois na adição temos que

$$A + B + C = (A + B) + C = A + (B + C) = (A + C) + B.$$

¹⁰Propriedade *distributiva*: realizamos o produto do termo externo ao parênteses com seus termos internos, ou seja, na adição $A * (B + C) = A * B + A * C$.

¹¹*Elemento nêutro*: valor que não modifica o resultado da operação, na adição ao somar zero não altera o resultado, pois $A + 0 = A$.

¹²Propriedade *distributiva*: realizamos o produto do termo externo ao parênteses com seus termos internos, ou seja, na subtração $A * (B - C) = A * B - A * C$.

¹³*Elemento nêutro*: valor que não modifica o resultado da operação, subtrair zero não altera o resultado, pois $A - 0 = A$.

```
>>> 122.544
```

Como para o operador `+`, existe um segundo uso para o operador `-` como sinal negativo, por exemplo, `-7` ou `+20.06`, e cujo uso é mais comum, para explicitar valores considerados negativos.

2.4.1.3 Multiplicação

O operador `*` permite efetuar a multiplicação ou o produto de dois valores, tomando dois operandos, com uso como segue.

```
A = 537 # Valores arbitrários, podem ser outros
B = 215
A * B
```

```
>>> 115455
```

Este operador pode efetuar o produto de qualquer combinação de valores inteiros e reais, além de obedecer as propriedades *comutativa*¹⁴, *associativa*¹⁵ e do *elemento nêutro*¹⁶ da multiplicação. Como antes, o produto de valores inteiros produz resultados de tipo `int`, mas se multiplicados valores inteiros e reais, o resultado será de tipo `float`.

2.4.1.4 Divisão real, divisão inteira e resto da divisão

O operador `/` realiza a divisão de dois valores, obtendo um quociente a partir de dois operandos, com uso como segue.

```
A = 537 # Valores arbitrários, podem ser outros
B = 215
A / B
```

```
>>> 2.4976744186046513
```

Podem ser combinados valores inteiros e reais com este operador, que também possui um *elemento nêutro*¹⁷. Também deve ser destacado que este operador

¹⁴Propriedade *comutativa*: a ordem dos operandos não altera o resultado, pois na multiplicação $A * B = B * A$.

¹⁵Propriedade *associativa*: a associação dos operandos não modifica o resultado, pois na multiplicação $A * B * C = (A * B) * C = A * (B * C) = (A * C) * B$.

¹⁶*Elemento nêutro*: valor que não modifica o resultado da operação, a multiplicação por um não modifica o resultado, pois $A * 1 = A$.

¹⁷*Elemento nêutro*: valor que não modifica o resultado da operação, a divisão por um não modifica o resultado, pois $A / 1 = A$.

realiza a divisão real dos operandos indicados, produzindo um resultado de tipo `float`, ou seja, que pode conter uma parte fracionária, com uma ou mais casas decimais. Mesmo que o resultado da divisão seja exato e não possua uma parte fracionária, seu tipo será `float`.



Deve-se tomar cuidado com a divisão por zero, que provoca o erro `ZeroDivisionError`.

Se desejado, pode ser utilizado o operador `//`, que realiza a divisão inteira (*floor division*) de seus operandos, descartando a parte fracionária, retornando um resultado sempre do tipo `int`.

```
A = 537 # Valores arbitrários, podem ser outros
B = 215
A // B
```

```
>>> 2
```

Também é possível obter o resto da divisão inteira com o operador `%`, ou seja, a parcela inteira descartada pela divisão inteira. Por exemplo a divisão `6 / 4` produz `1.5`, um valor real; enquanto a divisão inteira `6 // 4` resulta `1`, sendo que o resto desta divisão `6 % 4` permite obter `2`. Este operador também é conhecido como *módulo*.

```
A = 17 # Valores arbitrários, podem ser outros
B = 3
A % B
```

```
>>> 2
```

2.4.1.5 Potenciação

Python oferece um operador para realização da *potenciação* (ou da *exponenciação*) que é `**` (duplo asterisco, sem espaço em branco), usado na forma `base ** expoente`, onde tanto a base, como o expoente, podem ser números inteiros ou reais, como segue:

```
A = 17 # Valores arbitrários, podem ser outros
B = 3
A ** B
```

```
>>> 2
```

Assim, `2 ** 10` representa dois elevado à décima potência e `10 ** 3` calcula dez elevado ao cubo.

Como na matemática, expoentes negativos representam potências inversas, por exemplo `2 ** -3` equivale à `1 / (2 ** 3)`; e expoentes entre 0 e 1 permitem efetuar a *radiciação* (obter raízes), ou seja, `16 ** (1/2)` e `16 ** 0.5` permitem calcular a raiz quadrada de 16, enquanto `5 ** (1/3)` e `5 ** 0.3333` calculam a raiz cúbica de 5, tal como no fragmento seguinte.

```
A = 5 # Valores arbitrários, podem ser outros
B = 0.3333
A ** B
```

```
>>> 1.7098842124667966
```

2.4.2 Operadores Relacionais

Os operadores relacionais permitem comparar valores determinando as existência de relações específicas entre eles, tal como mostra a Tabela 2.4. Vários dos operadores relacionais são compostos por dois caracteres, entre os quais *não pode existir espaços em branco*.

Tabela 2.4: Operadores relacionais

Operador	Relação	Aridade
<code>></code>	Maior que.	2
<code>>=</code>	Maior ou igual a.	2
<code><</code>	Menor que.	2
<code><=</code>	Menor ou igual a.	2
<code>==</code>	Igual.	2
<code>!=</code>	Diferente.	2

Todos os operadores relacionais tomam dois operandos e retornam como resultado um valor do tipo `bool`, ou seja, um resultado que só pode ser `False`, quando a relação indicada não existe (é falsa), ou `True`, quando se confirma a relação indicada (ou seja, é verdadeira). Exemplos simples do uso destes operadores estão no fragmento que segue, no qual se verifica se o valor 1964 *é menor* que 1995, que produz um retorno `True`, e se 1995 *é maior* que 2021, que resulta em `False`.

```
1964 < 1995
>>> True
1995 > 2021
>>> False
```

O próximo fragmento mostra outros usos destes operadores, onde se compara o conteúdo de `A` (que é 1931) e o literal 2021 em relação a sua igualdade, o que resulta em `False`, e verifica se o literal 1995 é diferente de `A`, que produz `True`.

```
A = 1964
A == 2021
>>> False
1995 != A
>>> True
```



Observe com atenção o uso dos operadores de atribuição `=` e de igualdade `==`, que produzem efeitos bastante distintos.

Como será visto na seção 2.5, os operadores relacionais pode ser combinados com operadores aritméticos e lógicos para formar expressões compostas capazes de verificar relações mais complexas.

2.4.3 Operadores Lógicos

Os operadores lógicos, listados da Tabela 2.5, permitem realizar as operações fundamentais da álgebra de Boole que são a conjunção (*e-lógico*), a disjunção (*ou-Lógico*) e a negação (*não-lógico*).

Tabela 2.5: Operadores lógicos

Operador	Operação	Aridade
<code>and</code>	E-lógico (conjunção).	2
<code>or</code>	Ou-lógico (disjunção).	2
<code>not</code>	Não-lógico (negação).	1
<code>in</code>	Membro de.	2
<code>is</code>	Identidade.	2

As operações realizadas por estes operadores consideram os valores lógicos `False` e `True`, do tipo `bool`.



O Python, como a linguagem C, considera valores 0 como **False** e valores diferentes de zero como **True**, realizando esta equivalência *lógico-numérica* automaticamente.

A operação de conjunção ou *e-lógico* verifica o estado lógico de seus dois operandos e retorna um resultado verdadeiro (**True**) apenas se ambos são verdadeiros, como mostra a Tabela 2.6, que toma as variáveis A e B como seus operandos.

Tabela 2.6: Tabela-verdade¹⁸ do *e-lógico* (conjunção)

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True

No fragmento que segue, verifica-se um dos resultados possível da operação *e-lógico*.

```
A = True
B = False
A and B
```

```
>>> False
```

Variáveis do tipo `bool` e outras numéricas podem ser combinadas com os operadores lógicos devido a equivalência *lógico-numérica* do Python, como segue.

```
B = True
10 and B
```

```
>>> True
```

A operação de disjunção ou *ou-lógico* também verifica o estado lógico de seus dois operandos, mas retorna um resultado falso (**False**) apenas se ambos os operandos são falsos, como mostra a Tabela 2.7.

Tabela 2.7: Tabela-verdade do *ou-lógico* (disjunção)

¹⁸Uma *tabela-verdade* mostra todas as combinações possíveis dos operandos de uma função lógica e seus resultados. O número de combinações possíveis sempre é 2operandos.

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True

O próximo fragmento mostra um dos resultados possível da operação ou-lógico.

```
A = True
B = False
A or B
```

```
>>> True
```

Finalmente, a operação de negação ou *não-lógico* retorna o oposto (ou inverso) de seu único operando, ou seja, quando este tem valor **False**, sua negação retorna **True**, e vice-versa, como na Tabela 2.8.

Tabela 2.8: Tabela-verdade do *não-lógico* (negação)

A	not A
False	True
True	False

O uso do operador não-lógico é bastante direto, como segue:

No fragmento que segue, verifica-se um dos resultados possível da operação e-lógico.

```
A = False
not A
```

```
>>> True
```

Os operadores lógicos **and**, **or** e **not** permitem conectar logicamente o resultado de diferentes expressões aritméticas, relacionais ou lógicas, o que permite construir expressões compostas de várias partes e, portanto, mais complexas, como será visto na seção 2.5.

Os operadores **in** (*teste de membro*) e **is** (*teste de identidade*) serão vistos nas seções que tratam de estruturas de dados e uso de objetos.

2.5 Expressões

Uma *expressão* é uma combinação de valores literais (seção 2.1), variáveis (seção 2.3) e operadores (seção 2.4), que produz um resultado como consequência do encadeamento dos elementos nela indicados. A determinação do valor resultante de uma expressão é que se denomina *avaliação da expressão*.

O Python avalia as expressões da *esquerda para direita*, ou seja, no sentido usual de leitura, por exemplo, considere as variáveis `x` e `y` inicializadas como seguem;

```
x = 2
y = 3
```

Uma expressão simples pode combinar valores literais e variáveis, como segue:

```
1 + x + y + 4
```

```
>>> 10
```

O resultado, 10, é obtido da soma dos valores expressos diretamente pelos literais e recuperados das variáveis indicadas, ou seja, `1 + 2 + 3 + 4`.

Podemos nos referir às expressões conforme o resultado que produzem, ou seja, existem:

- expressões de *atribuição*, não produzem resultado, mas o *efeito* de determinar o valor de variáveis com o resultado de uma expressão;

```
z = 7
```

- expressões *aritméticas*, que combinam valores literais, variáveis e operadores aritméticos para produzir um resultado numérico (de tipos `int`, `float` ou `complex`);

```
2 * z + 1
```

```
>>> 15
```

- expressões *relacionais*, que combinam expressões aritméticas e operadores relacionais para produzir um resultado lógico (de tipo `bool`); e

```
2 * x + 1 >= 10
```

```
>>> False
```

- expressões *lógicas*, que combinam valores literais, variáveis e expressões relacionais para produzir um resultado lógico (de tipo `bool`).

```
z == 7 and 2 * x + 1 <= 10
```

```
>>> True
```

Assim, as expressões podem combinar e utilizar operadores diferentes, como:

```
2 * x + y / 4
```

```
>>> 4.75
```

Aqui, o resultado 4.75 mostra que, quando operadores diferentes são misturados, a ordem de avaliação esquerda-para-direita é modificada. Isto ocorre porque alguns operadores possuem maior *prioridade* (ou *precedência*).

A *prioridade* dos operadores, ou sua *precedência*, é o critério matemático que estabelece a ordem com que os operadores serão executados em uma expressão, além de como os operadores envolvidos serão tomados (sua *associatividade*).

Para toda e qualquer expressão, sempre são aplicadas as regras de precedência da linguagem, garantindo que a expressão produza sempre o mesmo resultado, independentemente da plataforma ou do computadores utilizado.

2.6 Prioridade dos operadores

Como uma expressão pode combinar operadores diferentes e com aridade distinta, é necessário estabelecer um critério para determinar qual a ordem de execução dos operadores, garantindo resultados consistentes na avaliação da expressões, seja qual for a combinação empregada.

A Tabela 2.9 relaciona a prioridade dos operadores em Python, da maior (nível 1) para a menor (nível 18). Várias das indicações tratam de construções que serão vistas nos próximos capítulos deste material.

Tabela 2.9: Prioridade (precedência) dos operadores em Python

Nível	Operadores	Descrição
1	$(expr)$, $[expr,...]$, $\{ch:val\}$, $\{expr,...\}$	Expressões parentisadas, listas, dicionários e conjuntos
2	$[idx]$, $[ini:fim]$, $x(args)$, $.$	Subscrição, fatiamento, passagem de argumentos, seleção
3	<code>await x</code>	Expressão <code>await</code>
4	<code>**</code>	Potenciação
5	<code>+x</code> , <code>-x</code> , <code>~</code>	Positivo, negativo, não <i>bitwise</i>
6	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>	Multiplicação, divisão, divisão inteira, módulo
7	<code>+</code> , <code>-</code>	Adição, subtração
8	<code><<</code> , <code>>></code>	Deslocamento à esquerda e à direita
9	<code>&</code>	E <i>bitwise</i>
10	<code>^</code>	Ou-exclusivo <i>bitwise</i>
11	<code> </code>	Ou <i>bitwise</i>
12	<code><</code> , <code><=</code> , <code>></code> , <code>>=</code> , <code>==</code> , <code>!=</code> , <code>in</code> , <code>not in</code> , <code>is</code> , <code>not is</code>	Relacionais, teste de membro e teste de identidade
13	<code>not x</code>	Não lógico
14	<code>and</code>	E lógico
15	<code>or</code>	Ou lógico
16	<code>if - else</code>	Expressão condicional
17	<code>lambda</code>	Expressão lambda
18	<code>=</code>	Atribuição

Os operadores existentes num mesmo nível possuem a mesma precedência, sendo assim avaliados conforme encontrados da esquerda para a direita.

Os parênteses são operadores especiais, que podem ser utilizados para alterar a precedência pré-estabelecida de avaliação dos operadores, permitindo determinar uma sequência específica para o cálculo de uma expressão. Sempre é avaliado o conteúdo dos parênteses mais internos, prosseguindo com o conteúdo dos mais externos, até que a expressão seja completamente avaliada. Dentro de cada parênteses, a prioridade dos operadores é aplicada normalmente. A expressão $9 * 5 + 2$ equivale à $(9 * 5) + 2$, pois os parênteses não alteram a prioridade das operações, mas que é diferente de $9 * (5 + 2)$, cujo parênteses *força* que a soma $5 + 2$ ocorra *antes* da multiplicação, como mostram os fragmentos que seguem.

```
A = 9
B = 5
C = 2
A * B + C
```

```
>>> 47
```

```
A * B + C == (A * B) + C
```

```
>>> True
```

Mas a próxima expressão só poderá ser avaliada corretamente se estiverem presentes os parênteses indicados, caso contrário será obtido o resultado das expressões anteriores:

```
(A * B) + C == A * (B + C)
```

```
>>> False
```



O uso de parênteses, mesmo quando indicando a ordem natural de precedência, permite construir expressões cuja leitura é mais fácil, além de evitar alguns erros, constituindo assim uma boa prática de programação.

2.7 Atribuição

A *atribuição* é uma operação importante, pois permite determinar o valor das variáveis de um programa. Vale repetir que o uso das variáveis permite que o programador armazene valores literais ou resultados de cálculos diversos, para que possam ser utilizados em etapas posteriores do programa, evitando a repetição de sua entrada ou de seu processamento.

A atribuição simples é realizada por meio do operador = segundo a sintaxe:

variável = expressão

No fragmento que segue, temos a atribuição dos diferentes tipos de expressões à diferentes variáveis, utilizadas nas próprias expressões.

```
v = 2 # atribuição de literal
x = v # atribuição de variável
y = 2.5 * x + 1.7 # atribuição de expressão aritmética
w = x < y # atribuição de expressão relacional
z = w or y # atribuição de expressão lógica
print(f'v = {v}, x = {x}, y = {y}, w = {w}, z = {z}')
```

```
>>> v = 2, x = 2, y = 6.7, w = True, z = True
```



Python permite atribuir um mesmo valor para várias variáveis fazendo, por exemplo, `x = y = z = 0`, onde as variáveis `x`, `y`, `z` recebem o valor 0.

2.7.1 Atribuição Múltipla

Python permite a atribuição simultânea de múltiplas variáveis, com a sintaxe:

```
variável_1, variável_2 [,...] = expressão_1, expressão_2 [,...]
```

Ou seja, uma lista de variáveis podem ser inicializadas ou alteradas em uma única operação de atribuição, desde que sejam indicadas uma expressão para cada variável, como no exemplo:

```
a, b, c = 0, 1.5 * x - y, w or y
print(f'a = {a}, b = {b}, c = {c}')
```

```
>>> a = 0, b = -3.7, c = True
```

2.7.2 Atribuição Composta

Na programação é bastante comum que uma variável tenha seu valor modificado por meio de uma expressão da qual faz parte, ou seja, a própria variável é usada no cômputo de seu próximo valor, como por exemplo:

```
a = a + 1
b = b * 2
v = v - a / 2
```

Quando for identificada a situação que segue, envolvendo um operador aritmético:

```
var = var op expressao
```

Podem ser aplicados os operadores de atribuição compostos, que combinam a atribuição com o operador aritmético presente na expressão, ou seja:

```
var op= expressao
```

Assim o fragmento anterior pode ser reescrito como:

```
a += 1
b *= 2
v -= a / 2
```

A atribuição composta oferece uma pequena simplificação nas expressões, mas constitui um recurso útil.

2.8 Funções

Durante a programação é bastante comum que tarefas determinadas sejam realizadas com muita frequência, o que sempre é natural em praticamente qualquer área do conhecimento. Para evitar a repetição de um segmento de código utilizado muitas vezes, é comum a construção de uma *função*.

Conforme Brandão (BRANDÃO, 2021):

A ideia básica de uma função, implementada em alguma linguagem de programação, é encapsular um código que poderá ser invocado/chamado por qualquer outro trecho do programa. Seu significado e uso são muito parecidos com o de funções matemáticas, ou seja, existe um nome, uma definição e posterior invocação à função.

Uma função, um procedimento ou uma subrotina nada mais são do que segmentos de código, organizados de uma maneira que possam ser reutilizados por um programa. Cada linguagem de programação tem uma sintaxe própria para a definição de funções, mas é comum que cada função:

- tenha um nome que permita seu acionamento;
- receba parâmetros, que permitem passar dados para a função;
- um trecho de código que realiza uma tarefa específica;
- possa retornar um resultado.

As funções podem ser usadas na construção de expressões, ampliando muito suas possibilidades de realização de cálculos, além de prover enorme simplificação.

O Python dispõe de um conjunto de funções pré-definidas, que podem ser utilizadas a qualquer momento, denominadas funções *built-in*. Algumas destas funções são `abs()`, `bin()`, `bool()`, `chr()`, `float()`, `hex()`, `id()`, `input()`,

`int()`, `len()`, `max()`, `min()`, `oct()`, `ord()`, `pow()`, `print()`, `round()`, `str()` e `type()`.

Seguem as descrições e exemplos de uso destas funções.

abs(x) Retorna o valor absoluto (sem sinal) de um número, que pode ser inteiro, real ou complexo (quando sua magnitude é retornada).

```
A = -7.5 # um real negativo qualquer
abs(A)
```

```
>>> 7.5
```

bin(x) Converte um número inteiro em uma *string* (um texto) binária, com prefixo `0b`.

```
bin(10) # binário de um inteiro qualquer
```

```
>>> '0b1010'
```

bool(x) Avalia o valor recebido e retorna `False` ou `True` de acordo com as regras de equivalência do Python, por exemplo, números diferentes de zero são considerados `True`. Se o argumento é omitido, retorna `False`.

```
bool('jandl') # um literal qualquer
```

```
>>> True
```

chr(x) Retorna uma *string* contendo o caractere correspondente o *code point* Unicode do inteiro dado.

```
chr(80) # inteiro que corresponde ao caractere 'P'
```

```
>>> 'P'
```

float(x) Retorna um número real, do tipo `float` correspondente ao número ou *string* fornecido. Se o argumento é omitido, retorna `0.0`.

```
float('314E-2') # converte string em float
```

```
>>> 3.14
```

hex(x) Converte um número inteiro em uma *string* (um texto) hexadecimal em minúsculas, com prefixo `0x`.

```
hex(2021) # hexadecimal de um inteiro qualquer
```

```
>>> '0x7e5'
```

id(objeto) Retorna um número inteiro único que identifica o objeto recebido. Na maior parte das implementações de Python, este número corresponde ao endereço de memória onde o objeto está alocado.

```
id(A) # identidade da variável A
```

```
>>> 579352232
```

input(prompt) Exibe a mensagem de *prompt*, se fornecida e retorna a leitura do texto digitado pelo usuário (até que seja pressionado ENTER).

```
nome = input('Digite seu nome: ') # entrada de dados
```

int(x) Retorna um número inteiro, do tipo `int` correspondente ao número ou *string* fornecido. Trunca as casas decimais quando o argumento é um valor real (de tipo `float`). Se o argumento é omitido, retorna 0.

```
int('2013') # converte string em int
```

```
>>> 2013
```

len(objeto) Retorna o comprimento (o número de itens) de um objeto, que deve ser uma sequência ou coleção.

```
len('Python') # retorna o número de caracteres
```

```
>>> 6
```

max(arg0, *arg1) Retorna o maior item de um *iterable*, ou entre dois ou mais argumentos fornecidos.

```
max(736, 13) # retorna o maior item
```

```
>>> 736
```

min(arg0, *arg1) Retorna o maior item de um *iterable*, ou entre dois ou mais argumentos fornecidos.

```
min(736, 13) # retorna o menor item
```

```
>>> 13
```

oct(x) Converte um número inteiro em uma *string* (um texto) octal, com prefixo 0o.

```
oct(2021) # octal de um inteiro qualquer
```

```
>>> '0o3745'
```

ord(c) Dada uma *string* contendo um caractere, retorna seu *code point* Unicode inteiro.

```
ord('P') # retorna o código Unicode da letra P
```

```
>>> 80
```

pow(base, exp) Retorna o resultado da base elevada ao expoente. A base e o expoente devem ser valores numéricos, inteiros ou reais.

```
pow(5, 4) # retorna 5 a quarta potência
```

```
>>> 625
```

print(*argumento) Imprime, no dispositivo de saída padrão (console), um ou mais argumentos recebidos. Possibilita controlar o separador dos argumentos e o finalizador de cada impressão. Os argumentos podem ser de qualquer tipo (numéricos, de texto, lógico ou outros).

```
A = 1
B = 2.3
print('Soma de', A, 'e', B, '=', A + B) # saída de dados
```

```
>>> Soma de 1 e 2.3 = 3.3
```

round(x, [digitos]) Efetua o arredondamento do valor recebido como um inteiro, ou, opcionalmente, com o número de dígitos indicado.

```
round(12.345678, 2)
```

```
>>> 12.35
```

str(objeto) Retorna uma representação de texto do objeto dado.

```
str(print)
```

```
>>> '<built-in function print>'
```

type(objeto) Retorna o tipo do objeto fornecido.

```
type('0x1234ABCD')
```

```
>>> <class 'str'>
```

A lista completa das funções *built-in* do Python pode ser consultada em sua documentação.

O Capítulo 8 detalhará a construção de funções, assim como sua utilização, discutindo também suas vantagens e desvantagens. A linguagem Python, tal como as demais linguagens de programação, permite explorarmos a utilização das funções.

Chapter 3

Entrada e Saída

Os programas de computador podem manipular dados e realizar cálculos, desde simples até os mais complexos. O que os torna ainda mais convenientes é a possibilidade de obterem estes dados de fontes diferentes, o que permite que executem suas tarefas considerando valores diferentes, ampliando muito sua utilidade.

Os dados requeridos para que um programa possa realizar os cálculos necessários são considerados como *dados de entrada*. Já os resultados desejados, produzidos pela execução do programa, são entendidos como *dados de saída*. A sequência de operações que o programa realiza, ou seja, a computação realizada a partir dos dados de entrada para produzir os dados de saída, é o que chamamos de *processamento de dados*.

Muitos programas são construídos para trabalhar dessa maneira, efetuando uma etapa de *entrada de dados*, seguida do *processamento de dados*, que produz os resultados que serão apresentados ou armazenados na etapa de *saída de dados*, como ilustrado na Figura 3.1.

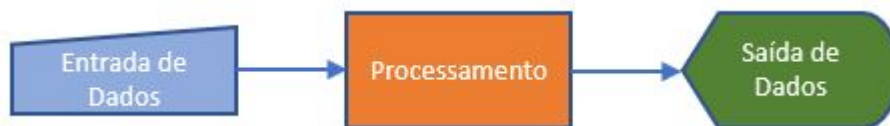


Figure 3.1: Etapas típicas de um programa

Programas mais sofisticados podem ter várias etapas distintas de entradas, utilizar dados de fontes diversas, podem ter o processamento subdividido em partes específicas, além de produzir várias saídas enviadas para destinos diferentes. As possibilidades, embora infinitas, são variações do tema *entrada-processamento-saída*.

Os programas Python são executados pelo interpretador da linguagem que, *naturalmente*, utiliza o console da plataforma, ou seja, o ambiente interativo e em modo texto oferecido pelos sistemas operacionais, mais conhecidos como *prompt de comandos* no Microsoft Windows ou como *terminais* nas distribuições Linux, Apple Mac OS e outros sistemas semelhantes. O console é composto por dois dispositivos: o teclado, considerado como a *entrada-padrão*, e o monitor, tratado como *saída-padrão* de todos esses sistemas operacionais.

Sendo assim, as operações de *entrada* e de *saída* realizadas no console dos sistemas são essenciais.

3.1 Saída Simples

A apresentação de resultados é uma etapa fundamental de qualquer programa, pois permite comunicar ao usuário informações sobre andamento do programa, bem como os resultados produzidos e, até mesmo, erros ou avisos sobre situações específicas encontradas durante sua execução.

A função *built-in* `print()` tem como objetivo escrever os argumentos que recebe na *saída-padrão*, ou seja, no console, produzindo texto legível pelo usuário como resultado. Sua sintaxe é simples:

```
print( argumento* )
```

Tipicamente a função `print()` recebe um ou mais argumentos separados por vírgulas. Cada argumento pode ser uma *string*, um número (inteiro, real ou complexo), um valor lógico ou uma variável, tal como segue:

```
print('Python, guia do programador') # str
print(2021) # int
print(7.654321) # float
print(True) # bool
z = 1.2 - 3.4j # complexo
print(z) # variável
```

```
>>> Python, guia do programador
>>> 2021
>>> 7.654321
>>> True
>>> (1.2-3.4j)
```

Neste fragmento, cada uso de `print()` recebeu um único argumento, dos tipos `str`, `int`, `float`, `bool` e `complex`. O de tipo `str` foi *transcrito* diretamente na

saída-padrão, ou seja, no console, enquanto os demais foram convertidos para o tipo `str` e então exibidos.

Mas `print()` pode receber qualquer quantidade e combinação de tipos de argumentos, os quais devem ser separados por uma vírgula (`,`), o que oferece inúmeras possibilidades, como no próximo fragmento que combina o nome da variável na *string* `'x ='` e o valor desta variável `x`:

```
x = 123456789
print('x =', x)
```

```
>>> x = 123456789
```

Aqui a função `print()` combinou um argumento do tipo `str` com outro do tipo `int`, convertendo cada um num texto, separado automaticamente por um espaço em branco, permitindo informar o usuário sobre o valor de uma variável específica do programa. O mesmo pode ser feito para mais variáveis, com textos complementares diversos.

```
x = 123456789
y = 9
print('x =', x, 'e y =', y)
```

```
>>> x = 123456789 e y = 9
```

Além de valores literais, a função `print()` aceita expressões, de todos os tipos, como argumentos, ampliando ainda mais as maneiras de gerar saídas nos programas.

```
x = 123456789
y = 9
print('x * y =', x * y)
```

```
>>> x * y = 1111111101
```

Deve ser observado que `'x * y ='` é uma *string* (um literal do tipo `str`), pois está delimitada por aspas-simples, enquanto `x * y` é uma expressão que envolve o produto das variáveis `x` e `y`, definidas no início do fragmento. Além disso, note que, no resultado produzido (*impresso no console*), a função `print()` adicionou um espaço em branco entre os dois argumentos, tornando-os mais claros.

3.1.1 Caracteres Especiais

Nos literais de texto, isto é, nas *strings*, é possível acrescentar caracteres especiais, em geral, caracteres *não imprimíveis*, como a quebra-de-linha, que oferecem maior controle sobre as mensagens exibidas por um programa. A Tabela 3.1 lista os caracteres especiais do Python, conhecidos também como *escape characters*.

Tabela 3.1: Caracteres especiais do Python

Código	Descrição
\'	Aspas simples
\"	Aspas duplas
\\	Barra invertida (<i>backslash</i>)
\n	Nova linha (<i>new line</i>)
\r	Retorno de carro (<i>carriage return</i>)
\t	Tabulação
\b	Retrocesso (<i>backspace</i>)
\o	Prefixo para valor octal
\x	Prefixo para valor hexadecimal

Todos os caracteres especiais são, na verdade, sequências de dois caracteres, onde o primeiro é obrigatoriamente a barra invertida (\). Assim, para que a própria barra invertida possa fazer parte de uma *string*, ela deve ser indicada como a sequência de escape \\\.

As aspas simples e duplas são os caracteres delimitadores das *strings*, assim devem ser precedidas por uma barra invertida para que possam fazer parte da própria *string*. O fragmento que segue mostra como incorporar as aspas e a barra invertida em *strings* exibidas pela função `print()`.

```
texto_1 = 'Uso de aspas \'simples\''
texto_2 = "ou \"duplas\" e da barra \\\!"
print(texto_1, texto_2)
```

```
>>> Uso de aspas 'simples' ou "duplas" e da barra \!
```

Também é possível incluir tabulações e quebras de linha numa *string*, com o uso das sequências de escape `\t` e `\n` respectivamente. A tabulação introduz um espaçamento horizontal, cujo tamanho, de fato, depende do programa ou interface em uso, embora seja comum sua expansão para 4 ou 8 espaços. Observe o fragmento que segue, onde uma mesma frase é impressa quatro vezes, sem e com o uso de tabulação e quebra de linha.


```
print("Um texto simples")
>>> Um texto simples
print("\tUm texto simples")
>>>     Um texto simples
print("Um texto\nsimples")
>>> Um texto
>>> simples
print("Um\ntexto\n\tsimples")
>>> Um
>>> texto
>>>     simples
```

Na saída produzida por este fragmento, percebemos que a inclusão de um `\t` no início da *string* produz um espaçamento em sua apresentação. Já o caractere especial `\n` gera uma quebra de linha, dividindo a exibição da *string* em duas linhas. Os efeitos destes caracteres e as demais sequências de escape pode ser combinado para criação de mensagens, como se vê no último uso de `print()`.

3.1.2 Controle de Separação e Finalização

Como visto nos exemplos anteriores, a função *built-in* `print()` adiciona, por padrão, um espaço em branco entre a impressão dos argumentos fornecidos, além disso, após imprimir o último argumento, finaliza a impressão com um caractere `\n`, preparando uma nova linha para a próxima impressão. Então temos três situações distintas no uso de `print()`:

1. Uso sem argumentos: impressão gera apenas uma quebra de linha.

```
print() # sem argumentos
```

2. Uso com um único argumento: a impressão adiciona um espaço em branco entre os argumentos e uma quebra de linha (`\n`) para finalizar a saída.

```
print("Argumento_1") # um argumento
>>> Argumento_1
```

3. Uso com dois ou mais argumentos: a impressão adiciona um espaço em branco entre todos os argumentos e uma quebra de linha (`\n`) para finalizar a saída.

```
print("Argumento_1", "Argumento_2") # dois argumentos
>>> Argumento_1 Argumento_2
```

Neste exemplo, foram usados apenas argumentos do tipo *string* para facilitar a compreensão, mas poderiam ser de qualquer tipo, ou ainda expressões. A Figura 3.1 ilustra a inserção do separador e do finalizador realizada pelo `print()`.

```
print( "Argumento_1", "Argumento_2", "Argumento_3" )
```

```
>>> Argumento_1 Argumento_2 Argumento_3
>>>
```

Figure 3.2: Inserção de separador e finalizador no `print()`

Além disso, a função `print()` permite que outro separador ou finalizador sejam utilizados, ampliando ainda mais suas possibilidades de uso. Isto é possível com a indicação explícita do parâmetro `sep` para escolha de um novo separador e do parâmetro `end` para escolha de um novo finalizador de impressão. Veja no fragmento que segue o uso da *string* `--` como separador.

```
print("Argumento_1", "Argumento_2", sep='--')
```

```
>>> Argumento_1--Argumento_2
```

É fácil perceber que os argumentos passados para `print()` agora estão separados por `--`.



Para incluir o separador no início de uma impressão, ou em seu final, antes do finalizador, basta adicionar um argumento de tipo *string* vazio, ou seja, `' '`, por exemplo: `print(' ', 'Argumento 1', 'Argumento 2', ' ', sep=' | ')` que produz: `>>> | Argumento1 | Argumento2 |`

Segue outro fragmento que mostra o emprego do parâmetro `end` definido como 'separador `||`'.

```
print("Argumento_1", end='||')  
print("Argumento_2")
```

```
>>> Argumento_1||  
>>> Argumento_2
```

Com isso, observamos que a sintaxe mais completa da função *built-in* `print()` é, de fato:

```
print( argumento*, sep=' ', end='\n')
```

A função `print()`, embora simples, é muito flexível e seu uso indispensável.

3.2 Entrada

É comum que os programas sejam construídos para trabalhar seguindo as três etapas ilustradas na Figura 3.1: efetuando uma *entrada de dados*, seguida do *processamento de dados*, que produz os resultados apresentados como sua *saída de dados*. A entrada de dados é uma importante etapa que, em geral, dá início a esse processo.

A função *built-in* `input()` tem como objetivo capturar dados fornecidos pelo usuário do programa na *entrada-padrão*, usualmente o teclado, um dos dispositivos que fazem parte do console. Sua sintaxe é simples:

```
variavel = input( prompt )
```

A função `input()` pode receber um argumento opcional do tipo `str`, ou seja, uma *string* que serve como mensagem de orientação para o usuário, que deveria explicar qual dado é esperado, ou seja, o tipo de dado a ser fornecido, a faixa de valores aceita ou outra característica esperada. Após a impressão da mensagem na *saída-padrão*, aguarda que o usuário realize a digitação do dado, prosseguindo apenas quando é pressionada a tecla **ENTER**. Esta função então retorna o texto fornecido, usualmente armazenado em uma variável para uso futuro.

Isto significa que o uso de `input()` faz com que o programa pare sua execução, esperando pela digitação do usuário, prosseguindo apenas quando a *entrada de dados* é finalizada com o acionamento de **ENTER**, armazenando o texto capturado na variável indicada pelo programador, o que é bastante conveniente.

```
nome = input('Digite seu nome: ')  
print('O nome digitado foi:', nome)
```

```
>>> Digite seu nome: Peter
>>> O nome digitado foi: Peter
```

Cada valor de entrada necessário à um programa deve ser lido por meio de uma chamada à função `input()`. Por exemplo;

```
nome = input('Digite seu nome: ')
cidade = input('Informe sua cidade: ')
print('Dados lidos:', nome, ', ', cidade)
```

```
>>> Digite seu nome: Peter
>>> Informe sua cidade: Itatiba
>>> Dados lidos: Peter , Itatiba
```

Embora o uso da função `input()` seja simples, é necessário destacar que todos os dados lidos são tratados como texto, ou seja, do tipo `str`. Considere o fragmento que segue.

```
a = input('Digite um valor inteiro: ')
b = input('Digite outro valor inteiro: ')
print('Valores lidos:', a, ', ', b)
```

```
>>> Digite um valor inteiro: 12
>>> Digite outro valor inteiro: 34
>>> Valores lidos: 12 , 34
```

Se o valor contido nas variáveis `a` e `b` for somado, teremos:

```
res = a + b
print('a + b =', res)
```

```
>>> a + b = 1234
```

O que parece ser um erro, na verdade está correto, pois a função `input()` trata todas as entradas fornecidas pelo usuário como texto, ou seja, valores do tipo `str`, que ao serem *somados* são, de fato, *concatenados*, ou seja, o texto de um operando é justaposto ao texto do outro operando. Podemos comprovar que `a`, `b` e `res` são do tipo `str` com:

```
print('type(a) =', type(a), '\ntype(b) =', type(b),
      '\ntype(res) =', type(res))
```

```
>>> type(a) = <class 'str'>
>>> type(b) = <class 'str'>
>>> type(res) = <class 'str'>
```

Para que possamos utilizar os dados fornecidos pelo usuário como valores numéricos, é necessário converter o texto lido no tipo de dados desejado. A conversão de texto para inteiro é feita com auxílio da função *built-in* `int()`, que pode ser usada como segue:

```
a = int(input('Digite um valor inteiro: '))
b = int(input('Digite outro valor inteiro: '))
print('Valores lidos:', a, ',', b)
```

```
>>> Digite um valor inteiro: 12
>>> Digite outro valor inteiro: 34
>>> Valores lidos: 12 , 34
```

Note que o resultado produzido por `input()`, que é o texto capturado como entrada do usuário, é passado como argumento da função `int()`, que realiza sua conversão para um valor inteiro. Agora é obtido o resultado esperado quando o valor contido nas variáveis `a` e `b` é somado:

```
res = a + b
print('a + b =', res)
```

```
>>> a + b = 46
```

A conversão de texto para valores reais, do tipo `float` é semelhante, como segue:

```
x = int(input('Primeiro valor real: '))
y = int(input('Segundo valor real: '))
z = x + y
print('Soma:', z)
print('Produto:', x * y)
```

Agora que sabemos como *computar*, isto é, efetuar cálculos diversos, e também realizar as operações de *entrada e saída*, torna-se possível construir programas Python para resolver muitos tipos de problemas.

```
>>> Primeiro valor real: 3.45
>>> Segundo valor real: 8.76
>>> Soma: 12.21
>>> Produto: 30.222
```



Atenção com o uso de `input()` quando são requeridos dados numéricos ou lógicos, pois isto requer transformar o texto lido no tipo desejado, usualmente com as funções *built-in* `int()`, `float()` e `bool` que permitem as conversões para os respectivos tipos.



Se o usuário fornece uma entrada inválida, isto é, que não pode ser convertida pelas funções *built-in* `int()`, `float()` e `bool`, é lançada uma exceção `ValueError` indicando o problema. Além disso, caso esta exceção não seja tratada, o programa é abortado.

3.3 Saída Formatada

Como veremos, em muitos programas surgirá a necessidade de apresentar os resultados de maneira mais organizada, eventualmente com uma aparência mais familiar para o usuário. Para isso, é necessária a aplicação de padrões de apresentação aos dados produzidos pelo programa, ou mais simplesmente, é necessário *formatar* a saída.

Esta seção é dedicada a explorar capacidades um pouco mais avançadas do Python em relação a produção de saída de dados formatada. Exatamente por isso, sua leitura pode ser adiada até o momento em que sua necessidade se apresente, quando os exemplos dos próximos capítulos não forem suficientes para a compreensão das técnicas de formatação, ou quando um maior aprofundamento for desejado.

Existem algumas maneiras convenientes de produzir saída formatada no Python, obtido com:

- A função *built-in* `format()`;
- A interpolação de *strings* com `%()`; e
- As *formatted literal strings*, ou simplesmente, as *f-strings*.

3.3.1 Formatação com função `format()`

A função *built-in* `format()` é destinada a converter um valor em sua representação *formatada*, ou seja, aplica um padrão de apresentação, definido por uma *string* que emprega uma sintaxe própria. Sua utilização permite controlar como valores numéricos e não numéricos são exibidos para o usuário. Esta função é indiretamente utilizada pela função de mesmo nome `format()` da classe `str`, com a sintaxe que segue:

```
'padrao_de_formatacao'.format( *valor)
```

O padrão de formatação é uma *string* na qual são inseridos *marcadores de formatação* ou *placeholders*, um para cada valor que se pretende formatar e exibir. Na função `format()`, aplicada ao *padrão de formatação* por meio do operador `.` (denominado *seletor*¹), devem ser dispostos os valores correspondentes aos marcadores de formatação. Seguem alguns exemplos para auxiliar na compreensão do uso da função `format()`.

Os *marcadores de formatação* tem uma estrutura simples:

```
{ [id] [: formato] }
```

Todos os marcadores são delimitados por chaves obrigatórias, dentro das quais são dispostos um `id` e um `formato`, separados por dois-pontos quando o segundo elemento está presente. O `id` é um número inteiro opcional que pode ser utilizado para identificar um dos valores presentes em `format()`, o que permite sua reutilização, evitando sua repetição. Seu uso será visto mais à frente. O `formato` também é opcional e tem uma sintaxe própria, na qual são utilizados os especificadores relacionados na Tabela 3.2.

Tabela 3.2: Alguns especificadores de formatação do Python

Código	Descrição
<code>b</code>	Inteiro em formato binário (base 2)
<code>c</code>	Inteiro como caractere
<code>d,n</code>	Inteiro decimal
<code>e,E</code>	Real em notação científica
<code>f,F</code>	Real com precisão fixa
<code>o</code>	Inteiro em formato octal (base 8)
<code>%</code>	Porcentagem
<code>s</code>	String (cadeia de caracteres)
<code>x,X</code>	Inteiro em formato hexadecimal (base 16)

No fragmento que segue, o número inteiro contido na variável `x` é impresso sem aplicação de um padrão de formatação, mas como uso da função `format()`.

```
x = 15
print('{}'.format(x))
```

```
>>> 15
```

¹No Python tudo é tratado como um objeto, até mesmo os literais numéricos e as *strings*. Assim, o uso do operador *seletor* possibilita o acesso aos métodos destes objetos, ou seja, às funções-membro que estão disponíveis em suas respectivas classes.

Neste exemplo, o marcador vazio {} serve apenas para indicar onde o valor da variável `x` deve ser colocado na *string de formatação*, funcionando como um *marcador posicional*. Esta alternativa é bastante conveniente quando é necessário intercalar vários valores em uma mensagem, como no exemplo que segue.

```
x = 15
y = 19
print('Se x =', x, 'e y =', y, 'então x == y -->',
      x == y)
```

```
>>> Se x = 15 e y = 19 então x == y --> False
```

```
print('Se x = {} e y = {}, então x == y --> {}'.format(x, y, x == y))
```

```
>>> Se x = 15 e y = 19, então x == y --> False
```

No próximo fragmento, a função `print()` recebe uma *string* de formatação que contém três marcadores: os dois primeiros, com a especificação do tipo do valor `{:d}`, recebem o conteúdo das variáveis `x` e `y`; e o último, um marcador posicional {}, recebe o resultado da expressão `x * y`.

```
x = 20
y = 6
print('{:d} * {:d} = {}'.format(x, y, x * y))
```

```
>>> 20 * 6 = 120
```

Embora o efeito produzido aqui seja semelhante, o funcionamento destes marcadores é distinto. No marcador posicional, o valor correspondente é transformando numa *string* inclusa na posição dada na *string de formatação*, sendo indiferente o tipo do valor ali inserido. Já nos marcadores com especificadores é necessário que o valor correspondente seja de tipo compatível com o indicado, caso contrário será lançada a exceção `ValueError`.

Segue outro fragmento, onde outro número inteiro é exibido de maneiras diferentes. Observe que o padrão de formatação pode conter uma mensagem na qual se intercalam os marcadores de formatação desejados. Observe também que uma *string* pode ser dividida em duas ou mais linhas, fechando seus delimitadores e usando uma barra invertida \ para indicar a divisão.


```
r = 12345
print('Valor não formatado {:d} e com formato de 7 ' \
      'colunas {:7n}'.format(r, r))
```

```
>>> Valor não formatado 12345 e com formato de 7 colunas 12345
```

Dois aspectos deve ser observados aqui: o uso direto de um especificador, tal como `d`, apenas apresenta o valor correspondente (o conteúdo da variável `r`), sem formatação; mas ao acrescentar o inteiro `7` como prefixo do especificador `n`, indica-se que o valor deverá ocupar sete colunas, ou seja, sua apresentação ocupará o espaço fixo de sete colunas (*largura*). Este expediente pode ser utilizado com qualquer especificador para determinar a largura da apresentação.

Os especificadores para inteiros tem, então, a estrutura `[s][w]d` ou `[s][w]n` onde:

- `s`, opcional, indica como o sinal é mostrado: ausente ou `-`, apenas o sinal negativo é exibido; `+` exibe sinais positivos e negativos;
- `w`, opcional, estabelece o número de colunas da largura da apresentação;
- `d` ou `n` são os códigos dos especificadores para valores inteiros.

O fragmento que segue mostra o uso do indicador de sinal.

```
print('{:-8d}; {: -8n}'.format(123456, -654321))
```

```
>>> 123456; -654321
```

```
print('{:+8d}; {:+8n}'.format(123456, -654321))
```

```
>>> +123456; -654321
```



Quando a quantidade de caracteres necessárias para apresentar o valor dentro do padrão é menor do que largura indicada, espaços em branco são acrescentados, garantindo que o valor formatado ocupe a largura estabelecida. Nesta situação, os valores são alinhados à direita. Nada acontece quando a largura indicada é a mesma requerida pelo valor formatado.



Quando o valor requer mais caracteres de apresentação do que a largura indicada, o padrão de formatação é desconsiderado e o valor é apresentado integralmente.

Em algumas situações, um mesmo valor é utilizado mais de uma vez na saída produzida. Para evitar repetição do valor literal, variável ou expressão, podem ser empregados os **ids** nos marcadores para selecionar qual valor da lista contida em `format()` deve ser usado.

No fragmento que segue, a *string* de formatação possui quatro marcadores, apenas com a indicação do **id**, onde o número 0 indica do primeiro valor da lista presente em `format()` (neste exemplo a variável `x`), 1 indica o segundo (aqui a variável `y`) e assim sucessivamente.

```
x = 'ra'
y = 'qua'
print('A{0}{0}{1}{0}!'.format(x, y))
```

```
>>> Araraquara!
```

Observe que o mesmo **id** pode ser indicado na *string* de formatação, de maneira repetida e em qualquer ordem, evitando a repetição dos valores presentes em `format()`.

A formatação de números reais é bastante flexível, permitindo controlar não apenas a largura da apresentação, mas também o número de casas decimais exibidas e até mesmo a notação empregada.

No fragmento que segue o número real contido na variável `z` é impresso de quatro maneiras diferentes, genericamente, real sem formatação e com dois usos da notação de *real com precisão fixa*, na qual se estabelece um número fixo de casas decimais.

```
z = 123.456789
print('z = {0} ou {0:f} ou {0:8.2f} ou {0:10.4f}!'.format(z))
```

```
>>> z = 123.456789 ou 123.456789 ou 123.46 ou 123.4568!
```

Neste fragmento, todos os marcadores fazem uso do primeiro valor contido em `format()`. O primeiro marcador `{0}` exibe o valor como uma *string*, sem verificar seu tipo. O segundo marcador, `{0:f}` verifica se o valor é um número real (de tipo `float`), mas sem aplicar qualquer formatação. Os demais marcadores

aplicam a formatação de número *real com precisão fixa*: `{0:8.2f}` indica apresentação com oito colunas no total, das quais uma para o separador decimal e duas casas decimais; enquanto `{0:10.4f}` requer apresentação com dez colunas no total, uma para o separador decimal e quatro casas decimais.

Os especificadores para reais tem, então, a estrutura `[s][w][.p]f` ou `[s][w][.p]e` onde:

- **s**, opcional, indica como o sinal é mostrado: ausente ou -, apenas o sinal negativo é exibido; + exibe sinais positivos e negativos;
- **w**, opcional, estabelece o número de colunas da largura da apresentação;
- **p**, opcional, complemento que indica o número de casas decimais apresentado;
- **f** ou **e** são os códigos dos especificadores para valores reais, o primeiro para precisão fixa e o segundo para notação científica.

Quando **p** é indicado, aparece separado de **w** por um ponto, e determina o número de colunas da apresentação reservado para as casas decimais e o separador decimal.

O fragmento que segue exemplifica a formatação de uma variável real **z** com precisão fixa e com notação científica.

```
z = 123.456789
print('z = {0} ou {0:.4f} ou {0:+.4e} ou {0:14.5E}!'
      .format(z))
```

```
>>> z = 123.456789 ou 123.4568 ou +1.2346e+02 ou 1.23457E+02!
```

As *strings* também podem ter sua exibição ser controlada com o especificador **s**, como mostra o fragmento que segue, onde barras verticais foram adicionadas para indicar os limites da *string* impressa.

```
texto = 'Peter Jandl Jr'
print('Nome: |{0}|\nNome: |{0:25s}|'.format(texto))
```

```
>>> Nome: |Peter Jandl Jr|
>>> Nome: |Peter Jandl Jr      |
```

```
print('Nome: |{0:>25s}|\nNome: |{0:^25s}|\nNome: |{0:<25s}|'
      .format(texto))
```

```
>>> Nome: |                Peter Jandl Jr|
>>> Nome: |      Peter Jandl Jr          |
>>> Nome: |Peter Jandl Jr                |
```

O especificador `s` que tem a estrutura `[a][w]s` onde:

- `a`, opcional, controla o alinhamento: `<` para esquerda, padrão para maioria dos objetos; `>` para direita, padrão para valores numéricos; `^` para centralizado;
- `w`, opcional, indica o número de colunas da largura da apresentação;
- `s` é o código do especificador para *string*.

A sintaxe dos especificadores é muito flexível, embora relativamente complexa, de maneira que sua documentação deve ser consultada sempre que necessário.

3.3.2 Interpolação de *string*

O Python dispõe de outro idioma para formatação de valores, denominado *interpolação de strings*, que utiliza o operador `%` e uma série de valores. Observe o fragmento que segue, que produz a mesma saída formatada para uma variável real utilizando a função `format()` e a interpolação de *string* equivalente.

```
n0 = 745.02
print('Valor = R$ {:.2f}'.format(n0))
```

```
>>> Valor = R$ 745.02
```

```
print('Valor = R$ %.2f' % (n0))
```

```
>>> Valor = R$ 745.02
```

Na interpolação de *string*, cada marcador `{}` é substituído por um prefixo `%` (*placeholder*), usando a mesma sintaxe dos especificadores vistos seção 3.3.1; e o acionamento da função `format()` simplificado para `%()`, onde deve ser relacionados um valor para cada *placeholder*.

Este outro fragmento mostra a formatação de valores inteiros e reais, obtidos de variáveis e de expressões.

```
n1 = 190
n2 = 125
print('n1 = {:5d}, n2 = {:5d}, n1 / n2 = {:.4f}'
      .format(n1, n2, n1 / n2))
```

```
>>> n1 = 190, n2 = 125, n1 / n2 = 1.5200
```

```
print('n1 = %5d, n2 = %5d, n1 / n2 = %8.4f'
      % (n1, n2, n1 / n2))
```

```
>>> n1 = 190, n2 = 125, n1 / n2 = 1.5200
```

A interpolação de *string* é, em geral, um idioma mais compacto do que aquele requerido pela função `format()`, mas são, de fato equivalentes, portanto uma escolha pessoal de quem programa.

3.3.3 Formatação com *f-string*

A versão 3.6 do Python introduziu uma alternativa bastante interessante para formatação de saída denominada *formatted string literals* ou, mais simplesmente, *f-string*. Além de ser uma notação mais compacta e simples, é menos inclinada a erros e também mais eficiente, devendo constituir a escolha preferencial para esta tarefa.

Uma *f-string* deve ser iniciada com o prefixo `f` ou `F`. Nela podem ser inseridos um ou mais marcadores (*placeholders*) para indicar onde valores, formatados ou não, devem ser inseridos, como no fragmento que segue:

```
f0 = 745.02
i1 = 2021
s3 = 'Python'
print(f'Uma string {s3}, um inteiro {i1} e um real {f0}.')
```

```
>>> Uma string Python, um inteiro 2021 e um real 745.02.
```

O uso de uma *f-string* torna direta a inserção de valores não formatados, que são diretamente indicados dentro de chaves `{}`.

A formatação dos valores requer a adição de um especificador para cada marcador, com a mesma sintaxe daqueles utilizados pela função `format()` (seção 3.3.1), como no exemplo que segue, que compara o uso de `format()` e de *f-string*.

```
f0 = 745.02
print('Valor = R$ {:.2f}'.format(f0))
```

```
>>> Valor = R$ 745.02
```

```
print(f'Valor = R$ {n0:7.2f}')
```

```
>>> Valor = R$ 745.02
```

Note que o valor a ser inserido (literal, variável ou expressão) é indicado diretamente dentro do marcador, que tem a estrutura {<valor>[:<especificador>}}, como no exemplo: {n0:7.2f}, onde n0 é a variável, cujo valor é formatado, e 7.2f é o especificador da formatação (número real, com 7 colunas de apresentação e duas casas decimais).

No próximo fragmento, é comparado o uso de *interpolação de string* e de *f-string* na exibição de duas variáveis inteiras e uma expressão de resultado real.

```
n1 = 190
n2 = 125
print('n1 = %5d, n2 = %5d, n1 / n2 = %8.4f' % (n1, n2, n1 / n2))
```

```
>>> n1 = 190, n2 = 125, n1 / n2 = 1.5200
```

```
print(f'n1 = {n1:5d}, n2 = {n2:5d}, n1 / n2 = {n1 / n2:.4f}')
```

```
>>> n1 = 190, n2 = 125, n1 / n2 = 1.5200
```

As *f-strings*, tanto por sua simplicidade, como flexibilidade e eficiência, devem ser usadas sempre que possível.

Chapter 4

Sequenciação

A *sequenciação* é a habilidade requerida para organizarmos uma sequência de instruções que permita resolver um problema específico. De muitas maneiras, sequenciar instruções é o mesmo que criar um *algoritmo*¹, pois:

Na Matemática, representa um conjunto de regras para a resolução de um cálculo numérico, como, por exemplo, o algoritmo de Euclides encontra o máximo divisor comum de dois números inteiros. É também uma sequência de raciocínios ou operações que oferece a solução de certos problemas.

Na Informática, é um conjunto de regras que fornecem uma sequência de operações capazes de resolver um problema específico.

Uma definição um pouco mais precisa do ponto de vista da Ciência da Computação seria:

Um algoritmo é uma sequência organizada e finita de instruções que permite a solução de um problema específico ou de uma classe de problemas.

A habilidade da sequenciação trata, assim, de estabelecer uma sequência adequada de instruções para solução de um problema, ou seja, ela representa a lógica com a qual resolvemos um problema. A sequenciação determina como os dados de um problema serão processados, portanto combina todas as habilidades requeridas para a construção de programa de computador, que são a entrada e saída, a computação, a repetição, a decisão e a modularização.

¹Dicio (Dicionário On-Line de Português): <https://www.dicio.com.br/algoritmo/>

4.1 Programa Mínimo

O comportamento *natural* de um programa é ter suas instruções executadas uma a uma, da primeira até a última, na ordem em que se encontram. Isto é o que se denomina *fluxo sequencial*. As instruções, chamadas formalmente de diretivas (*statements*), são separadas, no Python, por uma quebra de linha, de maneira que um programa é um texto, contendo uma, várias ou muitas instruções organizadas numa sequência. O texto contendo as instruções de um programa é chamado de *programa-fonte*.

Na verdade, o Python é bastante direto para a construção de seus programas, pois uma sequência simples de instruções constitui um programa, de modo que o menor programa possível, ou seja, o *programa mínimo* seja aquele constituído de uma única instrução como no programa que segue, o clássico *Hello World*.

Esta simplicidade extrema torna o Python atraente tanto para aquele que está iniciando os estudos em programação, como para o especialista de outra área que pode se beneficiar da autonomia de criar seus próprios programas, assim como para os profissionais de Tecnologia da Informação que trabalham com desenvolvimento de software.

4.2 Estratégia E-P-S

Muitos programas de computador são construídos para realizar, da maneira mais simples possível, as tarefas necessárias para solucionar um problema. Em muitos casos, estas tarefas podem ser organizadas em três etapas simples, como descrito na seção 3, ou seja, efetuando uma ou mais operações de entrada de dados, seguida do processamento dos dados obtidos, e sendo finalizadas por uma etapa de saída de dados, a qual produz os resultados apresentados ao usuário ou armazenados para uso posterior. Essa é a estratégia **E-P-S** (*Entrada-Processamento-Saída*).

É comum que, para resolver um problema, seja seguida uma sequência de operações na qual são realizados cálculos sobre valores específicos ou são efetuados outros tipos de manipulação sobre os dados disponíveis. Na maioria das vezes não é possível iniciar esta sequência de operações sem que alguns dos dados requeridos para os cálculos sejam conhecidos.

Por isso é bastante típico que os programas sejam iniciados por uma etapa onde estes dados iniciais sejam solicitados ao usuário do programa. Esta é a etapa de *entrada de dados*, na qual os dados requeridos pelo programa são considerados como dados de entrada.

Já a solução do problema, ou seja, os resultados desejados, que são produzidos pela execução do programa, são entendidos como *dados de saída*, os quais são apresentados (ou exibidos) ao usuário após a realização da sequência de

operações que o programa realiza a partir dos dados de entrada. Ou seja, a computação, realizada a partir dos dados de entrada para produzir os dados de saída, é o que chamamos de *processamento de dados*.

Alguns problemas-exemplo pode auxiliar na compreensão e aplicação desta estratégia.

4.2.1 Problema I: temperatura média e amplitude térmica

Ao longo dos dias, é absolutamente comum a ocorrência de variações na temperatura. Estas variações são decorrentes de muitos fatores, como horário, localidade geográfica, estação do ano, fenômenos climáticos e outros. A Figura 4.1 mostra um gráfico de temperaturas para a cidade de São Paulo, no período de julho/20 a junho/21². A linha vermelha central representa a temperatura média de cada mês, o que permite observar que fevereiro foi o mês mais quente do ano, com temperatura média de 22.5°C, enquanto julho foi o mês mais frio do ano, com uma temperatura média de 16.2°C.

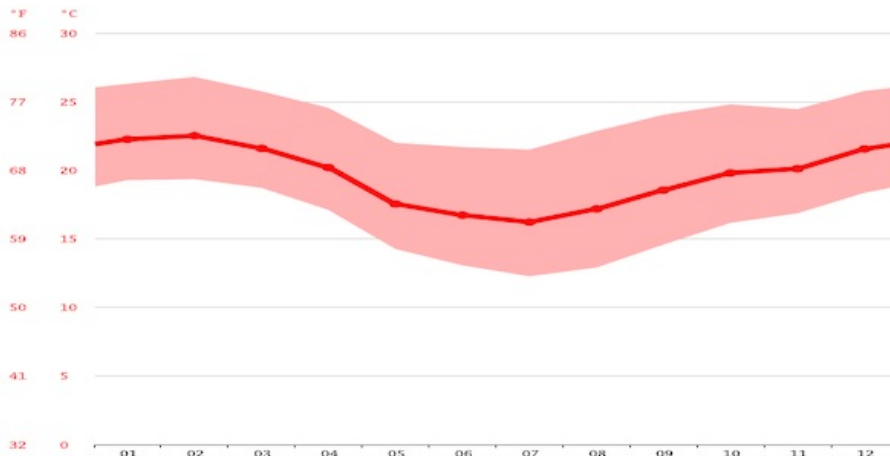


Figure 4.1: Temperaturas em São Paulo [julho/20-junho/21]

O problema que temos é:

Como escrever um programa para determinar a temperatura média e amplitude térmica de uma localidade em um período determinado?

Aqui existe uma questão que precisa ser enfatizada: para que seja possível escrever um programa para resolver um problema, precisamos, *primeiro*, resolver

²Fonte: <https://pt.climate-data.org/americas-do-sul/brasil/sao-paulo/sao-paulo-655/>, acesso em 30/06/2021.

este problema, ao menos esquematicamente, ou seja, precisamos estabelecer *como* são realizadas as operações e os cálculos necessários para sua solução. Só então, saberemos quais são os dados necessários para a entrada, isto é, os dados que constituem os pré-requisitos da solução.

Em resumo, *antes* de construirmos um programa, devemos ser capazes de solucionar o problema conceitualmente, de maneira a organizar este programa para que realize as operações e os cálculos que foram determinados para a solução deste problema particular.

Para o problema em questão, vamos considerar que t_{\max} é a temperatura máxima e t_{\min} é a temperatura mínima, observadas numa localidade específica e em um período determinado (dia, semana, mês, ano etc.), a temperatura média é a média aritmética simples de t_{\max} e t_{\min} , ou seja:

$$t_{\text{média}} = (t_{\max} + t_{\min}) / 2$$

A faixa ao redor da linha vermelha, que pode ser vista na Figura 4.1, representa a *amplitude térmica* ou *oscilação térmica* observada mês a mês. A *amplitude térmica* é a diferença existente entre os valores máximos e mínimos de temperatura, observada durante um período de tempo específico, em um determinado local. A amplitude térmica AT é calculada como:

$$AT = t_{\max} - t_{\min}$$

Então, para calcular a temperatura média e a amplitude térmica são requeridas as temperaturas mínima t_{\min} e máxima t_{\max} . A estratégia **E-P-S** é adequada aqui, pois:

1. Devemos obter as temperaturas (t_{\min} e t_{\max}) da localidade e período desejados (entrada);
2. Efetuamos os cálculos da temperatura média e da amplitude térmica (processamento);
3. Apresentamos os resultados obtidos (saída).

Assim, o problema é dividido em partes, simplificando sua solução.

4.2.1.1 Entrada

A entrada de dados deste problema requer obter as as temperaturas mínima t_{\min} e máxima t_{\max} da da localidade e período desejados. Antes de obter estes dados, é necessário considerar de que tipo são. Medidas de temperatura são grandezas reais, pois seus valores podem conter uma parte inteira e outra fracionária, assim, as medidas de temperatura mínima e máxima devem ser tratadas como valores reais, do tipo `float` no Python.

Toda vez que se realiza uma entrada de dados, devem ser exibidas mensagens de orientação para o usuário, dirigindo a leitura dos respectivos valores. O fragmento que segue pode efetuar a entrada da medida da temperatura mínima.

```
# Entrada da temperatura mínima
temp_minima = float(
    input('Digite a temperatura mínima [medida em C]: '))
```

```
>>> Digite a temperatura mínima [medida em C]: 19.4
```

A função *built-in* `input()` foi usada para exibir o *prompt* (mensagem de orientação ao usuário) e efetuar a leitura de um dado, enquanto a função *built-in* `float()` efetua a conversão do texto digitado pelo usuário e lido por `input()` em um número real (do tipo `float`) correspondente à temperatura mínima. Caberá ao usuário digitar valores válidos.

O mesmo pode ser feito para entrada da medida da temperatura máxima.

```
# Entrada da temperatura máxima
temp_maxima = float(
    input('Digite a temperatura máxima [medida em C]: '))
```

```
>>> Digite a temperatura máxima [medida em C]: 26.8
```

Com isso, o programa passa a dispor das duas medidas necessárias para calcular a temperatura média e a amplitude térmica. Cabe ao usuário fornecer tais dados, relativos à uma mesma localidade, dentro do período de tempo desejado.

4.2.1.2 Processamento

O cálculo da temperatura média e da amplitude térmica deve ser feito como indicado no começo desta seção, ou seja:

$$tmédia = (tmax + tmin) / 2$$
$$AT = tmax - tmin$$

Como a sintaxe das expressões algébricas do Python (e de praticamente todas as linguagens de programação) é a mesma da Matemática, estas fórmulas matemáticas podem ser usadas diretamente, apenas com adequação dos nomes das variáveis, como mostra o fragmento que segue, onde `tmédia`, `tmax` e `tmin` tornam-se, respectivamente, `temp_media`, `temp_maxima` e `temp_minima`.

```
# Cálculo da temperatura média
temp_media = (temp_maxima + temp_minima) / 2
```

O mesmo é feito no próximo fragmento onde, adicionalmente, AT (amplitude térmica) é substituída por `amp_termica`.

```
# Cálculo da amplitude térmica
amp_termica = temp_maxima - temp_minima
```

Como estes são os únicos cálculos necessários, a etapa do processamento está concluída, possibilitando que os resultados sejam apresentados ao usuário.

4.2.1.3 Saída

A etapa de saída de um programa deve apresentar os resultados obtidos por meio do processamento realizado, pois estes constituem a resposta do problema proposto. Neste exemplo, devem ser exibidos ao usuário os valores correspondentes à temperatura média e à amplitude térmica de uma localidade e período arbitrários. O uso da função *built-in* `print()` e da função `format()` permitem realizar esta tarefa.

```
# Exibição dos dados de entrada
print('Temperaturas mínima = {:.1f}C e máxima = {:.1f}C'
      .format(temp_minima, temp_maxima))
# Exibição dos dados de saída
print('Temperaturas média = {:.1f}C'.format(temp_media))
print('Amplitude Térmica = {:.1f}C'.format(amp_termica))
```

```
>>> Temperaturas mínima = 19.4C e máxima = 26.8C
>>> Temperaturas média = 23.1C
>>> Amplitude Térmica = 7.4C
```

Foram realizadas três chamadas à função `print()`: para exibir os dados de entrada (temperaturas mínima e máxima); para exibir o resultado da temperatura média; e finalmente para imprimir a amplitude térmica. A repetição dos dados de entrada foi feita apenas para clarificar a origem do resultado, e deve ser considerada caso a caso. Um dos comandos era longo e, assim, foi apresentado em duas linhas, sendo a divisão feita no operador `.` (seletor) e observando a indentação da linha adicional com 4 espaços (padrão do Python). Observe também o uso do marcador com especificador `{:.1f}` para formatar os números reais (tipo `float`) com apenas uma casa decimal. A unidade de temperatura foi inclusa na *string* de formatação logo após os marcadores de formatação.

4.2.1.4 Programa Completo

Os trechos de código desenvolvidos para realizar as etapas de entrada, processamento e saída podem ser organizados em um único *script* Python, possibilitando sua execução conveniente como um programa, o qual soluciona o problema proposto de cálculo da temperatura média e da amplitude térmica de uma localidade e período arbitrários. Esse *script* pode ser salvo no arquivo `amplitude_termica.py`.

Exemplo 4.1: `amplitude_termica.py`

```
print('Temperatura Média e Amplitude Térmica')

# Entrada de dados
temp_minima = float(
    input('Digite a temperatura mínima [medida em C]: '))
temp_maxima = float(
    input('Digite a temperatura máxima [medida em C]: '))

# Processamento
temp_media = (temp_maxima + temp_minima) / 2
amp_termica = temp_maxima - temp_minima

# Saída de dados
print('Temperaturas mínima = {:.1f}C e máxima = {:.1f}C'
      .format(temp_minima, temp_maxima))
print('Temperaturas média = {:.1f}C'.format(temp_media))
print('Amplitude Térmica = {:.1f}C'.format(amp_termica))
```

Esse programa pode ser salvo em um arquivo denominado, por exemplo, `amplitude_termica.py`. Para executá-lo em um computador dotado de uma instalação funcional de Python, basta fornecer o comando:

```
$ python amplitude_termica.py
```

Sua execução produz resultados como os ilustrados na Figura 4.2.

Embora simples, este programa cumpre com seus objetivos e sistematiza a obtenção da temperatura média e da amplitude térmica, auxiliando seus usuários, que não precisam saber como são realizados os cálculos necessários, mas apenas como usar o programa.

4.2.2 Problema II: caixas de papelão

O transporte de mercadorias, desde os primórdios do comércio, é uma atividade extremamente importante, pois torna disponíveis produtos, de praticamente

```
Anaconda Prompt (R-MINI~1)
$ python amplitude_termica.py
Temperatura Média e Amplitude Térmica
Digite a temperatura mínima [medida em C]: 19.4
Digite a temperatura máxima [medida em C]: 26.8
Temperaturas mínima = 19.4C e máxima = 26.8C
Temperaturas média = 23.1C
Amplitude Térmica = 7.4C
```

Figure 4.2: Execução de `amplitude_termica.py`

qualquer tipo, nos locais onde existe demanda, mas não são cultivados ou fabricados. Para facilitar o transporte é bastante comum que sejam utilizadas embalagens de papelão, pois são leves, relativamente resistentes e podem ser fabricadas para acomodar adequadamente produtos com características muito distintas.

O formato mais comum para as embalagens de papelão é de caixa, ou seja, de um paralelepípedo reto como na Figura 4.3, para o qual existem duas medidas importantes: sua área total e seu volume. Como o papelão é fornecido em folhas ou rolos que serão cortados e dobrados para fazer uma caixa, a área total ou superfície do paralelepípedo indica quanto papelão será necessário. Já o volume permite saber quanto cabe em uma caixa de papelão, facilitando a escolha da caixa adequada.

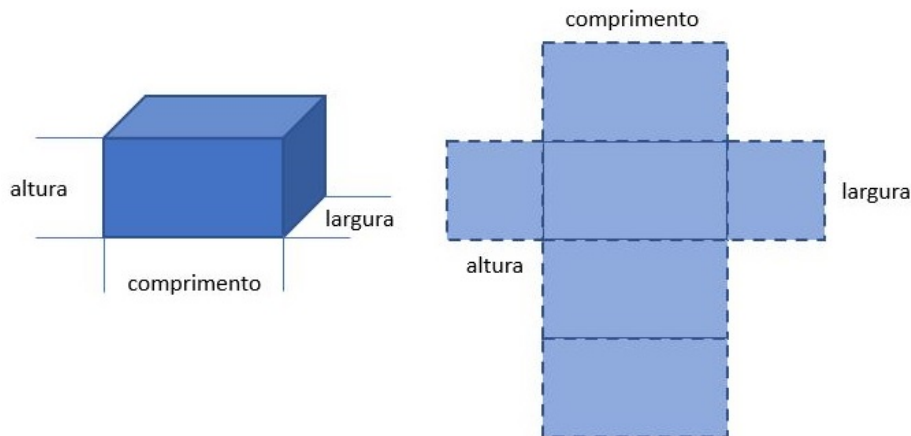


Figure 4.3: Paralelepípedo e sua planificação como caixa

O problema que se apresenta aqui é:

Como construir um programa que determine a área total e o volume de uma caixa retangular?

Uma caixa retangular é um paralelepípedo reto (GOUVEIA, 2021c), uma figura geométrica sólida tridimensional, cujas dimensões são conhecidas como altura, largura e comprimento. A área ou superfície total desta figura é a soma das área de suas 6 faces, mas como as faces são paralelas, existem 3 pares de faces distintas, de modo que:

$$\text{áreatotal} = 2.(\text{altura.largura} + \text{altura.comprimento} + \text{largura.comprimento})$$

O volume do paralelepípedo reto é calculado pelo produto de suas dimensões:

$$\text{volume} = \text{altura.largura.comprimento}$$

Então, para seja possível determinar a área total e o volume de uma caixa retangular de papelão, precisamos conhecer quais são suas dimensões (altura, largura e comprimento), que são um pré-requisito para realização destes cálculos. A estratégia **E-P-S** é adequada aqui, pois:

1. Devemos obter as dimensões (altura, largura e comprimento) da caixa (entrada);
2. Efetuamos os cálculos da área total e volume da caixa (processamento);
3. Apresentamos os resultados obtidos (saída).

Como antes, para saber quais são os dados necessários para a entrada, precisamos estabelecer *como* são realizados os cálculos que desejamos que o programa seja capaz de realizar. Assim *antes* de construir um programa, devemos ser capazes de resolver o problema, ao menos esquematicamente, como feito anteriormente.

4.2.2.1 Entrada

Na entrada de dados deste problema devemos obter as dimensões da caixa retangular, que são sua altura, largura e comprimento. Para obter estes dados, é necessário considerar, antes, de que tipo são. Medidas de comprimento são grandezas reais, pois seus valores podem conter uma parte inteira e outra fracionária, assim, as medidas altura, largura e comprimento devem ser tratadas como valores reais, do tipo `float` no Python.

A entrada destas medidas deve combinar mensagens de orientação para o usuário e a leitura dos respectivos valores. O fragmento que segue faz uso das funções `input()` e `float()` para efetuar a entrada da medida da altura.

```
# Entrada da medida da altura
altura = float(input('Digite a altura [medida em cm]: '))
```

```
>>> Digite a altura [medida em cm]: 5.5
```

O mesmo pode ser feito para entrada das medidas de largura e de comprimento.

```
# Entrada da medida da largura
largura = float(input('Digite a largura [medida em cm]: '))
# Entrada da medida do comprimento
comprimento = float(
    input('Digite o comprimento [medida em cm]: '))
```

```
>>> Digite a largura [medida em cm]: 10
>>> Digite o comprimento [medida em cm]: 15
```

Com isso, o programa passa a dispor das três medidas necessárias para calcular a área total e o volume de uma caixa retangular de papelão.

4.2.2.2 Processamento

O cálculo da área total e do volume da caixa retangular de papelão deve ser feito como indicado no começo desta seção, ou seja:

```
áreatotal = 2.(altura.largura + altura.comprimento + largura.comprimento)
volume = altura.largura.comprimento
```

Estas fórmulas matemáticas precisam de (pouca) adaptação para seguirem a sintaxe das expressões algébricas do Python, que é a mesma em praticamente todas as linguagens de programação. No fragmento que segue percebemos que `áreatotal` torna-se `area_total`, enquanto o operador de multiplicação da matemática `.` é substituído por `*`.

```
# Cálculo da área total
area_total = 2 * (altura * largura + altura * comprimento
    + largura * comprimento)
```

O mesmo é feito para `volume` no próximo fragmento.

```
# Cálculo do volume
volume = altura * largura * comprimento
```

Como não existem outros cálculos a serem realizados, o processamento está concluído e os resultados precisam ser apresentados ao usuário.

4.2.2.3 Saída

A etapa de saída de um programa deve apresentar os resultados obtidos que constituem a resposta ao problema que deveria ser resolvido. Neste exemplo, devem ser exibidos ao usuário os valores correspondentes à área total e ao volume da caixa retangular. O uso da função *built-in* `print()` e de *f-strings* permite obter uma saída adequada.

```
# Exibição dos dados de entrada
print(f'Dimensões:\n' \
      f'\taltura = {altura:.2f} cm\n' \
      f'\tlargura = {largura:.2f} cm\n' \
      f'\tcomprimento = {comprimento:.2f} cm')
# Exibição dos dados dos resultados
print(f'Resultados:\n' \
      f'\tárea total = {area_total:.3f} cm2\n' \
      f'\tvolume = {volume:.3f} cm3')

```

```
>>> Dimensões:
>>>     altura = 5.50 cm
>>>     largura = 10.00 cm
>>>     comprimento = 15.00 cm
>>> Resultados:
>>>     área total = 575.000 cm2
>>>     volume = 825.000 cm3

```

Em vez de realizar uma chamada da função `print()` para exibir cada dado de entrada e de saída, preferiu-se efetuar duas chamadas, uma para exibir as dimensões fornecidas para a caixa e outra para imprimir os resultados obtidos. Como as *f-string* necessárias seriam muito longas, utilizou-se uma barra invertida `\` para dividi-las em várias linhas, coincidindo com o *layout* desejado para a saída. Note o uso dos caracteres especiais de tabulação `\t` e de quebra de linha `\n` para o melhor espaçamento e divisão das linhas da mensagem.

Os valores reais das variáveis de entrada (`altura`, `largura` e `comprimento`) e de saída (`area_total` e `volume`) foram formatados para apresentar um número fixo de casas decimais, acompanhados de suas respectivas unidades (cm, cm² e cm³)

4.2.2.4 Programa Completo

Os trechos de código requeridos para realizar as etapas de entrada, processamento e saída podem ser organizados em um único *script* Python, salvo num arquivo como `caixa_de_papelao.py`, possibilitando sua execução conveniente

como um programa capaz de solucionar o problema proposto, do cálculo da área total e do volume de uma caixa de papelão retangular.

Exemplo 4.2: caixa_de_papelao.py

```
print('Caixa de Papelão Retangular::área e volume')

# Entrada de dados
altura = float(input('Digite a altura [medida em cm]: '))
largura = float(input('Digite a largura [medida em cm]: '))
comprimento = float(input('Digite o comprimento [medida em cm]: '))

# Processamento
area_total = 2 * (altura * largura + altura * comprimento
                 + largura * comprimento)
volume = altura * largura * comprimento

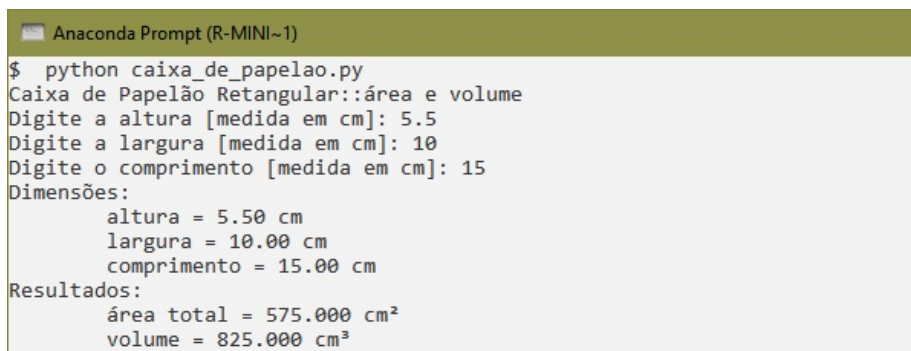
# Saída de dados
print(f'Dimensões:\n' \
      f'\taltura = {altura:.2f} cm\n' \
      f'\tlargura = {largura:.2f} cm\n' \
      f'\tcomprimento = {comprimento:.2f} cm')
print(f'Resultados:\n' \
      f'\tárea total = {area_total:.3f} cm2\n' \
      f'\tvolume = {volume:.3f} cm3')
```

Esse programa pode ser salvo em um arquivo denominado, por exemplo, caixa_de_papelao.py. Para executá-lo em um computador dotado de uma instalação funcional de Python, basta fornecer o comando:

```
$ python caixa_de_papelao.py
```

Sua execução produz resultados como os ilustrados na Figura 4.4.

Embora simples, este programa cumpre com seus objetivos e sistematiza a obtenção da área total e do volume de caixas de papelão, auxiliando seus usuários.



```
Anaconda Prompt (R-MINI~1)
$ python caixa_de_papelao.py
Caixa de Papelão Retangular::área e volume
Digite a altura [medida em cm]: 5.5
Digite a largura [medida em cm]: 10
Digite o comprimento [medida em cm]: 15
Dimensões:
    altura = 5.50 cm
    largura = 10.00 cm
    comprimento = 15.00 cm
Resultados:
    área total = 575.000 cm²
    volume = 825.000 cm³
```

Figure 4.4: Execução de caixa_de_papelao.py

Chapter 5

Repetição

Sabemos que a programação de computadores exige o domínio de várias habilidades, como a computação e as operações de entrada e saída, cujas instruções podem ser combinadas, por meio da sequenciação, para organizar programas capazes de resolver inúmeros problemas. No entanto, existem muitas situações onde a mesma instrução, instruções semelhantes ou um conjunto de instruções, se repetem, o que demanda uma nova maneira para sua programação.

A *repetição* trata da execução repetida de um conjunto de passos com base numa condição ou contagem. É uma habilidade que consiste em identificar as instruções cuja execução deve ocorrer mais de uma vez, o que inclui determinar o número de vezes que tais instruções serão executadas ou a condição que exige sua repetição.

Como veremos, o emprego da repetição permite reduzir a quantidade de instruções que são escritas para resolver um problema, o que simplifica o trabalho de programação e aumenta a produtividade do desenvolvedor, além de conferir maior flexibilidade às soluções criadas. A repetição é frequentemente utilizada para realizar:

- a contagem;
- a agregação;
- o atingimento de condição;
- o percurso em conjunto de elementos.

Como nas demais linguagens de programação, o Python oferece dois tipos de repetição que serão apresentadas nas seções que seguem: a *condicional* (seção 5.1) e a *automática* (seção 5.2).

5.1 Repetição Condicional

A repetição *condicional* possibilita que uma instrução ou várias instruções sejam executadas diversas vezes, enquanto uma condição permanece verdadeira. A diretiva Python que permite esse comportamento é o **while**, cuja sintaxe é:

```
while <condição>:  
    <comando(s)>
```

A *condição* deve ser uma expressão lógica, isto é, cuja avaliação produz um valor de tipo `bool`, de maneira que, se resulta **True**, as instruções associadas são executadas; mas quando resulta **False**, finaliza o comando de repetição, ou seja, interrompe a repetição. Após a condição é obrigatório o uso do dois-pontos : para finalizar a diretiva **while**.

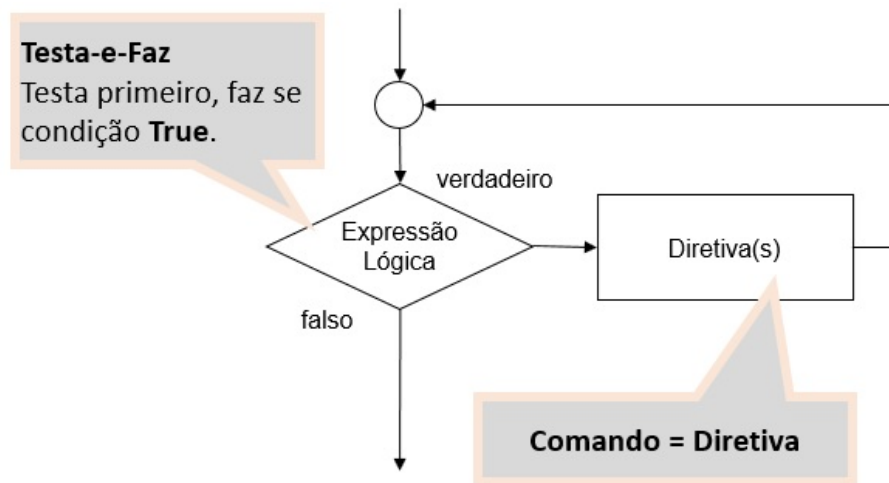
Podem ser associados um ou mais comandos à diretiva **while**, desde que estejam indentados¹ de quatro espaços à direita, que é o padrão do Python. É comum que algum dos comandos associados modifique o conteúdo de alguma variável presente na condição, fazendo que, em algum momento, seja avaliada logicamente como falsa.

O comportamento desta diretiva é ilustrado no fluxograma da Figura 5.1. A execução desta diretiva se inicia com a avaliação da expressão lógica da condição, ou seja, seu teste. Se a condição é avaliada como **True**, os comandos associados são executados, sequencialmente, uma vez; se condição é avaliada como **False**, a diretiva **while** é finalizada. Cada execução dos comandos associados é considerada uma *iteração*².

Ao final da execução dos comandos associados, ocorre um *desvio*, de maneira que a condição é avaliada novamente, repetindo o comportamento descrito. Ou seja: *enquanto* a condição é avaliada como **True**, os comandos associados são executados. Ainda na Figura 5.1, é possível notar um caminho fechado quando a condição é verdadeira, razão pela qual as diretivas de repetição são conhecidas como *laços* ou *loops*.

¹ *Indentação* é o termo que se refere ao espaço de abertura de de um parágrafo, ou seja, entre a margem e o começo da da escrita. A indentação proporciona maior legibilidade ao código, pois indica a estrutura de associação dos comandos em um programa de computador.

² Segundo o Dicio (Dicionário On-Line de Português), *iteração* é a ação de iterar ou de repetir. Na computação utiliza-se este termo para designar cada uma das execuções dos comandos ou instruções associadas à um laço de repetição que, assim, produz várias *iterações* ou *repetições* (<https://www.dicio.com.br/iteracao/>).

Figure 5.1: Fluxograma da diretiva `while`

A construção da *condição* empregada pela diretiva `while` requer muita atenção, pois se avaliada inicialmente como falsa, nenhum dos comandos associados é executado e nenhuma ação é produzida.



Se a avaliação da *condição* indicada no `while` não se tornar falsa, o laço nunca será encerrado, ou seja, teremos um laço infinito. Como o programa não prossegue, dizemos que ficou *pendurado* (*hang-up*), o que exige sua interrupção forçada e causa perda de dados, ou seja, é um erro grave de lógica.

As seções que seguem mostram como a diretiva `while` pode ser utilizada nas aplicações da repetição.

5.1.1 Contagem com `while`

A *contagem* é uma das aplicações mais básicas da repetição, pois permite controlar o número de vezes que uma operação definida pelo programador é realizada pelo programa. Exemplos de operações controladas pela contagem são a exibição de mensagens, a entrada de dados, as tentativas em um jogo etc.

O fragmento que segue mostra uma contagem simples de zero a quatro com o uso da diretiva `while`.

```
contador = 0
while contador <= 4:
    print(contador)
    contador += 1 # equivale à contador = contador + 1
```

```
>>> 0
>>> 1
>>> 2
>>> 3
>>> 4
```

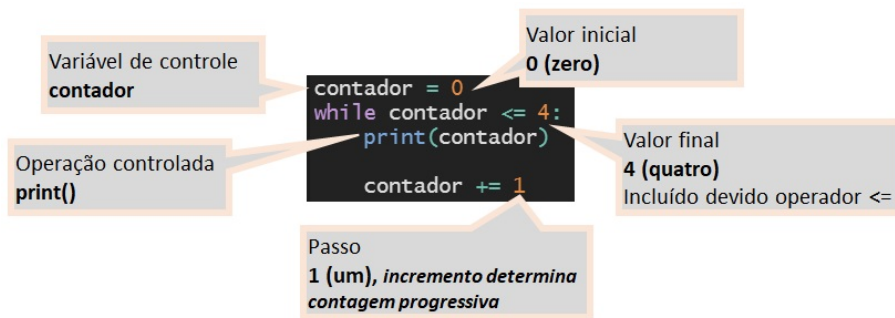
O funcionamento deste fragmento é simples. Começa com a inicialização da variável `contador` com o valor 0; o laço de repetição `while` avalia a condição `contador <= 4`, que resulta `True`, pois `contador` tem valor zero, executando as instruções associadas; a função `print()` exibe o valor de `contador`; então a variável `contador` recebe um incremento de 1 (e passa ter o valor 1); ocorre o desvio para que `while` reavalie a condição.

Assim, as instruções associadas ao `while` serão executadas *enquanto* a condição especificada for avaliada como verdadeira, ou seja, a partir de `contador = 0`, serão executadas para `contador = 0, 1, 2, 3, 4`, num total de cinco *iterações*. Quando a variável `contador` contém o valor 4, ao receber o incremento de uma unidade, passa a valor 5, de maneira que a avaliação da condição `contador <= 4` resulta em `False`, encerrando a execução da diretiva `while`.

Embora simples, toda contagem envolve cinco elementos que precisam ser identificados, como mostra a Figura 5.2:

- Uma *variável de controle* que armazena o valor corrente da contagem;
- Um *valor inicial* de onde parte a contagem que será realizada;
- Um *valor final* que indica o limite para o qual a contagem será encerrada;
- Um *passo* que especifica o incremento ou decremento usado na contagem;
- e
- A *operação* que será controlada pela contagem.

O fragmento que segue mostra uma contagem onde a variável de controle é `contador`, que recebe o valor inicial 10; o valor final será 2, pois a condição da diretiva `while` é `contador > 0`; e o passo negativo `-2` estabelece uma contagem regressiva de dois em dois. A operação controlada pela contagem é a impressão do valor da variável de controle com uso da função `print()`.

Figure 5.2: Elementos da contagem no `while`

```
contador = 10
while contador > 0:
    print(contador)
    contador = contador - 2 # equivale à contador -= 2
```

```
>>> 10
>>> 8
>>> 6
>>> 4
>>> 2
```

O uso da repetição é sempre vantajoso quando comparada à repetição direta de código. O fragmento que segue executa uma mesma operação quatro vezes, ou seja, efetua a impressão de uma mensagem com uso repetido da função `print()`.

```
print('Python')
print('Python')
print('Python')
print('Python')
```

```
>>> Python
>>> Python
>>> Python
>>> Python
```

Já no fragmento que segue, operação em si não se repete, mas é adicionado o código da contagem referente ao controle da repetição, produzindo o mesmo resultado.

```
contador = 0
while contador < 4:
    print('Python')
    contador += 1 # equivale à contador = contador + 1
```

```
>>> Python
>>> Python
>>> Python
>>> Python
```

Embora não seja óbvio, o fragmento que emprega o laço de repetição é mais vantajoso. Apesar da quantidade de código ser semelhante e da simplicidade da repetição direta, observe que o código que emprega a repetição com a diretiva **while** é praticamente o mesmo 1, 4, 10 ou 1000 execuções da operação, pois apenas o limite é alterado. Ou seja, quanto maior o número de repetições ou maior é número de instruções repetidas, maior também é a vantagem no uso de laços de repetição.



O uso dos laços de repetição fará com que o programa-fonte, assim como o código executável do programa sejam menores, com consequente redução do espaço em disco e de memória necessários. Mas o número de instruções executadas não será reduzido, portanto o desempenho será semelhante ao do código construído sem laços de repetição.

Se, no fragmento anterior, o literal 4 é substituído por uma variável, então a quantidade de repetições pode ser determinada por uma expressão calculada *antes* da execução do laço ou mesmo informada pelo usuário por meio de uma operação de entrada de dados.

Segue um programa, cujo arquivo é **media.py**, capaz de calcular as médias de um número arbitrário de alunos, que evidencia a utilidade dos laços de repetição. O programa se inicia solicitando o número (inteiro) de alunos para os quais serão calculadas as médias. Um laço de repetição **while** efetua a contagem de 1 até o **numero_alunos** inclusive, solicitando duas notas reais **nota_1** e **nota_2**, com as quais é calculada e exibida a **media**.

Exemplo 5.1: media.py

```
print('Média dos Alunos')

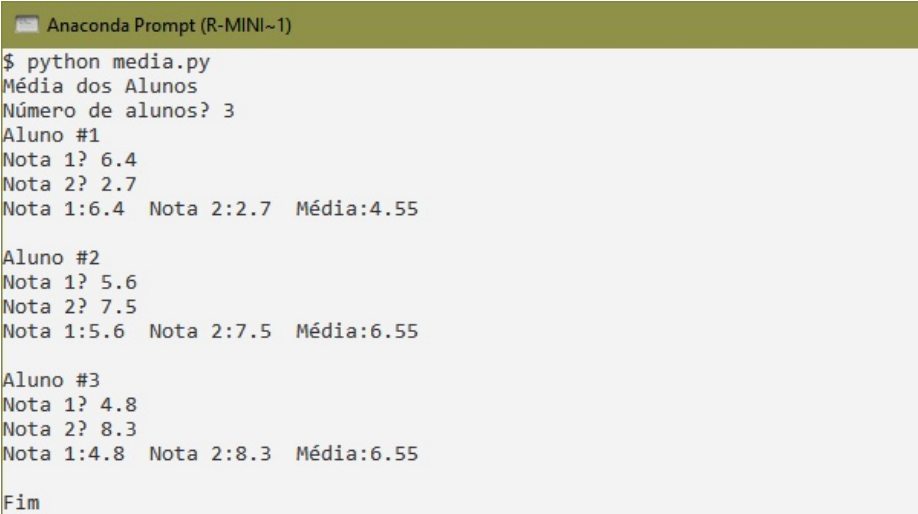
# Entrada de dados
```

```
numero_alunos = int(input('Número de alunos? '))

# Laço de Repetição
contador = 1
while contador <= numero_alunos:
    print(f'Aluno #{contador}')
    # lê notas
    nota_1 = float(input('Nota 1? '))
    nota_2 = float(input('Nota 2? '))
    # calcula média
    media = (nota_1 + nota_2) / 2
    # exibe média
    print(f'Nota 1:{nota_1:.1f}  Nota 2:{nota_2:.1f}  ' \
          f'Média:{media:.2f}\n')
    contador += 1 # incrementa contador

print('Fim')
```

Deve ser observado que, com exceção da diretiva `contador += 1`, todos os comandos associados ao `while` são destinados a realizar o cálculo da média de um aluno individual, identificado pelo número contido na variável `contador`, que permite o controle da repetição neste laço. O resultado deste programa pode ser visto na Figura 5.3.



```
Anaconda Prompt (R-MINI~1)
$ python media.py
Média dos Alunos
Número de alunos? 3
Aluno #1
Nota 1? 6.4
Nota 2? 2.7
Nota 1:6.4  Nota 2:2.7  Média:4.55

Aluno #2
Nota 1? 5.6
Nota 2? 7.5
Nota 1:5.6  Nota 2:7.5  Média:6.55

Aluno #3
Nota 1? 4.8
Nota 2? 8.3
Nota 1:4.8  Nota 2:8.3  Média:6.55

Fim
```

Figure 5.3: Execução de `media.py`

Vale observar que o laço de repetição `while` executa as etapas parciais de entrada, processamento e saída para cada aluno. Além disso, o mesmo código é

executado para um número variável de alunos, conferindo uma flexibilidade ao programa que não pode ser obtida com código de natureza estritamente sequencial (que teria que ser adaptado para cada número diferente de alunos).

5.1.2 Agregação com while

A *agregação* é outra operação frequentemente realizada nos programas, na qual um conjunto de dados é transformado em um resultado único, o *agregado*. Quando caixas são carregadas em um caminhão, é possível tanto contar o número de volumes, como somar o peso total embarcado, onde esta contagem e este somatório são exemplos de valores agregados, pois combinam um dado individual num resultado único.

Além de contagens e de somas (ou somatórios), outros exemplos de agregação são a determinação de médias, de produtos (ou produtórios), de valores máximos e mínimos.

O *script* que segue efetua duas agregações com os inteiros existentes entre 1 e 10: sua soma e seu produto, exibindo seus resultados. As variáveis `ini` e `fim` armazenam os limites da faixa de valores que se deseja processar. Este *script* é denominado `agregacao_while.py`.

Observe que as variáveis necessárias para conter as agregações devem ser inicializadas *antes* dos laços de repetição utilizados para realizar a agregação. Aqui `soma` é inicializada com zero e `produto` é inicializada com 1, respectivamente os elementos neutro da soma e da multiplicação. A variável `contagem` se destina ao controle da contagem dos valores inteiros entre `ini` e `fim`.

Exemplo 5.2: `agregacao.py`

```
ini = 1 # definição do início da agregação
fim = 10 # definição do final da agregação
print(f'Agregação [{ini}..{fim}]')

soma = 0 # variável para agregação de soma
produto = 1 # variável para agregação de produto

contador = ini # controle da contagem
while contador <= fim:
    soma += contador # soma no agregado
    produto *= contador # multiplica pelo agregado
    contador += 1 # incremento do contador

print(f'Soma = {soma}\nProduto = {produto}')
```

```
>>> Agregação [1..10]
```

```
>>> Soma = 55
>>> Produto = 3628800
```

As operações de agregação (a soma e o produto) são realizadas pelos comandos associados aos laços de repetição, pois devem ser aplicadas a cada um dos valores na faixa entre *ini* e *fim*. O incremento unitário do *contador* é realizado após as operações de agregação. A exibição dos resultados é realizada *após* os laços de repetição, ou seja, quando as operações de agregação já foram aplicadas a todos os dados, com eventual processamento adicional e exibição dos resultados finais.

Outras operações de agregação, como contagens, obtenção de mínimos e máximo, seguem a mesma estrutura, exigindo algum cuidado na seleção do valor de inicialização da variável destinada ao agregado.

No exemplo `media.py`, da seção anterior, são calculadas as médias individuais de um conjunto de alunos. A média geral da classe, uma informação adicional possível de ser extraída destes dados, é um outro exemplo de agregação, que pode ser obtida com as modificações introduzidas no exemplo que segue, denominado `media2.py`.

Exemplo 5.3: `media2.py`

```
print('Média dos Alunos')

# Entrada de dados
numero_alunos = int(input('Número de alunos? '))

# Laço de Repetição
contador = 1
soma_medias = 0 # variável para agregação (soma)
while contador <= numero_alunos:
    print(f'Aluno #{contador}')
    # lê notas
    nota_1 = float(input('Nota 1? '))
    nota_2 = float(input('Nota 2? '))
    # calcula média
    media = (nota_1 + nota_2) / 2
    # agrega média individual a soma das médias
    soma_medias += media
    # exibe média
    print(f'Nota 1:{nota_1:.1f}  Nota 2:{nota_2:.1f}  ' \
          f'Média:{media:.2f}\n')
    contador += 1 # incrementa contador

# calcula e exibe média geral
media_geral = soma_medias / numero_alunos
```

```
print(f'Média geral:{media_geral:.2f}\n')  
  
print('Fim')
```

São três as modificações realizadas para o cômputo da média geral:

1. Criação da variável `soma_medias`, inicializada com valor 0, o valor neutro da soma, para conter a agregação (da soma) das médias.

```
soma_medias = 0
```

2. Soma da médias individuais à variável `soma_medias`, realizando sua agregação.

```
soma_medias += media
```

3. Cálculo e exibição da média geral.

```
media_geral = soma_medias / numero_alunos  
print(f'Média geral:{media_geral:.2f}\n')
```

A Figura 5.4 exibe os resultados produzidos pelo programa `media2.py`.

Neste exemplo, é necessário realizar uma operação adicional para obtenção da média, que é divisão do agregado da soma pela número de elementos somados. Estas operações de finalização, assim como a exibição dos resultados finais é realizada *após* os laços de repetição, ou seja, quando as operações de agregação já foram aplicadas a todos os dados.

5.1.3 Atingimento de condição com `while`

Outra possibilidade do uso de laços de repetição condicionais é para que uma condição específica seja atingida, ou seja, para permitir a execução de um conjunto de operações até que um objetivo seja alcançado.

Considere a necessidade de um programa em obter um número, informado pelo usuário, que esteja dentro de uma faixa específica de valores. Por exemplo, a obtenção de um número entre 1 e 5, ou seja, um valor inteiro que pertence ao conjunto (1, 2, 3, 4, 5). O fragmento que segue é capaz de realizar a entrada necessária.



```
Anaconda Prompt (R-MINI~1)
$ python media2.py
Média dos Alunos
Número de alunos? 3
Aluno #1
Nota 1? 6.4
Nota 2? 2.7
Nota 1:6.4 Nota 2:2.7 Média:4.55

Aluno #2
Nota 1? 5.6
Nota 2? 7.5
Nota 1:5.6 Nota 2:7.5 Média:6.55

Aluno #3
Nota 1? 4.8
Nota 2? 8.3
Nota 1:4.8 Nota 2:8.3 Média:6.55

Média geral:5.88
Fim
```

Figure 5.4: Execução de `media2.py`

```
minimo = 1 # valor mínimo da faixa
maximo = 5 # valor máximo da faixa
valor = 0 # valor fornecido pelo usuário
while valor < minimo or valor > maximo:
    valor = int(input(
        f'Digite um valor entre {minimo} e {maximo}: '))
print('Valor obtido na faixa:', valor)
```

No início do fragmento, observa-se três variáveis: `minimo` e `maximo`, que contém os respectivos valores mínimo e máximo da faixa desejada; e `valor`, que conterá o valor a ser fornecido pelo usuário, mas é inicializada com um valor *fora* da faixa desejada.

O laço de repetição `while` tem como condição uma expressão lógica que determina se o conteúdo da variável `valor` está ou não na faixa de valores desejada, ou seja, verifica se `valor` é menor do que o valor mínimo *ou* se é maior do o máximo. Quando uma destas duas condições ocorre, o resultado da expressão lógica é `True`, indicando que o conteúdo de `valor` está *fora* da faixa desejada, permitindo executar a única operação associada ao laço, que efetua a entrada de dados.

Como a variável `valor` foi inicializada com conteúdo *fora* da faixa de valores, a condição indicada no laço de repetição é inicialmente satisfeita e uma operação de leitura efetuada, modificando o conteúdo de `valor`. Como não existem outros

comandos associados, a condição do laço é avaliada novamente.

A realização da operação de leitura modifica o conteúdo de variável valor, de maneira, que a condição é reavaliada. Assim, *enquanto* o conteúdo de `valor` estiver fora da faixa desejada, a entrada de dados será repetida, mas quando o objetivo é atingido (valor dentro da faixa), o laço de repetição é encerrado.



O uso da repetição condicional para atingimento de objetivo exige atenção para construção da expressão lógica, que deve garantir a execução das instruções associadas enquanto a condição desejada não é atingida e, ao mesmo tempo, seu encerramento quando o objetivo é alcançado.

5.1.4 Percurso com `while`

É bastante frequente que os programa manipulem muitos dados, de tipos e significados diferentes. No entanto, o uso de variáveis simples, tal como feito até o momento, não é adequado para armazenar, organizadamente, a grande variedade e quantidade de dados requeridas pelos programas. Nesta situação, são necessárias *estruturas de dados* mais apropriadas, tais como as listas, as tuplas, os conjuntos e os dicionários, disponíveis no Python.

Uma estrutura de dados é capaz de conter um número variável de dados, tratados como seus elementos, e, como será visto mais à frente, sempre oferece uma maneira para acessar individualmente cada um dos dados que contém. É comum que um programa realize o acesso à todos os elementos de uma estrutura de dados, o que é chamado de *percurso* sobre a estrutura.

Segue a declaração de uma *lista* em Python, uma estrutura de dados capaz de conter um número variável de elementos de qualquer tipo. No caso, a lista é denominada `copas` e nela são armazenados alguns valores inteiros, indicados como uma sequência de valores, separados por vírgulas, delimitados por colchetes:

```
copas = [1958, 1962, 1970, 1994, 2002]
```

As listas são indexáveis, ou seja, cada um de seus elementos pode ser acessado por meio de um índice inteiro, onde 0 corresponde ao primeiro elemento, 1 ao segundo elemento e assim por diante. Além disso, a função *built-in* `len()` permite determinar o comprimento (ou tamanho) de uma lista, isto é, seu número de elementos, de modo sua aplicação na lista `copas` retorna o valor 5, pois esta estrutura contém cinco valores.

Com isso em mente, a realização de uma contagem de 0 até o comprimento da lista permite recuperar, individualmente cada elemento desta lista, como segue:


```
indice = 0
while indice < len(copas):
    print(indice, ': ', copas[indice])
    indice += 1
```

```
>>> 0 : 1958
>>> 1 : 1962
>>> 2 : 1970
>>> 3 : 1994
>>> 4 : 2002
```

Vale destacar que a lista `copas` tem cinco elementos, portanto seu comprimento tem igual valor, e os índices válidos para acessar seus elementos são 0, 1, 2, 3, e 4, valores menores do que seu comprimento, como indicado na condição do laço de repetição `while`.

5.2 Repetição Automática

A diretiva `for` oferece um mecanismo de repetição *automática*, que permite a execução de uma instrução ou várias instruções para cada elemento de um sequência de valores. A sintaxe desta diretiva é:

```
for <var> in <sequência>:
    <comando(s)>
```

A variável de controle `var`, que pode ser usada nos comandos associados ao `for`, recebe todos os elementos da *sequência*, um de cada vez, executando os comandos associados para cada valor. A variável de controle é separada da sequência por meio da palavra reservada `in`. Além disso a diretiva `for` deve ser finalizada por um dois-pontos `:`, também obrigatório.

A *sequência* deve ser um objeto *iterable* do Python, ou seja, um objeto de qualquer tipo, mas que suporta o *percurso* por seus elementos, de maneira que, um a um, possam ser recuperados e aplicados à variável de controle `var`.

Também deve ser observado que os comandos associados devem ser indentados com quatro espaços à direita, o padrão do Python, para que sejam executados a cada iteração do laço de repetição automática.

O comportamento desta diretiva é ilustrado no fluxograma da Figura 5.5. A execução desta diretiva se inicia com a obtenção do objeto *iterable* indicado, ou seja, com o preparo da lista de valores que serão aplicados à variável de controle. Se existe valor disponível na lista, a variável de controle `var` recebe tal valor e os comandos associados são executados sequencialmente. Após a

execução destes comandos, ocorre um *desvio* para verificar a existência de outro valor na lista, repetindo-se este processo. Quando não existem mais valores na lista, a diretiva `for` é finalizada.

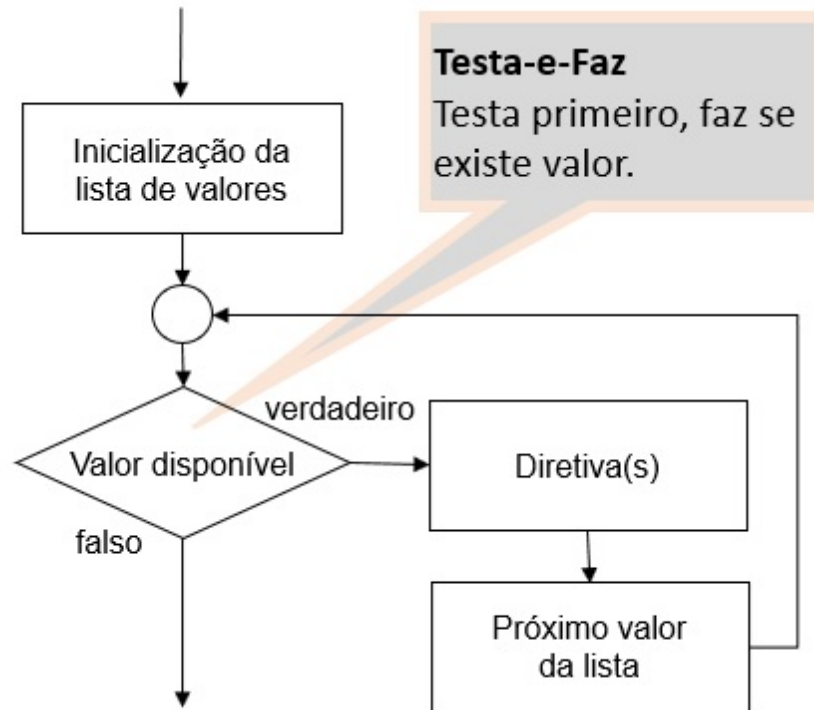


Figure 5.5: Fluxograma da diretiva `for`

Em resumo: *para cada* valor presente na sequência indicada pelo objeto `iterable` são executados os comandos associados. Na Figura 5.4 é possível observar um caminho fechado quando existem elementos disponíveis na sequência indicada, de maneira que a diretiva `for` também seja conhecida como um *laço* ou *loop*.



Caso a *sequência* indicada pela diretiva `for` esteja inicialmente vazia, nenhum dos comandos associados é executado e nenhuma ação é produzida.

As próximas seções mostram o uso da diretiva `for` nas várias aplicações da repetição.

5.2.1 Percurso com for

A diretiva `for` é capaz de percorrer, automaticamente, todos os elementos de objetos `iterable` do Python, como listas, tuplas, conjuntos e outras estruturas. O fragmento que segue mostra a impressão de todos os elementos da lista denominada `copas`:

```
copas = [1958, 1962, 1970, 1994, 2002]
for v in copas:
    print(v)
```

```
>>> 1958
>>> 1962
>>> 1970
>>> 1994
>>> 2002
```

Neste outro exemplo, uma lista de valores é definida diretamente na diretiva `for` e no comando de impressão associado é exibido o valor extraído da lista e seu quadrado. Aqui se combina a repetição com a computação.

```
for v in [1, 2, 3, 4, 5]:
    print(f'{v}**2 -->{v ** 2}')
```

```
>>> 1**2 -->1
>>> 2**2 -->4
>>> 3**2 -->9
>>> 4**2 -->16
>>> 5**2 -->25
```

O uso de `for` é bastante conveniente, pois requer apenas a definição de uma variável de controle, que, a cada iteração, recebe um dos valores armazenados na lista. Nada mais é necessário.

5.2.2 Contagem com for

Para que a diretiva `for` realize uma contagem, é necessário fornecer uma sequência com os valores presentes na contagem, além da variável de controle, como segue:

```
for v in [0, 1, 2, 3, 4]:
    print(v)
```

```
>>> 0
>>> 1
>>> 2
>>> 3
>>> 4
```

Embora simples, essa alternativa é claramente inconveniente quando a contagem tem mais do que alguns poucos valores. Nesses casos, é muito útil o uso auxiliar da função *built-in* `range()`, capaz de gerar a sequência de valores desejada. Observe o código que segue, que produz a mesma contagem do fragmento anterior.

```
for v in range(0, 5):
    print(v)
```

```
>>> 0
>>> 1
>>> 2
>>> 3
>>> 4
```

A função *built-in* `range()` é simples de usar e possui a seguinte sintaxe:

```
range([início, ] <fim> [, passo])
```

Os três elementos essenciais de qualquer contagem podem ser indicados em `range()`, como mostra a 5.6: o valor *inicial* `início`, de onde parte a contagem; o valor *final* `fim`, que indica o limite para o qual a contagem termina; e `passo`, para especificar o *incremento* ou *decremento* usado na contagem. Observe ainda o valor inicial `início` é opcional e, portanto, pode ser omitido, situação em que o valor 0 é utilizado. Da mesma forma, `passo` é opcional, sendo que sua omissão emprega o incremento unitário 1. Finalmente, deve ser destacado que o valor indicado em `fim` não é incluso na contagem, mas apenas os valores menores do que `fim`.



Os valores de *início*, *fim* e *passo* indicados na função `range()` devem ser inteiros, ou seja, do tipo `int`, caso contrário será lançada a exceção `TypeError`.

Por exemplo, uma contagem de 0 até 100 (incluso) pode ser indicada com qualquer uma das chamadas à função `range()` que seguem:

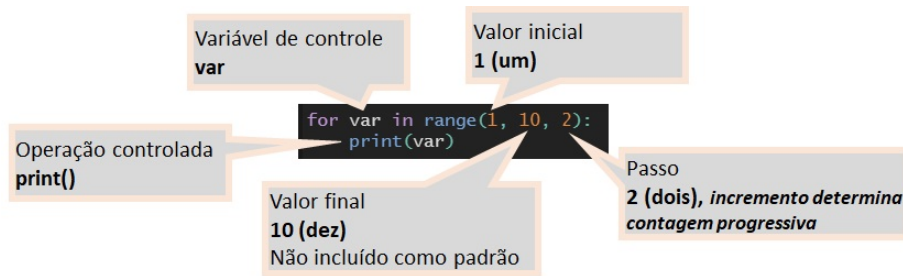


Figure 5.6: Elementos da contagem no for

```
# valor final é incluído com soma de 1
range(100 + 1)
range(101)
range(0, 100 + 1)
range(0, 101, 1)
```

```
>>> range(0, 101)
>>> range(0, 101)
>>> range(0, 101)
>>> range(0, 101)
```



Quando se escreve `range(N)`, a contagem se inicia em 0 e é finalizada em `N-1`, assim, a sequência gerada possui `N` valores, pois o 0 é incluído por padrão. Assim, se indicamos `range(3)`, a sequência gerada é 0, 1, 2, que contém três valores.

Outra contagem simples, por exemplo, de -25 até 25 (não incluso), de 3 em 3, pode ser realizada como segue. Na função `print()` substituímos o finalizador padrão `\n` por um espaço em branco, tornando a saída produzida mais compacta.

```
for x in range(-25, 25, 3):
    print(x, end=' ')
```

```
>>> -25 -22 -19 -16 -13 -10 -7 -4 -1 2 5 8 11 14 17 20 23
```

O uso combinado da diretiva `for` e da função `range()` simplifica a realização de contagens e de outras operações que empregam sequências simples de valores.

5.2.3 Agregação com for

A repetição automática `for` permite a realização de operações de agregação baseadas em contagens ou no percurso de sequências de dados, de maneira que podem ser obtidas somas, produtos, médias, obtenção de máximos e mínimos, além de outras contagens.

O exemplo que segue, denominado `agregacao_for.py`, efetua duas agregações com os inteiros existentes entre 1 e 10: sua soma e seu produto, exibindo seus resultados. As variáveis `ini` e `fim` armazenam os limites da faixa de valores que se deseja processar.

Observe que as variáveis necessárias para conter as agregações são inicializadas antes do laço de repetição `for` utilizado, sendo que `soma` é inicializada com zero e `produto` com 1, respectivamente os elementos neutros da soma e da multiplicação. A variável `contagem` recebe cada um dos valores da sequência gerada por `range(ini, fim+1)`.

Exemplo 5.4: `agregacao_for.py`

```
ini = 1 # definição do início da agregação
fim = 10 # definição do final da agregação
print(f'Agregação for [{ini}..{fim}]\n')

soma = 0 # variável para agregação de soma
produto = 1 # variável para agregação de produto

for contador in range(ini, fim + 1):
    soma += contador # soma no agregado
    produto *= contador # multiplica pelo agregado

print(f'Soma = {soma}\nProduto = {produto}')
```

```
>>> Agregação for [1..10]
>>> Soma = 55
>>> Produto = 3628800
```

Como nos exemplos `media.py` e `media2.py`, das seções anteriores, o script que segue, `media3.py`, calcula as médias individuais de um conjunto de alunos e também a média geral da classe, que é uma agregação.

Exemplo 5.4: `media3.py`

```
print('Média dos Alunos')

# Entrada de dados
numero_alunos = int(input('Número de alunos? '))
```

```
# Laço de Repetição
soma_medias = 0 # variável para agregação (soma)
for contador in range(numero_alunos):
    print(f'Aluno #{contador+1}')
    # lê notas
    nota_1 = float(input('Nota 1? '))
    nota_2 = float(input('Nota 2? '))
    # calcula média individual
    media = (nota_1 + nota_2) / 2
    # agrega média individual a soma das médias
    soma_medias += media
    # exibe média individual
    print(f'Nota 1:{nota_1:.1f}  Nota 2:{nota_2:.1f}  ' \
          f'Média:{media:.2f}\n')

# calcula e exibe média geral
media_geral = soma_medias / numero_alunos
print(f'Média geral:{media_geral:.2f}\n')

print('Fim')
```

A 5.7 exibe os resultados do programa `media3.py`, que são idênticos aos produzidos pelo programa `media2.py` quando usados os mesmos dados de entrada.



```
Anaconda Prompt (R-MINI~1)
$ python media3.py
Média dos Alunos
Número de alunos? 3
Aluno #1
Nota 1? 6.4
Nota 2? 2.7
Nota 1:6.4  Nota 2:2.7  Média:4.55

Aluno #2
Nota 1? 5.6
Nota 2? 7.5
Nota 1:5.6  Nota 2:7.5  Média:6.55

Aluno #3
Nota 1? 4.8
Nota 2? 8.3
Nota 1:4.8  Nota 2:8.3  Média:6.55

Média geral:5.88

Fim
```

Figure 5.7: Execução de `media3.py`

Todas as modificações introduzidas com o uso do `for` se relacionam à variável de controle `contador`, que passa a ser declarada na própria diretiva. A contagem dos alunos é gerada por `range(numero_alunos)`, que produz uma sequência de 0 até `numero_alunos-1`, que tem o número correto de valores, mas exige que na exibição da mensagem seja utilizada a expressão `contador+1` para exibir a sequência de 1 até `numero_alunos`, mais amigável para o usuário. Com o uso da repetição automática, não é necessário o incremento explícito da variável de controle `contador`. Nenhuma outra alteração é necessária, o que mostra que o uso do `for` simplifica o código.

Chapter 6

Decisão

A sequenciação é fundamental na programação, pois permite estabelecer uma sequência adequada de instruções, organizadas para solucionar um problema, ou seja, determina quais dados são necessários e como serão processados, representando, assim, a lógica com a qual resolvemos um problema. Desta maneira, são combinadas as operações de entrada e saída, de computação e de repetição.

Mas existem circunstâncias onde, considerados os resultados obtidos pelo programa, um determinado conjunto de instruções não é adequado para realizar uma etapa específica do problema, sendo requerido o uso de outra alternativa diferente para o processamento dos dados, ou seja, existem situações onde o fluxo sequencial de execução não mais atende as necessidades do problema.

A *decisão* é uma habilidade muito importante na programação de computadores, pois oferece uma maneira para escolher quais instruções serão executadas, isto é, permite que, durante a execução do programa, sejam selecionadas as instruções que serão executadas. A *decisão* ou *seleção* é realizada mediante a avaliação de uma expressão lógica que representa a *condição* que o programador estabelece como critério de escolha ou de seleção.

O Python oferece a diretiva de decisão `if`, que oferece várias possibilidades de uso:

- efetuar uma decisão simples (seção 6.1);
- realizar uma decisão completa (seção 6.2);
- aninhar várias decisões (seção 6.3);
- encadear várias decisões (seção 6.4).

Vejamos cada uma dessas possibilidades.

6.1 Decisão Simples

A *decisão simples* permite selecionar um comando ou um conjunto de comandos para execução, por meio da avaliação de uma expressão lógica, considerada como a *condição*, cujo resultado determina a execução dos comandos associados. A decisão simples é realizada pela diretiva `if` com a sintaxe que segue:

```
if <condição>:  
    <comando(s)>
```

A *condição* deve ser uma expressão de tipo `bool` (lógico), ou outra que possa ser avaliada logicamente. Podem ser associados um ou mais comandos à diretiva `if`, desde que indentados de quatro espaços à direita, que é o padrão do Python. Observe que existe um dois-pontos `:` obrigatório após a condição, tal como nas diretivas `while` e `for`. Observe o fluxograma exibido na Figura 6.1.

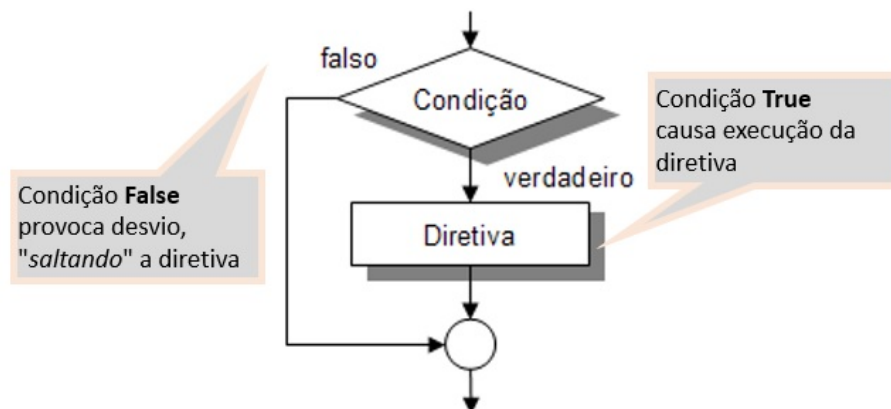


Figure 6.1: Fluxograma da decisão simples

A diretiva `if` é iniciada pela avaliação da condição. Quando a condição é avaliada como `True`, os comandos associados são executados sequencialmente; mas quando produz um resultado `False`, ocorre um *desvio* e a execução do programa continua com o comando seguinte à diretiva `if`.

Isto significa que *se* a condição é avaliada como verdadeira, os comandos associados são executados, mas *se* é avaliada como falsa, nenhum dos comandos associados é executado. Esse comportamento justifica que o `if` também seja conhecida como uma instrução de *desvio condicional*.

A condição utilizada pelo `if` pode ser uma expressão aritmética simples, cujo resultado será considerado `False` apenas quando resultar em zero, ou seja, qualquer valor diferente de zero, positivo ou negativo, é tomado como `True`. Também são válidas expressões aritméticas combinadas com operadores relacionais, que

são consideradas expressões lógicas devido seu resultado de tipo `bool`, ou expressões compostas, combinando várias expressões aritméticas e relacionais com o uso de operadores lógicos, que também produzem resultado `bool`.



É uma boa prática de programação utilizar como *condição* indicada no `if` expressões que resultem o tipo `bool`, explicitando qualquer comparação desejada, evitando o uso de operações implícitas. Isto traz mais clareza ao código e facilita sua manutenção.

Observe o fragmento que segue, onde a variável `valor` é *testada* pela diretiva `if`, isto é, seu conteúdo é avaliado logicamente. Como contém o inteiro 33, um valor não nulo, é considerada `True`, de maneira que a função `print()`, associado ao `if`, é executada, produzindo sua mensagem.

```
valor = 33
if valor:
    print('Valor contém', valor, 'que é NÃO NULO')
```

```
>>> Valor contém 33 que é NÃO NULO
```

Neste outro trecho, a diretiva `if` avalia se a variável `delta` tem conteúdo igual a zero, isto é, se seu conteúdo é nulo, de maneira que a função `print()`, associado ao `if`, é executada apenas nesta situação.

```
delta = valor - 33
if delta == 0:
    print('delta tem conteúdo NULO')
```

```
>>> delta tem conteúdo NULO
```

O exemplo que segue, `ingresso.py`, mostra uma aplicação mais efetiva da diretiva `if` no cálculo do valor a ser pago na compra de ingressos para o cinema. Inicialmente o programa define o `preco_ingresso` e o `desconto_padrao`, em seguida obtém duas informações do usuário: um inteiro que corresponde a *quantidade* de ingressos desejada e uma *string* com a resposta do usuário em relação a ser ou não estudante.

Com a *quantidade* de ingressos, informada pelo usuário, é calculado o `total_compra`. O `desconto_compra` é iniciado em zero. Se a resposta do usuário foi S ou s, a condição do `if` é avaliada como `True` e, assim, determinado um desconto para a compra com a aplicação da taxa `desconto_padrao` à *quantidade* de ingressos. Qualquer outra resposta diferente de S ou s é avaliada

como `False` e nenhum desconto é calculado, mantendo `desconto_compra` com o valor inicial 0. O programa se encerra com a impressão de mensagens detalhando os cálculos realizados e indicando o `total_final`, que é o preço a ser efetivamente pago pelos ingressos.

Exemplo 6.1: `ingresso.py`

```
print('Compra de Ingressos')
preco_ingresso = 22.00 # Preço do ingresso
desconto_padrao = 0.5 # Desconto para estudantes 50%

quantidade = int(
    input('Quantos ingressos deseja comprar? '))
resp = input('Você é estudante [S|N]? ')

total_compra = quantidade * preco_ingresso
desconto_compra = 0 # Precisa definir variável

if resp=='S' or resp=='s':
    # Aplica desconto SE É estudante
    desconto_compra = desconto_padrao * total_compra

total_final = total_compra - desconto_compra

print('-----')
print(f'Quant ingressos =', quantidade)
print(f' Preço ingresso = R$ {preco_ingresso:7.2f}')
print(f'Total da compra = R$ {total_compra:7.2f}')
print(f'      Desconto = R$ {desconto_compra:7.2f}')
print(f'      Total Final = R$ {total_final:7.2f}')
print('-----')
```

A Figura 6.2 mostra duas possibilidades da execução de `ingresso.py`, uma para um usuário que responde com `n` a questão *Você é estudante [S/N]?*, onde o desconto *não* é aplicado; outra quando é indicado `s` para tal pergunta, causando a aplicação do desconto.

Observe que o programa tem uma estrutura **E-P-S**, ou seja, realiza uma etapa de entrada, seguida do processamento que é condicional, isto é, realizado conforme a situação do usuário ser ou não um estudante, finalizada com a saída de mensagens informativas.

```

Anaconda Prompt (R-MINI~1)
$ python ingresso.py
Compra de Ingressos
Quantos ingressos deseja comprar? 3
Você é estudante [S|N]? n
-----
Quant ingressos = 3
Preço ingresso = R$    22.00
Total da compra = R$   66.00
Desconto = R$    0.00
Total Final = R$   66.00
-----

Anaconda Prompt (R-MINI~1)
$ python ingresso.py
Compra de Ingressos
Quantos ingressos deseja comprar? 3
Você é estudante [S|N]? s
-----
Quant ingressos = 3
Preço ingresso = R$    22.00
Total da compra = R$   66.00
Desconto = R$   33.00
Total Final = R$   33.00
-----

```

Figure 6.2: Possibilidades de `ingresso.py`

6.2 Decisão Completa

A *decisão completa* permite selecionar uma dentre duas alternativas para execução, onde cada alternativa pode ser constituída de um ou mais comandos. Para isso, efetua a avaliação de uma expressão lógica, que é a *condição* que determina tal escolha. A decisão completa utiliza a diretiva `if` e a *cláusula*¹ `else`, como na sintaxe que segue:

```

if <condição>:
    <comando(s)_1>
else:
    <comando(s)_2>

```

A diretiva `if/else` possui dois conjuntos de comandos associados, um para o `if` e outro para o `else` e seu funcionamento é ilustrado no fluxograma da Figura 6.3.

Como na decisão simples, a diretiva `if` é iniciada pela avaliação da condição, que deve ser uma expressão de tipo `bool` (lógico), ou que possa ser avaliada logicamente. Quando a condição é avaliada como `True`, os comandos associados à diretiva `if` são executados sequencialmente; mas quando produz um resultado `False`, são executados os comandos associados à cláusula `else`.

Isto significa que *se* a condição é avaliada como verdadeira, um conjunto de comandos é executado, *senão*, o outro conjunto de comandos é realizado, de maneira que *apenas* uma das alternativas é executada, ou seja, apenas um dos caminhos possíveis para execução do programa é percorrido, caracterizando a diretiva `if/else` como outra diretiva de desvio condicional.

Podem ser associados um ou mais comandos tanto à diretiva `if`, como à cláusula `else`, desde que indentados de quatro espaços à direita, que é o padrão do

¹Uma *cláusula* é sempre um elemento opcional de uma diretiva. Por exemplo, não é obrigatório que a diretiva `if` possua uma cláusula `else`.

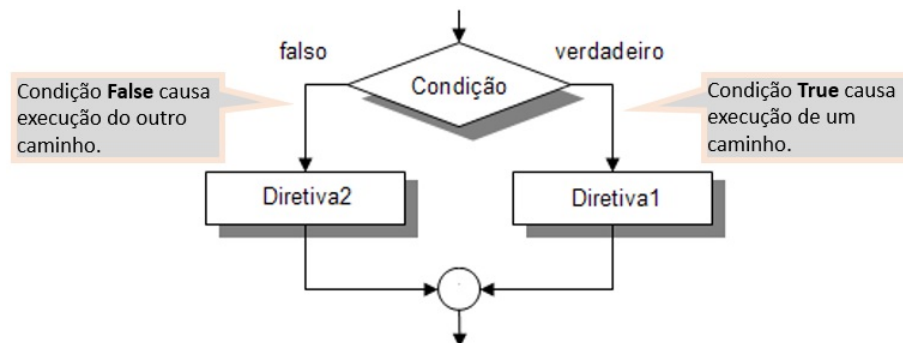


Figure 6.3: Fluxograma da decisão completa

Python. Observe que existem caracteres de dois-pontos : obrigatório após a condição e a cláusula `else`.

Cada diretiva `if` pode possuir apenas uma cláusula `else`, que é opcional e não pode ser usada isoladamente, ou seja, só pode ser empregada quando associada à uma diretiva `if`, formando um par `if/else`.

Considere a atribuição que segue de um literal a variável `valor`

```
valor = 7.654
```

No fragmento seguinte é testada a variável `valor`, verificando se seu conteúdo é positivo ou não, exibindo mensagens correspondentes.

```
if valor > 0:
    print('O número', valor, 'é positivo.')
else:
    print('O número', valor, 'é nulo ou negativo.')
```

```
>>> O número 7.654 é positivo.
```

Como `valor` contém 7.654, um número positivo, a mensagem exibida reflete esta situação. Se o conteúdo desta variável é alterado para um número negativo, como segue:

```
valor = -1.298
```

O mesmo trecho de código produz uma mensagem diferente, mostrando que apenas os comandos associados à diretiva `if` ou à cláusula `else` são executados.

```
if valor > 0:
    print('O número', valor, 'é positivo.')
else:
    print('O número', valor, 'é nulo ou negativo.')
```

```
>>> O número -1.298 é nulo ou negativo.
```

Neste outro trecho, a diretiva `if/else` avalia *se* o conteúdo da variável `nota` se encontra na faixa `[0, 10]`, produzindo mensagens apropriadas para cada situação.

```
nota = 6.5
if 0 <= nota and nota <= 10:
    print('Nota', nota, 'está na faixa [0, 10].')
else:
    print('Nota', nota, 'está FORA da faixa [0, 10].')
```

```
>>> Nota 6.5 está na faixa [0, 10].
```

O próximo exemplo traz um jogo simples (`adivinhe1.py`), que combina o uso de um laço de repetição condicional com uma decisão completa, além de operações de entrada e de saída. Inicialmente o programa realiza a importação da função `randrange()` do módulo `random`, usada em seguida.

São inicializadas três variáveis: `numero_secreto`, com um número aleatório sorteado entre `[0, 20]` (tal como para `range()`, o valor final não é incluído no intervalo indicado); `numero`, com valor `-1`, fora do intervalo do sorteio, para armazenar a tentativa do usuário; e `tentativas`, com valor `0`, para contar as tentativas realizadas pelo usuário.

O laço de repetição `while` testa se `numero` é diferente de `numero_secreto`, executando o código associado enquanto a tentativa do usuário não é o número secreto, ou seja, executando o código do jogo propriamente dito. No laço acontecem três ações: é lida uma tentativa (um palpite) do usuário armazenada em `numero`; o contador `tentativas` é incrementado; a variável `numero`, por meio de uma diretiva `if/else`, é testada em relação à sua igualdade com `numero_secreto`, de modo que a mensagem apropriada é exibida.

Exemplo 6.2: `adivinhe1.py`

```
from random import randrange

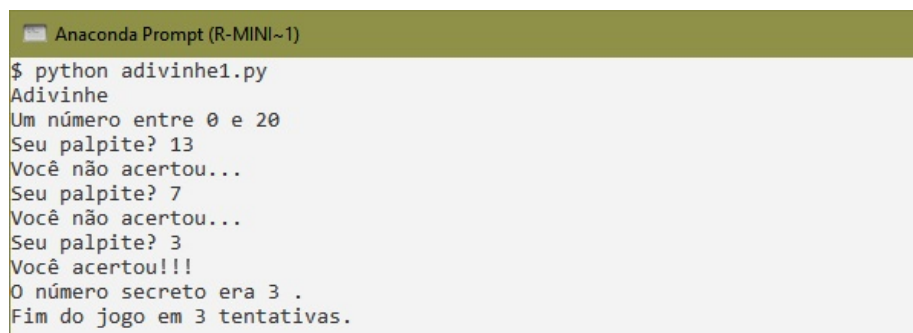
# Sorteia um inteiro entre 0 e 20
numero_secreto = randrange(0, 21)
```

```
print('Adivinhe\nUm número entre 0 e 20')

# inicializa número com valor inválido (fora da faixa)
numero = -1
tentativas = 0 # contador de tentativas
while numero != numero_secreto:
    # Lê palpite/tentativa do usuário
    numero = int(input('Seu palpite? '))
    tentativas += 1 # incrementa contador de tentativas
    # Verifica o palpite
    if numero != numero_secreto:
        print('Você não acertou...')
    else:
        print('Você acertou!!!')

print('O número secreto era', numero_secreto, '.')
print('Fim do jogo em', tentativas, 'tentativas.')
```

O laço continua apenas enquanto as tentativas do usuário são diferentes do número sorteado. Quando o usuário acerta o número secreto, o laço é encerrado e o jogo é finalizado com a impressão de mensagens adicionais. Os resultados desse programa podem ser vistos na Figura 6.4.



```
Anaconda Prompt (R-MINI~1)
$ python adivinhe1.py
Adivinhe
Um número entre 0 e 20
Seu palpite? 13
Você não acertou...
Seu palpite? 7
Você não acertou...
Seu palpite? 3
Você acertou!!!
O número secreto era 3 .
Fim do jogo em 3 tentativas.
```

Figure 6.4: Resultados de `adivinhe1.py`

O uso das decisões simples (`if`) e completas (`if/e/se`) confere enorme flexibilidade aos programas, que podem ter suas ações selecionadas conforme condições determinadas por suas variáveis, ou seja, conforme valores fornecidos pelos seus usuários e pelos cálculos realizados pelos próprios programas.

6.3 Decisões Aninhadas

No conjunto de instruções associado à uma diretiva `if` numa decisão simples (seção 6.1), e também à cláusula `else` numa decisão completa (seção 6.2), podem existir outras diretivas `if`, simples ou completas. Analise o trecho de código que segue.

```
frequencia = 0.78
media = 5.4
if frequencia >= 0.75:
    if media >= 6.0:
        print('Aluno aprovado')
    else:
        print('Aluno reprovado por nota')
else:
    print('Aluno reprovado por frequência')
```

```
>>> Aluno reprovado por nota
```

Neste trecho, as variáveis `frequencia` e `media` representam o percentual de presenças e a média final de um estudante; além disso, se a frequência é inferior à 0.75 (75%), ocorre a reprovação por falta, mas se superior a este valor, a aprovação depende da obtenção de média igual ou superior a 6.0. Assim, o primeiro `if`, externo, avalia se `frequencia` é maior ou igual à 0.75: caso verdadeiro, avalia a média, com o segundo `if`, interno, que verifica se a média é maior ou igual à 6.0, exibindo a mensagem apropriada; caso falso indica a reprovação por falta.

Várias diretivas `if` podem ser *aninhadas*, isto é, colocadas dentro de outras, o que permite criar estruturas mais complexas de decisão. Vale observar que o *aninhamento* ou associação de instruções é determinado, no Python, exclusivamente pela *indentação*, cujo padrão é de quatro espaços à direita.



Use o *aninhamento* de decisões quando as condições envolvidas utilizam variáveis diferentes, ou seja, quando expressam condições e verificações complementares, que graficamente poderiam ser organizadas como árvores de decisão.

6.4 Decisões Encadeadas

Outra possibilidade para o uso das operações de decisão é seu *encadeamento*, ou seja, duas ou mais decisões, simples ou completas, podem ser organizadas de

maneira a determinar três, quatro ou mais possibilidades distintas. A cláusula `elif`, usada em conjunto com diretiva `if/else` permite o encadeamento da avaliação de condições, com a sintaxe que segue:

```
if <condição_1>:
    <comando(s)_1>
elif <condição_2>:
    <comando(s)_2>
:
else:
    <comando(s)_finais>
```

A diretiva `if`, que inicia o encadeamento, avalia uma expressão lógica que constitui a condição que caracteriza o primeiro caso, para o qual pode ser associada um conjunto de comandos. O uso da cláusula opcional `elif` permite acrescentar a avaliação de uma segunda condição, juntamente com seu conjunto de comandos. Uma cláusula `else` possibilita indicar um último conjunto de comandos.

O par `if/else` permite selecionar a execução de um conjunto de comandos dentre dois conjuntos indicados, um para o `if`, outro para o `else`. Com a introdução de uma cláusula `elif`, sua condição secundária e seu respectivo conjunto de comandos, torna-se possível selecionar uma dentre três alternativas. Outras cláusulas `elif` podem ser adicionadas seguindo a mesma estrutura, ampliando as possibilidades da decisão completa.

O fragmento que segue mostra a utilização da diretiva `if/elif/else`, para efetuar a seleção dos comandos apropriados que devem ser executados para cada faixa de valores de `quantidade`. No exemplo, é calculado o `preco_total` decorrente da compra de uma `quantidade` de produtos com `preco_unidade`, ou seja, `quantidade * preco_unidade`. Mas, conforme a quantidade, é aplicado um desconto, de 5% para quantidades entre 11 e 20, de 10% para quantidade entre 21 e 50, e 15% para quantidades superiores a 50.

```
quantidade = 34
preco_unidade = 1.15
print(f'Preço unidade: R${preco_unidade:.2f} x ' \
      f'Quantidade: {quantidade}')

if quantidade <= 10:
    print('Sem desconto')
    preco_total = quantidade * preco_unidade
elif quantidade <= 20:
    print('Desconto de 5%')
    preco_total = 0.95 * quantidade * preco_unidade
elif quantidade <= 50:
    print('Desconto de 10%')
```

```
preco_total = 0.90 * quantidade * preco_unidade
else:
    print('Desconto de 15%')
    preco_total = 0.85 * quantidade * preco_unidade

print(f'Preço total: R${preco_total:.2f}')
```

```
>>> Preço unidade: R$1.15 x Quantidade: 34
>>> Desconto de 10%
>>> Preço total: R$35.19
```



Use o *encadeamento* de decisões quando as condições envolvidas utilizam as mesmas variáveis, ou seja, quando, em geral, se deseja determinar ações específicas para três ou mais faixas de valores.

O encadeamento de decisões é bastante útil, mas pode ser substituído pelo aninhamento de diretivas `if/else`. Compare o próximo fragmento de código, que usa aninhamento, com o anterior que usa encadeamento. Ambos têm o mesmo funcionamento.

```
quantidade = 34
preco_unidade = 1.15
print(f'Preço unidade: R${preco_unidade:.2f} x ' \
      f'Quantidade: {quantidade}')
```

```
if quantidade <= 10:
    print('Sem desconto')
    preco_total = quantidade * preco_unidade
else:
    if quantidade <=20:
        print('Desconto de 5%')
        preco_total = 0.95 * quantidade * preco_unidade
    else:
        if quantidade <=50:
            print('Desconto de 10%')
            preco_total = \
                0.90 * quantidade * preco_unidade
        else:
            print('Desconto de 15%')
            preco_total = \
                0.85 * quantidade * preco_unidade
```

```
print(f'Preço total: R${preco_total:.2f}')
```

```
>>> Preço unidade: R$1.15 x Quantidade: 34
>>> Desconto de 10%
>>> Preço total: R$35.19
```

Pode ser observado de cada `elif` (do encadeamento) é substituído por uma cláusula `else`, em cujo código existe uma diretiva `if/else` na qual a condição é a mesma indicada pelo `elif` correspondente. O funcionamento dos dois fragmentos é o mesmo, embora a versão com aninhamento seja ligeiramente maior.



A *aninhamento* de diretivas `if/else` sempre pode substituir seu *encadeamento*. No entanto, o *encadeamento* não pode substituir todas as possibilidades do *aninhamento*.

O próximo exemplo modifica o jogo simples apresentado no exemplo anterior (seção 6.2), mantendo a maior parte do código. A alteração mais importante é a utilização da diretiva `if/elif/else`, de modo que, cada tentativa do usuário é sinalizada com as indicações *alto* e *baixo*, quando o palpite é comparado com o `numero_secreto`. Por conta destas mensagens, o número aleatório é sorteado pela função `randrange()` do módulo `random` entre `[0, 100]`. Segue o código de `adivinhe2.py`.

Exemplo 6.3: `adivinhe2.py`

```
from random import randrange

# Sorteia um inteiro entre 0 e 100
numero_secreto = randrange(0, 101)

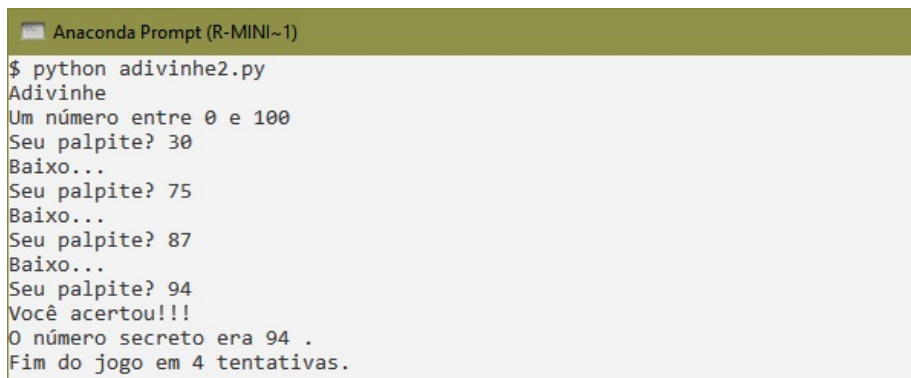
print('Adivinhe\nUm número entre 0 e 100')

# Inicializa número com valor inválido (fora da faixa)
numero = -1
tentativas = 0 # Contador de tentativas
while numero != numero_secreto:
    # Lê palpite/tentativa do usuário
    numero = int(input('Seu palpite? '))
    tentativas += 1 # incrementa contador de tentativas
    # Verifica o palpite
    if numero < numero_secreto:
        print('Baixo...')
```

```
elif numero > numero_secreto:
    print('Alto...')
else:
    print('Você acertou!!!')

print('O número secreto era', numero_secreto, '.')
print('Fim do jogo em', tentativas, 'tentativas.')
```

A Figura 6.5 mostra resultados possíveis do programa `adivinhe2.py`.

A screenshot of the Anaconda Prompt terminal window. The title bar reads "Anaconda Prompt (R-MINI~1)". The terminal shows the execution of a Python script named "adivinhe2.py". The output of the script is as follows: "Adivinhe", "Um número entre 0 e 100", "Seu palpite? 30", "Baixo...", "Seu palpite? 75", "Baixo...", "Seu palpite? 87", "Baixo...", "Seu palpite? 94", "Você acertou!!!", "O número secreto era 94 .", and "Fim do jogo em 4 tentativas.".

```
Anaconda Prompt (R-MINI~1)
$ python adivinhe2.py
Adivinhe
Um número entre 0 e 100
Seu palpite? 30
Baixo...
Seu palpite? 75
Baixo...
Seu palpite? 87
Baixo...
Seu palpite? 94
Você acertou!!!
O número secreto era 94 .
Fim do jogo em 4 tentativas.
```

Figure 6.5: Resultados de `adivinhe2.py`

6.5 Desvio Incondicional

O Python oferece, além da diretiva de desvio condicional `if`, três diretivas de *desvio incondicional*, isto é, que executam um desvio no fluxo sequencial de execução, independente da avaliação de qualquer condição. Tais diretivas são:

- **break**, que permite o encerramento da execução de laços de repetição (seção 6.5.1);
- **continue**, que possibilita a aceleração da execução de laços de repetição (seção 6.5.2) e
- **return**, que efetua o retorno de resultado ao ponto de chamada da função (seção 7.2).

As diretivas `break` e `continue` serão apresentadas e discutidas na sequência, enquanto `return` será tratado juntamente com as funções (Capítulo 7).

6.5.1 Desvio com break

A diretiva `break` se destina a encerrar a execução das diretivas de repetição `while` e `for`, evitando que sejam executadas até seu final, fazendo com que a execução prossiga com as diretivas seguintes às estruturas em que foi usada. Sua sintaxe é simples:

```
break
```

Como esta diretiva é incondicional, nunca é aplicada diretamente, pois impede a execução das diretivas seguintes. Por isso, sempre é utilizada em combinação com a diretiva `if`, executando o desvio em condições específicas.

Na construção de laços, às vezes é desejável que a repetição seja interrompida quando uma situação especial é encontrada. Por exemplo: ao pesquisar por um nome ou palavra em uma lista de *string*, o laço de repetição usado pode ser interrompido quando termo desejado é encontrado, como no fragmento que segue, que procura pelo nome *Guido* na lista `nomes`.

```
nomes = [ 'Dennis', 'Brian', 'James', 'Guido', 'Bjarne',
          'Alan', 'John', 'Grace', 'Yukihiro', 'Anders' ]
alvo = 'Guido' # nome a ser procurado na lista
testes = 0 # número de nomes pesquisados
for n in nomes:
    testes += 1
    if n == alvo:
        print(f"{alvo}" foi encontrado na lista.')
        break
print('Foram executados', testes, ' testes de',
      len(nomes), '.')
```

```
>>> "Guido" foi encontrado na lista.
>>> Foram executados 4 testes de 10 .
```

No exemplo `primo.py`, que segue, é testado o número inteiro fornecido pelo usuário para determinar se é um número primo ou não. Os números primos são aqueles maiores do que '1' e divisíveis apenas eles mesmos (GOUVEIA, 2021b), assim, para determinar se um número é primo, basta garantir que não possua divisores entre 2 e o valor de sua metade. Com o operador `%` (resto da divisão inteira) é possível verificar se um número é divisor de outro: fazendo `A % B`, um resto 0 indica que B é divisor de A (ou que A é múltiplo de B); e um resto diferente de 0 indica a inexistência dessas relações.

Um laço `for` é empregado para efetuar a divisão do `numero` dado com todas as possibilidades do intervalo `[2, numero//2]`. Se encontrado algum divisor,

o número não é primo e, portanto, não existe necessidade de prosseguir com o teste, momento em que uma diretiva **break** pode encerrar o laço. A variável auxiliar **primo** atua como flag: inicialmente recebe o valor **True**, quando a divisão é possível, recebe **False**, pois número não é primo, permitindo selecionar a exibição da mensagem final correta.

Exemplo 6.4: primo.py

```
print('Teste para numeros primos')
numero = int(
    input('Digite um inteiro para verificação: '))

primo = True # supõe que número é primo
# laço para todas as possibilidades de teste
for n in range(2, numero // 2):
    # verifica se número divisível por n
    if numero % n == 0:
        primo = False # tem divisor, não é primo
        break

if primo:
    print(f'{numero} É primo!')
else:
    print(f'{numero} NÃO É primo!')
```

A Figura 6.6 mostra resultados possíveis do programa `primo.py`.



```
Anaconda Prompt (R-MINI~1)
$ python primo.py
Teste para numeros primos
Digite um inteiro para verificação: 14303
14303 É primo!
```

Figure 6.6: Resultados de `primo.py`

A diretiva **break** também pode ser utilizada para interromper a execução de laços condicionais **while**, como no fragmento que segue.

```
i = 14 # valor inteiro positivo arbitrário
while i > 0 : # repete enquanto i é maior que zero
    # operações desejadas
    #
    if i == 10: # condição especial
        break # interrompe o laço
    #
    # outras operações
```

```
i -= 2 # modifica variável de controle  
print('Valor final de i:', i)
```

```
>>> Valor final de i: 10
```



Caso sejam aninhadas duas ou mais estruturas de repetição **for** ou **while**, ou seja, colocadas uma dentro das outras, a execução de uma diretiva **break** encerra apenas a estrutura na qual foi acionada.

Para finalizar várias estruturas aninhadas, a partir das mais internas, a diretiva **break** deverá ser utilizada em conjunto com outras variáveis auxiliares.

6.5.2 Desvio com **continue**

A diretiva **continue** permite acelerar a execução de um laço de repetição por meio da abreviação da iteração corrente, reduzindo o número de comandos executados. Sua sintaxe é simples:

```
continue
```

Sabemos que as diretivas **while** e **for** são estruturas de controle de repetição que permitem construir laços nos quais os comandos associados são iterados, ou seja, têm sua execução repetida. A diretiva **continue** permite interromper a iteração corrente, mas, diferentemente do **break**, faz com que a estrutura de controle avalie se é possível realizar uma próxima iteração, continuando caso afirmativo. Assim, o **continue** permite *abreviar* a repetição corrente, não executando os comandos associados que a seguem.

O uso de **continue** pode ser interessante em certos casos, podendo ser empregado com qualquer uma das estruturas de repetição disponíveis. Observe o fragmento que segue, onde um laço **for** realiza uma contagem regressiva de 10 até 1. Nos comandos associados ao laço, são testados os múltiplos de 3 ou 5, situação na qual a diretiva **continue** é executada, evitando o processamento posterior destes valores. Para os demais, ocorre sua acumulação na variável **soma** e a impressão de uma mensagem. Após o laço a variável **soma** é exibida.


```
soma = 0
for n in range(10, 0, -1):
    if n % 3 == 0 or n % 5 == 0:
        continue;
    soma += n
    print(n, n * '#')
print('soma =', soma)
```

```
>>> 8 #####
>>> 7 #####
>>> 4 ####
>>> 2 ##
>>> 1 #
>>> soma = 22
```

O Exemplo 6.5 (`polinomio.py`) mostra a aplicação do `continue` com a diretiva `while` num programa que exibe um gráfico de dispersão simples para um polinômio de grau dois (GOUVEIA, 2021a) na forma $f(x) = a * x^2 + b * x + c$, cujos coeficientes `a`, `b` e `c`, e a faixa de sua abcissa (variável `x`) são definidos e exibidos no início do programa. Por questões de simplificação, o eixo `y` (ordenada) como uma reta na horizontal, formada por traços, intercalada de valores da escala, com uso de uma diretiva `for`.

Um laço `while` percorre todos os pontos da abcissa entre os limites dados (`xi` até `xf`), com uma variação fixa de `0.2`, acrescentando novas linhas ao gráfico, de modo que o eixo `x` (abcissa) fica desenhado na vertical, *orientado* para baixo. Para cada ponto `x` da abcissa (exibidos com `print()` sem finalização de linha) é determinado o valor `y` da ordenada e incrementado o valor de `x` (abcissa) de `0.2`. Então é testado o valor da ordenada, verificando se é negativo ou maior que `5`, pois como nesses casos o valor não pode ser plotado adequadamente, a linha é finalizada com um `print()` e iteração é abreviada com o uso da diretiva `continue`, a qual evita que as demais diretivas do laço sejam executadas. Se o valor da ordenada estiver no intervalo `[0, 5]`, o ponto correspondente é exibido com um caractere `+` posicionado em escala, ou seja, precedido por um número de espaços calculado pela expressão `int(10 * y - 1)`.

Exemplo 6.5: `polinomio.py`

```
# Coeficientes a, b, c do polinômio de 2o grau
a, b, c = -1.5, 3, 2.2
# ordenadas inicial e final
xi, xf = 0, 2.6

print('Polinomio de 2o. grau')
print(f'Coeficientes a={a:.1f}, b={b:.1f}, c={c:.1f}')
```

```

print(f'Ordenada inicial={xi:.1f}, final={xf:.1f}\n')

# desenha eixo da abcissa
for i in range(0, 6):
    if i == 0: print('    ', end='')
    print(f'{i}-----', end='')
print()

# determina pontos do polinômio
x = xi
while x <= xf:
    # exibe valor da abcissa
    print(f'{x:5.1f}', end='');
    # calcula ordenada
    y = a * x ** 2 + b * x + c
    x += 0.2 # incremento fixo da abcissa
    if y < 0 or y > 5: # se ordenada fora de [0, 5]
        print()
        continue
    # plota valor da ordenada em escala 10:1
    print(int(10 * y - 1) * ' ', '+ ', round(y, 2), sep='')

```

```

>>> Polinomio de 2o. grau
>>> Coeficientes a=-1.5, b=3.0, c=2.2
>>> Ordenada inicial=0.0, final=2.6
>>>
>>>  0-----1-----2-----3-----4-----5-----
>>>
>>>  0.0                      + 2.2
>>>  0.2                      + 2.74
>>>  0.4                      + 3.16
>>>  0.6                      + 3.46
>>>  0.8                      + 3.64
>>>  1.0                      + 3.7
>>>  1.2                      + 3.64
>>>  1.4                      + 3.46
>>>  1.6                      + 3.16
>>>  1.8                      + 2.74
>>>  2.0                      + 2.2
>>>  2.2          + 1.54
>>>  2.4      + 0.76
>>>  2.6

```

O objetivo deste exemplo é mostra o uso de `continue` num problema real e ilustra a dificuldade da construção de gráficos, que podem ser gerados de maneira

muito mais rica e bonita com os vários pacotes do Python destinados à esta tarefa, como o Matplotlib, Seaborn, Bokeh ou PyX.

Os exemplos deste capítulo exploraram não apenas a habilidade da decisão, mas sua combinação com a computação, operações de entrada e saída, repetição e, é claro, sequenciação. Nos próximos capítulos será tratada a última das habilidades essenciais da programação, a modularização, que abrirá muitas novas possibilidades para o desenvolvimento de programas.

Chapter 7

Funções

Uma função é um trecho de código independente, um fragmento especial do programa, ou seja, um subprograma, o qual pode ser reutilizado. Toda função tem:

- Um nome que constitui seu identificador e
- Um trecho de código que realiza uma tarefa específica.

Além disso, uma função pode, opcionalmente:

- Receber um conjunto de parâmetros que podem customizar a realização da tarefa; e
- Retornar um valor, de qualquer tipo, como resultado da tarefa executada.

A construção de funções é uma prática tanto típica, quanto essencial na programação, pois de utilidade inegável. A sintaxe para definição de uma função é simples:

```
def nome_da_funcao ([lista_parametros]):  
    comando(s)  
    return [valor]
```

A palavra reservada **def** inicia, obrigatoriamente, a declaração de uma função, onde:

- o **nome_da_funcao** define o nome pelo qual a função será acionada, é seu identificador;
- a **lista_parametros** é uma lista delimitada por parênteses (), opcional, que pode conter uma ou mais declarações de parâmetros da função;

- os *comandos* constituem o conjunto de instruções que executam a tarefa da função;
- a diretiva `return` indica o término da função e, opcionalmente, determina o resultado retornado.

Deve ser observado que: são obrigatórios os parênteses que delimitam a lista de parâmetros, mesmo que vazia, assim como os dois pontos : seguintes. Os comandos contidos na definição da função devem ser indentados à direita com quatro espaços, seguindo o padrão do Python.

7.1 Tipos de funções

7.2 Retorno de Valor

7.3 Passagem de Parâmetros

7.4 Parâmetros *Default*

7.5 Parâmetros Variáveis

Chapter 8

Módulos

Python é uma plataforma de programação bastante conveniente, pois oferece várias alternativas para sua utilização. A mais simples é usar o *interpretador* Python em seu modo de console, que permite executar um comando de cada vez, interativamente, para declarar variáveis, realizar operações de computação, executar qualquer diretiva, além de possibilitar a definição de funções e sua utilização.

Outra alternativa é a organização das construções necessárias em um arquivo de texto, usualmente com extensão `.py`, que podem ser executadas pelo *interpretador* como um *script*, evitando, com isso, que sequências longas ou complexas tenham que ser redigitadas, possibilitando ainda a inclusão de comentários e a exibição de mensagens mais detalhadas.

O uso de *scripts* facilita muito o uso do Python e pode ser ainda mais interessante e efetivo quando emprega seu sistema de *módulos*.

Um *módulo* é um arquivo que contém diretivas e definições do Python (PSF, 2021), o qual pode ser importado e, assim, utilizado pelo interpretador em modo interativo, durante a execução de um *script* ou mesmo por outros módulos. O nome do arquivo de um módulo é, em geral, o *nome* do módulo ao qual se adiciona a extensão `.py`.

8.1 Criação de módulos

8.2 Importação geral

Importação geral, não seletiva

```
import <modulo>
```

Após sua importação, o nome do módulo se torna disponível como uma *string* armazenada na variável global `__name__`.

8.3 Programa Principal

```
if __name__ == '__main__':  
    print('Programa Principal')  
else:  
    print('Módulo')
```

```
>>> Programa Principal
```

8.4 Importação seletiva

Importação seletiva

```
from <modulo> import <elemento>
```

8.5 Pacotes

Appendix A

Instalação básica

[illegible][illegible]

Appendix B

Instalação do Anaconda

[illegible][illegible]

Bibliography

- BRANDÃO, L. d. O. (2021). *Introdução às funções*. IME/USP, São Paulo, SP. Material didático para Introdução à Programação. Acesso em 26 jun. 2021.
- GOUVEIA, R. (2021a). *Equação do Segundo Grau*. Toda Matéria. Acesso em 14 jun. 2021.
- GOUVEIA, R. (2021b). *Números Primos*. Toda Matéria. Acesso em 12 jul. 2021.
- GOUVEIA, R. (2021c). *Paralelepípedo*. Toda Matéria. Acesso em 29 jun. 2021.
- KOPEC, D. (2019). *Classic Computer Science Problems in Python*. Manning Publications Co., Shelter Island, New York, 1st edition. ISBN 978-1617295980.
- PSF, P. S. F. (2021). 6.modules. The Python Tutorial. Acesso em 14 jul. 2021.
- XIE, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.
- XIE, Y. (2021). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.22.