Python - Guia do Programador

Peter Jandl Junior

2021-06-28

Contents

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 **R Markdown** e uso do pacote **bookdown**[1] (Xie, 2021, 2015), portanto emprega o suporte do **Pandoc**, como, por exemplo, para indicar expressões matemáticas, tal como $f(x) = a * x^2 + b * x + c$.

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

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

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

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

[1] Bookdown no GitHub: https://github.com/rstudio/bookdown.

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, sintaxa bela e suscinta, 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 Rossun, que então trabalhava no *Stichting Mathematish 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), Rossun, 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 Rossun, 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 portabilidade[2]. Além de ser

[1] Veja 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 tambem 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 7), 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 um conjunto de instruções que devem ser executados mais de uma vez, o que inclui determinar o número de vezes que tais instruções serão executadas ou a condição que exige sua repetição. Além disso, a repetição permite reduzir a quantidade de instruções

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

necessárias para resolver um problema, e 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

[1] Dicio (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 SQL[2] 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 boleanos), 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
`int`	Inteiro (ou integral), valor númerico sem parte fracionária.
`float`	Real, valor numérico com parte fracionária em ponto flutuante.
`bool`	Lógico (ou boleano).
`string`	*String* ou cadeia de caracteres.
`complex`	Número complexo, com a parte imaginária identificada pelo sufixo `j`.

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

[3] O 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 boleano, 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`:

```
x = 15
```

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

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
```

[4]Segundo 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/).

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)
```

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

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
```

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

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

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

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 inteiros 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
```

[8] Propriedade *comutativa*: a ordem dos operandos não altera o resultado, pois na adição temos que `A + B = B + A`.

[9] Propriedade *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`.

[10] Propriedade *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`.

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
```

```
>>> 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.

[12] Propriedade *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`.

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

[15] Propriedade *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`.

```
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 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
```

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

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 raíz 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
`>`	Maior que.	2
`>=`	Maior ou igual a.	2
`<`	Menor que.	2
`<=`	Menor ou igual a.	2
`==`	Igual.	2

Operador	Relação	Aridade
`!=`	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). Um exemplo simples do uso destes operadores está no fragmento que segue, no qual se verifica se o valor `1964` *é menor* que `1995` e produz um retorno `True`.

```
1964 < 1995
```

```
>>> True
```

O próximo fragmento mostra outro uso simples destes operadores, onde se compara o conteúdo de `A` (que é `1931`) e o literal `2021` em relação a sua igualdade, o que produz um retorno `False`.

```
A = 1964
A == 2021
```

```
>>> False
```

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
`and`	E-lógico (conjunção).	2
`or`	Ou-lógico (disjunção).	2
`not`	Não-lógico (negação).	1
`in`	Membro de.	2
`is`	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-verdade[18] 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
```

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

Variáveis do tipo `bool` e outras numéricas podem ser combinadas com os operadores lógicos devido a equivalância *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)

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`.

Mas 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 estabele 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	`await x`	Expressão `await`
4	`**`	Potenciação
5	`+x`, `-x`, `~`	Positivo, negativo, não *bitwise*
6	`*`, `/`, `//`, `%`	Multiplicação, divisão, divisão inteira, módulo
7	`+`, `-`	Adição, subtração
8	`<<`, `>>`	Deslocamento à esquerda e à direita
9	`&`	E *bitwise*
10	`^`	Ou-exclusivo *bitwise*
11	`|`	Ou *bitwise*
12	`<`, `<=`, `>`, `>=`, `==`, `!=`,`in`, `not in`, `is`, `not is`	Relacionais, teste de membro e teste de identidade
13	`not x`	Não lógico
14	`and`	E lógico
15	`or`	Ou lógico

Nível	Operadores	Descrição
16	`if - else`	Expressão condicional
17	`lambda`	Expressão lambda
18	`=`	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 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 inteiro 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
```

```
>>> 578246000
```

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 7 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 tipo `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 (*backslash*)
`\n`	Nova linha (*new line*)
`\r`	Retorno de carro (*carriage return*)
`\t`	Tabulação
`\b`	Retrocesso (*backspace*)
`\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 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 argumentos e uma quebra de linha (`\n`) para finalizar a saída.

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

Neste exemplos, 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()`.

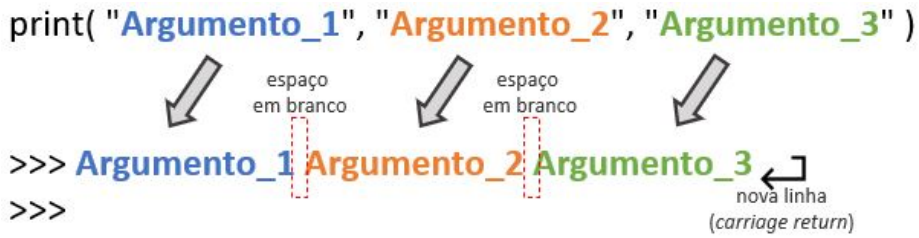

Figure 3.2: Inserção de separador e finalizador no `print()`

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 explica 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-int* `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-int* `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
`b`	Inteiro em formato binário (base 2)
`c`	Inteiro como caractere
`d`,`n`	Inteiro decimal
`e`,`E`	Real em notação científica
`f`,`F`	Real com precisão fixa
`o`	Inteiro em formato octal (base 8)
`%`	Porcentagem
`s`	String (cadeia de caracteres)
`x`,`X`	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()`.

[1] No 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.

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

```
>>> 15
```

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.

```
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. Mas, caso o valor requeira 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 exempo 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 estabele 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$ {:7.2f}'.format(n0))
```

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

```
print('Valor = R$ %7.2f' % (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 = {:8.4f}'.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'A linguagem {s3}, um valor inteiro {i1} e outro real {f0}.')
```

```
>>> A linguagem Python, um valor inteiro 2021 e outro 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$ {:7.2f}'.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 tanto quanto 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.

[1] Dicio (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 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.

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*.

```
print('Hello, world!')
```

```
>>> 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.

Chapter 5

Repetição

Mais uma habilidade...

5.1 Repetição Condicional

Dá-lhe *while*!

5.2 Repetição Automática

Dá-lhe *for*!

Chapter 6

Decisão

6.1 Decisão Simples

Aqui tratamos do *if*!

6.2 Decisão Completa

Aqui tratamos do *if/else*!

6.3 Decisões Encadeadas

Aqui tratamos do *if/elif/else*!

Chapter 7

Modularização

7.1 Programa Principal

7.2 Funções

7.2.1 Tipos de funções

7.2.2 Retorno de Valor

7.2.3 Passagem de Parâmetros

7.2.4 Parâmetros *Default*

7.2.5 Parâmetros Variáveis

7.3 Importação

7.4 Criação de pacotes e módulos

Appendix A

Instalação básica

Blá blá.

Blá blá.

Appendix B

Instalação do Anaconda

Blá blá.

Blá blá.

Bibliography

Brandão, L. d. O. (2021). *Introdução às funções*. IME/USP, São Paulo, SP. Material didático para Introdução à Programação.

Kopec, D. (2019). *Classic Computer Science Problems in Python*. Manning Publications Co., Shelter Island, New York, 1st edition. ISBN 978-1617295980.

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.