Curso de Python

Helton Maia

Jan 02, 2024

CONTENTS

Bem-vindo ao curso de Python para Iniciantes

Este curso é uma introdução à linguagem de programação Python, voltada para estudantes universitários. Ele foi desenvolvido para tornar o aluno apto a utilizar linguagens de programação de alto nível para implementar soluções computacionais a problemas provenientes das diversas áreas das engenharias e das ciências.

O curso é organizado em 10 capítulos, que cobrem os principais conceitos da programação em Python. No início, são apresentados os conceitos básicos, como variáveis, expressões, loops e condicionais. Em seguida, são abordados os conceitos mais avançados, como funções, módulos e bibliotecas. O curso também inclui um capítulo sobre orientação a objetos, que pode ser usado como material complementar.

O curso é um recurso valioso para estudantes que desejam aprender a programar em Python. Ele é claro, conciso e inclui muitos exemplos e exercícios para ajudar os alunos a aprenderem e praticarem o conteúdo abordado.

CHAPTER

ONE

CAPÍTULO 1: INTRODUÇÃO À PROGRAMAÇÃO EM PYTHON

1.1 O que é programação de computadores?

Programar computadores é a arte e ciência de conceber e criar conjuntos de instruções que capacitam computadores a realizar tarefas específicas. Esse processo envolve a expressão lógica de algoritmos por meio de uma linguagem de programação, atuando como a ponte entre a mente humana e a máquina.

Essa habilidade é fundamental para aqueles que buscam atuar no universo da computação, desempenhando um papel essencial em diversas disciplinas, como engenharia, ciência, negócios, saúde, educação e entretenimento. A capacidade de programar não apenas possibilita a automação de processos, mas também estimula a resolução criativa de problemas e impulsiona a inovação tecnológica.

Na prática da programação, os desenvolvedores convertem conceitos abstratos em linguagem compreensível pelos computadores, proporcionando-lhes a habilidade de executar tarefas complexas. Essa interação entre humanos e máquinas desempenha um papel fundamental na contínua evolução da sociedade digital, moldando desde avanços científicos até transformações sociais significativas.

A habilidade de programar transcende a mera condição técnica, transformando-se em uma ferramenta importante para explorar novas ideias e o aprimoramento pessoal. Filosoficamente falando, programar é também uma forma de enxergar o mundo sob diferentes perspectivas, processos e abstrações.

Dentro do contexto do Python, esta é uma linguagem considerada de alto nível, sendo interpretada e multiparadigma. Tal caracterização implica que o Python destaca-se por sua facilidade de aprendizado e uso, sendo aplicável a uma ampla gama de propósitos. Sua versatilidade se destaca ainda mais pela capacidade de suportar diversos paradigmas de programação, proporcionando aos desenvolvedores uma abordagem flexível e adaptável para resolver problemas em diferentes domínios.

O Python é uma linguagem de programação versátil que desempenha papéis significativos em diversas áreas. A seguir, um breve resumo de algumas dessas possibilidades:

- **Engenharia e Ciências:** Utilizado em simulação, análise e visualização de dados, assim como em projetos de aprendizado de máquina. Sua sintaxe clara e concisa, bem como sua ampla biblioteca de módulos científicos, tornam-o uma escolha popular para tais abordagens.
- **Negócios:** Como uma ferramenta essencial para a análise de dados, automação de processos e desenvolvimento de aplicativos web. Sua flexibilidade e eficiência o tornam uma escolha versátil para soluções empresariais.
- **Educação:** O Python é a linguagem de programação mais popular para o ensino de programação em escolas e universidades. Sua sintaxe simples e intuitiva torna-o uma linguagem fácil de aprender, mesmo para iniciantes.
- **Entretenimento:** É uma linguagem de programação versátil e robusta que é frequentemente utilizada na criação de jogos, aplicativos móveis e outros softwares de entretenimento. Sua flexibilidade e robustez permitem que seja usada para criar aplicações de alta qualidade e de todos os tipos.
- **Saúde:** Fortemente utilizado em análise de dados médicos e desenvolvimento de softwares especializados e pesquisa médica. Sua capacidade analítica e adaptabilidade o tornam uma ferramenta valiosa para a pesquisa e a inovação em saúde.

Python é uma linguagem de programação versátil que atende a uma ampla gama de necessidades, desde automatização de tarefas repetitivas até desafios avançados. Suas aplicações incluem automação, análise de dados, desenvolvimento de jogos, inteligência artificial, automação de redes, aplicações científicas, desenvolvimento de aplicativos de desktop e web, construção de APIs, segurança cibernética, simulações científicas e matemáticas, Internet das Coisas (IoT) e produção de mídia. A versatilidade do Python o torna uma ferramenta indispensável em diversas áreas, proporcionando uma base sólida para inovação no cenário tecnológico atual. Dominar Python não é apenas uma habilidade essencial, mas também uma maneira de explorar as constantes inovações e desafios tecnológicos em evolução.

1.2 O que você precisa para começar?

Para iniciar seu aprendizado em Python, além desta documentação, é fundamental contar com os seguintes elementos:

1. **Computador com Acesso à Internet:** Recomenda-se utilizar um computador com conexão à internet para facilitar o download de pacotes adicionais e o acesso à documentação online, enriquecendo sua experiência de aprendizado. No entanto, é válido destacar que é possível programar em Python em um ambiente offline, o que se torna uma opção viável em situações em que a conexão à internet não está disponível.
2. **Editor de Texto ou IDE (Ambiente de Desenvolvimento Integrado):** Escolha um editor de texto que atenda às suas preferências e necessidades. Pode ser um editor simples como o Notepad ou algo mais avançado como o Sublime Text, Atom, Visual Studio Code, ou editores online como o Replit, Google Colab ou o Jupyter Notebook. Além disso, o PyCharm é uma poderosa IDE específica para Python que oferece recursos avançados e é amplamente utilizada por desenvolvedores.
3. **Interpretador Python:** Faça o download do interpretador Python diretamente do site da Python Software Foundation (https://www.python.org/). Alternativamente, você pode utilizar ambientes online como o Replit, que já incluem um interpretador Python e um editor de texto integrados.

Equipado com esses recursos, você estará pronto para explorar e aprimorar suas habilidades em Python. Seja trabalhando localmente em seu computador ou em ambientes online, você terá a flexibilidade necessária para mergulhar no mundo da programação, adaptando-se ao seu estilo de aprendizado preferido.

Observação: O Python é uma linguagem de programação interpretada, ou seja, seu código é executado diretamente pelo interpretador, sem ser convertido para um formato de código de máquina. Isso torna o Python uma linguagem mais fácil de aprender e usar, pois não é necessário compilar o código antes de executá-lo.

1.3 Escrevendo seu primeiro programa em python

Neste material introdutório, apresentamos um exemplo conciso de um programa em Python que realiza a soma de dois números e exibe o resultado na tela. O arquivo pode ser nomeado "soma.py", e o código é o seguinte:

```
a = 1
b = 2
soma = a + b
print(soma)
```

```
3
```

Neste exemplo didático, valores são atribuídos às variáveis `a` e `b`. Posteriormente, uma nova variável chamada `soma` armazena o resultado da adição desses valores. Finalmente, o comando `print` é utilizado para exibir o resultado da soma.

Aqui está a explicação de cada linha do código:

- **`a = 1`**: Atribui o valor inteiro 1 à variável `a`. Variáveis armazenam dados em Python.

- **`b = 2`**: Atribui o valor 2 à variável `b`. Agora, `a` contém 1 e `b` contém 2.
- **`soma = a + b`**: Cria a variável `soma` e atribui a ela a soma dos valores em `a` e `b`, resultando em 3.
- **`print(soma)`**: Utiliza a função `print` para exibir o valor armazenado em `soma`, que é 3.

Resumidamente, o programa define dois valores (1 e 2) em duas variáveis (`a` e `b`), realiza a soma desses valores e armazena o resultado em uma terceira variável (`soma`), e finalmente, imprime o resultado (3) na tela.

O que são variáveis?

São identificadores nomeados que armazenam e representam dados, essenciais para a manipulação dinâmica de informações em um programa.

O que são funções?

São blocos de código reutilizáveis que realizam tarefas específicas. A função `print` foi usada para exibir informações no console ou terminal.

Para executar o código Python:

1. Escreva o código em um editor de texto.
2. Salve o arquivo como ".py".
3. Abra um terminal ou prompt de comando.
4. Navegue até o diretório do arquivo.
5. Execute com `python nome_do_arquivo.py`.
6. Observe a saída no terminal.

```
python soma.py
```

A saída será o número 3, resultado da soma de 1 e 2.

1.4 Como um programa em Python funciona?

O processo de execução de um código Python no computador é composto por várias etapas, cada uma desempenhando um papel fundamental. Tudo começa com o desenvolvimento do código fonte, usualmente armazenado em arquivos com extensão ".py". Esse código é então submetido ao interpretador Python.

O interpretador é o programa encarregado de ler e processar o código fonte. Ele converte o código Python em código objeto (bytecode), uma forma intermediária que é independente da arquitetura do hardware. Este bytecode é uma representação de baixo nível que servirá como entrada para a Máquina Virtual Python (PVM).

A PVM é a camada que efetivamente executa o programa. Ela gerencia a execução do bytecode, cuida do gerenciamento de memória e interage com o sistema operacional. Além disso, algumas implementações do interpretador Python, como o CPython, podem incorporar um Compilador Just-In-Time (JIT).

1.5 Estrutura básica de um programa em Python

Um programa em Python segue uma estrutura básica, um algoritmo, que é uma sequência de passos definidos para realizar uma tarefa. Similar a uma receita culinária, o algoritmo tem entrada (dados), processamento (passos a seguir), e saída (resultado).

Conceitualmente, a estrutura fundamental de um programa Python é:

```
def main():
    # Bloco de código principal

if __name__ == "__main__":
    main()
```

- `main()`: Função principal do programa, chamada quando o programa é executado.
- Bloco de código principal: Onde a execução do programa ocorre, identado para indicar que faz parte do bloco principal.
- `if __name__ == "__main__":`: Verifica se o arquivo está sendo executado como um programa principal, garantindo que a função `main()` seja executada.

Exemplo de um programa básico:

```
def main():
    print("Hello, world!")

if __name__ == "__main__":
    main()
```

Para executar, salve o código em um arquivo ".py", como "hello_world.py", e no terminal, execute:

```
python hello_world.py
```

Lembre-se da importância da formatação adequada (indentação) para indicar a estrutura do programa. A consistência, preferencialmente utilizando quatro espaços por nível, é recomendada pela PEP 8. Essa prática aprimora a legibilidade e mantém um estilo uniforme.

1.6 Exemplo aprimorado: capturando nome e idade

Vamos aprimorar nosso código inicial para aprender a interagir com o usuário. Neste exemplo, criaremos um programa que solicita e armazena o nome e a idade do usuário, para então exibir essas informações. Siga os passos abaixo:

1. Crie um novo arquivo para armazenar o código do programa.
2. Copie ou digite o seguinte código no arquivo:

   ```
   nome = input("Qual é o seu nome? ")
   idade = input("Qual é a sua idade? ")
   print(nome)
   print(idade)
   ```

   O código utiliza a função `input` para solicitar que o usuário insira seu nome e idade. Os dados fornecidos são armazenados nas variáveis `nome` e `idade`, respectivamente. Em seguida, o programa imprime essas informações na tela.

3. Salve o arquivo e execute-o usando o interpretador Python.
4. O programa solicitará que você insira o nome e a idade.

```
Qual é o seu nome? Ana Maria
Qual é a sua idade? 25
```

5. Após fornecer as informações, o programa imprimirá o nome e a idade na tela.

```
Ana Maria
25
```

Este exemplo ilustra o uso da função `input` para interagir com o usuário, capturando dados e exibindo as informações posteriormente. A utilização de `input` é fundamental para criar programas interativos e dinâmicos.

1.7 Estratégias cientificamente fundamentadas para estudar programação

Estudar programação é uma tarefa desafiadora, mas também gratificante. Com uma abordagem estruturada e focada, é possível aprender os conceitos básicos e avançar rapidamente na carreira de desenvolvimento de software. Aqui estão algumas estratégias respaldadas por princípios de aprendizagem e cognição que podem ajudá-lo a aprender programação de forma mais eficaz:

Repetição espaçada: A repetição espaçada é uma técnica de aprendizado que envolve revisar conceitos em intervalos crescentes, fortalecendo a retenção a longo prazo. Utilize programas de flashcards ou aplicativos de repetição espaçada para implementar essa técnica.

Interleaving: O interleaving é uma técnica que envolve misturar diferentes tópicos durante o estudo. Isso evita a dependência excessiva de um único conceito e promove a aplicação flexível de conhecimentos. Alternar entre tópicos relacionados durante as sessões de estudo é uma prática eficaz.

Prática ativa: Engaje-se em práticas ativas, como resolver problemas e desenvolver projetos. Essa abordagem estimula a aplicação prática do conhecimento, consolidando a compreensão e fortalecendo as habilidades práticas.

Compreensão profunda: Foque na compreensão profunda dos conceitos, indo além da memorização para entender os princípios subjacentes. Concentre-se no "porquê" de algo funcionar, não apenas no "como".

Foco e mínima interrupção: Mantenha sessões de estudo focadas, minimizando interrupções para preservar a profundidade da concentração. Escolha ambientes tranquilos e utilize técnicas de meditação para melhorar a concentração.

Aprendizagem baseada em problemas: Aborde a programação como resolução de problemas, tornando a aprendizagem mais contextual e eficaz. A resolução prática de desafios promove a aplicação real dos conceitos.

Recursos diversificados: Utilize diversos recursos de aprendizagem, como livros, vídeos, tutoriais interativos e cursos online. Essa variedade enriquece a compreensão dos tópicos.

Espaço para reflexão: Reserve tempo após o estudo para reflexão, consolidando o conhecimento e aplicando-o a novas situações. Escrever um diário de aprendizado ou discutir com um mentor pode ser útil.

Revisão regular: Mantenha o conhecimento fresco na memória por meio de revisões regulares. Crie um cronograma ou utilize aplicativos de revisão para identificar lacunas na compreensão.

Aprendizagem colaborativa: Participe de grupos de estudo, fóruns online ou projetos colaborativos. A aprendizagem colaborativa proporciona feedback valioso e a oportunidade de compartilhar ideias.

Ao seguir essas estratégias, você pode aumentar suas chances de sucesso no aprendizado de programação e desenvolver uma base sólida de habilidades.

CHAPTER

TWO

CAPÍTULO 2: VÁRIÁVEIS E TIPOS DE DADOS

2.1 Definindo Variáveis e Tipos de Dados

Uma variável é um identificador que representa um valor. Elas são utilizadas para armazenar informações como nomes, números ou textos. A declaração de uma variável é feita usando o operador de atribuição (=). Por exemplo, a linha a seguir cria uma variável chamada `nome` e atribui o valor "Maria" a ela:

```
nome = "Maria"
```

Para acessar o valor armazenado em uma variável, basta utilizar o seu nome. O código a seguir, por exemplo, imprime o valor da variável `nome`:

```
print(nome)
```

Isso resultará na saída:

```
Maria
```

2.2 Tipos de dados Básicos

2.2.1 Números Inteiros

O tipo inteiro é utilizado para representar números inteiros, tanto positivos quanto negativos, como 1, 2, 3, -1, -2, -3, entre outros. Esses valores são empregados para expressar quantidades e índices em programação, desempenhando um papel fundamental em diversas aplicações.

Exemplos de uso de inteiros incluem:

```
# Declaração de uma variável do tipo inteiro
x = 10

# Atribuição de um valor a uma variável do tipo inteiro
x = 20

# Soma de dois inteiros
print(x + 10)
```

```
30
```

2.2.2 Números de Ponto Flutuante

O tipo de dado ponto flutuante, representado pelo tipo float em Python, é utilizado para expressar valores decimais, como 1.0, 2.5, 3.14, -1.0, -2.5, -3.14, entre outros. Esses valores são apropriados para representar grandezas que envolvem medidas, preços, e outras grandezas fracionárias.

Aqui estão alguns exemplos de uso de números de ponto flutuante:

```
# Declaração de uma variável do tipo float
y = 3.14

# Atribuição de um valor a uma variável do tipo float
y = 2.5

# Subtração de dois floats
print(y - 1.0)
```

```
1.5
```

2.2.3 Strings

O tipo de dado string epresenta sequências de caracteres, como “Olá, pessoal!”, “123”, “abc”, etc. Essas estruturas são comumente utilizadas para representar texto e nomes.

Aqui estão alguns exemplos de uso de strings:

```
# Declaração de uma variável do tipo string
z = "Olá, pessoal!"

# Atribuição de um valor a uma variável do tipo string
z = "123"

# Concatenação de duas strings
print(z + ", como vai?")
```

```
123, como vai?
```

Em Python, tanto as aspas simples (`'`) quanto as aspas duplas (`"`) podem ser usadas para definir literais de strings. Não há diferença funcional entre elas; você pode usar qualquer uma delas de forma intercambiável para criar strings. A escolha entre as aspas simples e duplas é principalmente uma questão de estilo e preferência pessoal. Por exemplo:

```
aspas_simples = 'Esta é uma string com aspas simples.'
aspas_duplas = "Esta é uma string com aspas duplas."
```

Você pode usar um tipo de aspas para definir uma string que contenha o outro tipo de aspas sem problemas. Por exemplo:

```
com_outras_aspas = "Esta string contém 'aspas simples' dentro dela."
```

Ou:

```
com_outras_aspas = 'Esta string contém "aspas duplas" dentro dela.'
```

Ambos os exemplos são válidos, e o Python permite que você use aspas simples dentro de uma string delimitada por aspas duplas e vice-versa.

2.2.4 Booleanos

Os tipos booleanos em Python representam valores lógicos, ou seja, podem ser True ou False. Esses valores são comumente utilizados para representar condições, resultados de testes e expressões lógicas.

Aqui estão alguns exemplos de uso de booleanos:

```
# Declaração de uma variável do tipo booleano
a = True

# Atribuição de um valor a uma variável do tipo booleano
a = False

# Comparação de dois valores
print(10 > 20)
```

```
False
```

2.3 Tipos de dados compostos

2.3.1 Listas

As Listas são um tipo de dados mutável, o que significa que podem ser alteradas após serem criadas. Elas são declaradas utilizando colchetes, e os valores são separados por vírgulas. Listas podem conter valores de qualquer tipo, incluindo inteiros, números de ponto flutuante, strings, booleanos e outros tipos de dados.

Aqui estão alguns exemplos de uso de listas:

```
# Declaração de uma lista com três valores inteiros
lista1 = [1, 2, 3]

# Declaração de uma lista com três valores reais
lista2 = [1.0, 2.5, 3.14]

# Declaração de uma lista com três valores strings
lista3 = ["Estou", "programando", "!"]

# Declaração de uma lista com três valores booleanos
lista4 = [True, False, True]
```

As listas podem ser utilizadas para representar uma ampla variedade de dados. Por exemplo, podemos usar listas para representar listas de compras, listas de tarefas, etc.

Aqui estão alguns exemplos específicos de uso de listas:

Para representar uma lista de compras, podemos usar uma lista:

```
lista_de_compras = ["pão", "leite", "ovos"]
```

Representando uma lista de tarefas, podemos usar uma lista:

```
lista_de_tarefas = ["lavar a louça", "varrer a casa", "estudar python"]
```

Para representar uma lista de números, podemos usar uma lista:

```
lista_de_numeros = [1, 2, 3, 10, 15]
```

Uma mesma lista também pode receber tipos diversos em sua atribuição. É comum a necessidade dessa mistura de tipos para solução de alguns tipos de problemas.

```
lista_mista_tipos = [1, 2.5, "Estou", True, [1, 2, 3]]
```

Como pode ser visto, a lista `lista_mista_tipos` contém um inteiro, um número de ponto flutuante, uma string, um booleano e uma lista aninhada.

Vamos agora conhecer os principais métodos utilizados quando trabalhamos com listas. Mas antes, o que são métodos em Python?

Em termos simples, métodos em Python são como instruções especiais que dizemos a uma lista para ela realizar tarefas específicas. Cada método tem uma função específica, como adicionar, remover ou organizar elementos na lista.

```
# Criando a Lista inicial
lista_mista = [1, 2.5, "Estou", True]

# Adiciona um elemento ao final da lista
lista_mista.append(4)
print(lista_mista)

# Adiciona vários elementos ao final da lista
lista_mista.extend([5, 6, 7])
print(lista_mista)

# Insere um elemento em uma posição específica da lista
lista_mista.insert(2, "Olá")
print(lista_mista)

# Remove um elemento da lista
lista_mista.remove("Estou")
print(lista_mista)

# Remove o último elemento da lista
lista_mista.pop()
print(lista_mista)

# Conta quantas vezes um elemento aparece na lista
ocorrencias = lista_mista.count(2.5)
print(ocorrencias)

# Inverte a ordem da lista
lista_mista.reverse()
print("Lista invertida:", lista_mista)
```

```
[1, 2.5, 'Estou', True, 4]
[1, 2.5, 'Estou', True, 4, 5, 6, 7]
[1, 2.5, 'Olá', 'Estou', True, 4, 5, 6, 7]
[1, 2.5, 'Olá', True, 4, 5, 6, 7]
[1, 2.5, 'Olá', True, 4, 5, 6]
1
Lista invertida: [6, 5, 4, True, 'Olá', 2.5, 1]
```

Em Python, os elementos em uma lista são organizados em uma sequência numerada chamada índice. O índice é como

a posição de cada elemento na lista, começando do zero para o primeiro elemento. Por exemplo, em uma lista `[1, 2, 3, 4]`, o número 1 está no índice 0, o 2 no índice 1, e assim por diante.

Como Funciona:

- **Indexação Positiva:** Você pode acessar elementos contando a partir do início da lista. O primeiro elemento tem índice 0, o segundo tem índice 1, e assim por diante.
- **Indexação Negativa:** Também é possível contar a partir do final da lista. O último elemento tem índice -1, o penúltimo tem índice -2, e assim por diante.

Como Utilizar:

- Para acessar um elemento em uma lista, você utiliza o operador de colchetes `[]` com o índice desejado. Por exemplo, `lista[2]` retorna o terceiro elemento da lista.
- Para modificar um elemento, você pode usar a mesma notação de índice. Por exemplo, `lista[1] = 10` atribui o valor 10 ao segundo elemento da lista.

Aqui está um exemplo prático:

```
# Lista de exemplo
numeros = [10, 20, 30, 40, 50]

# Acessando elementos
print("Primeiro elemento:", numeros[0])
print("Último elemento:", numeros[-1])

# Modificando elementos
numeros[2] = 35
print("Lista modificada:", numeros)
```

```
Primeiro elemento: 10
Último elemento: 50
Lista modificada: [10, 20, 35, 40, 50]
```

Lembre-se de que os índices devem estar dentro da faixa válida da lista para evitar erros.

2.3.2 Tuplas

As tuplas são uma sequência de valores imutáveis. Isso significa que, uma vez criadas, as tuplas não podem ser alteradas. As tuplas são declaradas usando parênteses, com os valores separados por vírgulas. Por exemplo, a seguinte declaração cria uma tupla com três valores:

```
tupla = (1, 2, 3)
```

As tuplas podem conter valores de qualquer tipo, incluindo inteiros, reais, strings, booleanos, etc.

Aqui estão alguns exemplos de uso:

```
# Declaração de uma tupla com três valores inteiros
inteiros = (1, 2, 3)

# Declaração de uma tupla com três valores reais
reais = (1.0, 2.5, 3.14)

# Declaração de uma tupla com três valores strings
```

(continues on next page)

(continued from previous page)

```
strings = ("Olá", "mundo", "!")

# Declaração de uma tupla com três valores booleanos
booleanos = (True, False, True)

# Declaração de uma tupla com valores de tipos diferentes
tupla_mista = (True, 19, 7.5, "a")
```

As tuplas são um tipo de dados útil para representar dados que não precisam ser alterados. Por exemplo, podemos usar tuplas para representar coordenadas, dados de identificação, etc.

Aqui estão alguns exemplos específicos de uso de tuplas em Python:

Para representar as coordenadas de um ponto no plano cartesiano, podemos usar uma tupla:

```
ponto = (1.0, 2.0)
```

Representaando o número de identificação de um funcionário, podemos usar uma tupla:

```
identificacao = (1234567890, "João da Silva")
```

Para representar os dados de um produto, podemos usar uma tupla:

```
produto = ("Camisa", "P", 100.0)
```

Índices em Tuplas

Assim como em listas, as tuplas também utilizam índices para acessar seus elementos. Os índices em tuplas começam do 0 para o primeiro elemento e seguem uma sequência numérica.

Exemplo de Acesso por Índice:

```
# Declaração de uma tupla
tupla = (10, 20, 30, 40, 50)

# Acessando elementos por índice
primeiro_elemento = tupla[0]
terceiro_elemento = tupla[2]

print("Primeiro elemento:", primeiro_elemento)
print("Terceiro elemento:", terceiro_elemento)
```

```
Primeiro elemento: 10
Terceiro elemento: 30
```

Observações: Os índices negativos funcionam da mesma forma que em listas, onde -1 refere-se ao último elemento, -2 ao penúltimo, e assim por diante.

```
# Acessando o último elemento por índice negativo
ultimo_elemento = tupla[-1]
print("Último elemento:", ultimo_elemento)
```

```
Último elemento: 50
```

Índices em Tuplas Mistas

```
# Declaração de uma tupla com valores de tipos diferentes
tupla_mista = (True, 19, 7.5, "a")

# Acessando elementos por índice
primeiro_elemento_misto = tupla_mista[0]
ultimo_elemento_misto = tupla_mista[-1]

print("Primeiro elemento misto:", primeiro_elemento_misto)
print("Último elemento misto:", ultimo_elemento_misto)
```

```
Primeiro elemento misto: True
Último elemento misto: a
```

Os índices em tuplas permitem acessar e trabalhar com os elementos individualmente, facilitando a manipulação dessas estruturas de dados imutáveis em Python.

Assim como as listas, as tuplas em Python possuem alguns métodos especiais que podem ser utilizados para realizar operações específicas. No entanto, devido à imutabilidade das tuplas (não é possível modificar uma tupla após sua criação), esses métodos são mais limitados em comparação com os disponíveis para listas.

Aqui estão alguns dos métodos especiais comumente utilizados em tuplas:

- **count(valor):** Retorna o número de ocorrências do valor especificado na tupla.

  Exemplo:

```
minha_tupla = (1, 2, 2, 3, 4, 2)
numero_de_dois = minha_tupla.count(2)
print(numero_de_dois)
```

```
3
```

- **index(valor[, start[, stop]]):** Retorna o índice da primeira ocorrência do valor especificado. Você pode opcionalmente fornecer os argumentos `start` e `stop` para limitar a busca a uma sub-tupla.

  Exemplo:

```
minha_tupla = (1, 2, 3, 4, 5)
indice_do_tres = minha_tupla.index(3)
print(indice_do_tres)
```

```
2
```

Lembre-se de que, devido à imutabilidade das tuplas, métodos que alteram o conteúdo (como `append`, `extend`, `remove`, `pop`, `insert`, etc.) não estão disponíveis para tuplas.

2.3.3 Dicionários

Os dicionários servem para armazenar informações por meio de pares de chave e valor. Cada elemento do dicionário consiste em uma chave única associada a um valor correspondente.

Para criar um dicionário, utilizamos chaves { } e separamos cada par chave-valor por vírgulas. Exemplo:

```
dicionario = {"nome": "João", "idade": 30, "cidade": "Natal"}
```

- **Chaves:** São rótulos exclusivos que acessam os valores associados.
- **Tipos de Chaves:** Podem ser de qualquer tipo de dado imutável, como strings, números inteiros ou reais.
- **Tipos de Valores:** Podem ser de qualquer tipo de dado, inclusive listas ou tuplas.

Os dicionários são úteis quando queremos associar informações relacionadas entre si. Por exemplo, no dicionário acima, "nome" é a chave associada ao valor "João". Essa estrutura flexível e poderosa é amplamente utilizada em Python para representar dados estruturados de forma eficiente.

Exemplos de Uso:

```
# Acesso a um valor do dicionário
nome = dicionario["nome"]
print("Nome:", nome)

# Alteração de um valor do dicionário
dicionario["idade"] = 31
print("Idade atualizada:", dicionario["idade"])

# Remoção de um valor do dicionário
del dicionario["cidade"]
print("Dicionário após remoção:", dicionario)
```

Acesso a Elementos Individualmente:

Podemos acessar cada valor individualmente no dicionário utilizando suas chaves:

```
# Acesso a elementos individualmente
telefone = contato["telefone"]
print("Telefone de contato:", telefone)

# Acesso a configuração de idioma
idioma_config = configuracoes["idioma"]
print("Configuração de idioma:", idioma_config)

# Acesso à cor do produto
cor_produto = produto["cor"]
print("Cor do produto:", cor_produto)
```

A capacidade de acessar elementos individualmente nos dicionários permite recuperar informações específicas de maneira direta e eficiente. Essa característica torna os dicionários uma estrutura de dados poderosa para representar e manipular dados em Python.

Os dicionários oferecem uma variedade de métodos para manipular e acessar os dados armazenados. Aqui estão alguns dos principais métodos de dicionários:

`clear()`: Remove todos os itens do dicionário.

```
meu_dicionario = {"nome": "João", "idade": 25}
meu_dicionario.clear()
print(meu_dicionario)
```

```
{}
```

`copy()`: Retorna uma cópia do dicionário.

```
meu_dicionario = {"nome": "Maria", "idade": 30}
copia_dicionario = meu_dicionario.copy()
print(copia_dicionario)
```

```
{'nome': 'Maria', 'idade': 30}
```

`get(chave[, valor_padrão])`: Retorna o valor associado à chave especificada. Se a chave não existir, retorna um valor padrão (ou `None` se não fornecido).

```
meu_dicionario = {"nome": "Carlos", "idade": 28}
idade = meu_dicionario.get("idade")
print(idade)
```

```
28
```

`items()`: Retorna uma lista de tuplas contendo pares chave-valor.

```
meu_dicionario = {"nome": "Ana", "idade": 35}
itens = meu_dicionario.items()
print(itens)
```

```
dict_items([('nome', 'Ana'), ('idade', 35)])
```

`keys()`: Retorna uma lista contendo todas as chaves do dicionário.

```
meu_dicionario = {"nome": "Lucas", "idade": 22}
chaves = meu_dicionario.keys()
print(chaves)
```

```
dict_keys(['nome', 'idade'])
```

`values()`: Retorna uma lista contendo todos os valores do dicionário.

```
meu_dicionario = {"nome": "Julia", "idade": 27}
valores = meu_dicionario.values()
print(valores)
```

```
dict_values(['Julia', 27])
```

pop(chave[, valor_padrão]): Remove e retorna o valor associado à chave especificada. Se a chave não existir, retorna um valor padrão (ou gera um erro se não fornecido).

```
meu_dicionario = {"nome": "Pedro", "idade": 32}
idade = meu_dicionario.pop("idade")
print(idade)
```

```
32
```

popitem(): Remove e retorna o último par chave-valor do dicionário como uma tupla.

```
meu_dicionario = {"nome": "Fernanda", "idade": 29}
ultimo_item = meu_dicionario.popitem()
print(ultimo_item)
```

```
('idade', 29)
```

update(dicionario): Atualiza o dicionário com pares chave-valor de outro dicionário ou iterável.

```
meu_dicionario = {"nome": "Rafael", "idade": 26}
outro_dicionario = {"cidade": "São Paulo"}
meu_dicionario.update(outro_dicionario)
print(meu_dicionario)
```

```
{'nome': 'Rafael', 'idade': 26, 'cidade': 'São Paulo'}
```

Estes são apenas alguns dos métodos disponíveis para dicionários em Python. A escolha do método dependerá da operação específica que você deseja realizar.

2.3.4 Conjuntos

Os conjuntos são uma estrutura de dados em Python que permite armazenar uma coleção de elementos únicos. Isso significa que, cada elemento de um conjunto deve ser diferente de todos os outros elementos do conjunto.

Os conjuntos são declarados usando chaves, com os elementos separados por vírgulas. Por exemplo, a seguinte declaração cria um conjunto com três elementos:

```
conjunto = {1, 2, 3}
```

Os elementos dos conjuntos podem ser de qualquer tipo de dados, incluindo inteiros, reais, strings, booleanos, etc. Aqui estão alguns exemplos de uso de conjuntos em Python:

```
# Declaração de um conjunto com três elementos
conjunto = {1, 2, 3}

# Adição de um elemento
conjunto.add(4)

# Remoção de um elemento
```

(continues on next page)

(continued from previous page)

```
conjunto.remove(2)

print(conjunto)
```

```
{1, 3, 4}
```

```
# União de conjuntos
conjunto_1 = {1, 2, 3}
conjunto_2 = {4, 5, 6}
conjunto_uniao = conjunto_1 | conjunto_2
print(conjunto_uniao)

# Interseção de conjuntos
conjunto_1 = {1, 2, 3}
conjunto_2 = {2, 3, 4}
conjunto_intersecao = conjunto_1 & conjunto_2
print(conjunto_intersecao)

# Diferença de conjuntos
conjunto_1 = {1, 2, 3}
conjunto_2 = {2, 3, 4}
conjunto_diferenca = conjunto_1 - conjunto_2
print(conjunto_diferenca)
```

```
{1, 2, 3, 4, 5, 6}
{2, 3}
{1}
```

Os conjuntos são um tipo de dados que pode ser usado para representar uma ampla gama de dados. Por exemplo, podemos usar conjuntos para representar coleções de números, letras, etc.

Aqui estão alguns exemplos específicos de uso de conjuntos em Python. Para representar uma coleção de números primos, podemos usar um conjunto:

```
primos = {2, 3, 5, 7, 11, 13, 17, 19}
```

Para representar uma coleção de letras do alfabeto, podemos usar um conjunto:

```
letras = {"a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o",
 ↪"p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"}
```

Para representar uma coleção de cores, podemos usar um conjunto:

```
cores = {"vermelho", "verde", "amarelo", "azul", "roxo", "preto", "branco"}
```

Exemplos de nomes didáticos para declarações de conjuntos:

```
# Declaração de um conjunto com três elementos
conjunto_de_numeros = {1, 2, 3}

# Declaração de um conjunto com as cores do arco-íris
conjunto_de_cores = {"vermelho", "laranja", "amarelo", "verde", "azul", "anil",
 ↪"violeta"}
```

A escolha do nome ideal para um conjunto depende do contexto em que ele será usado.

2.4 Exercícios

1. Declare duas variáveis, uma do tipo int e outra do tipo float. Em seguida, atribuir a elas valores e imprimir o resultado da soma, da subtração, da multiplicação e da divisão entre as duas variáveis.
2. Declarar uma variável do tipo str e atribuir a ela uma string. Em seguida, imprimir o comprimento da string, a primeira letra da string, a última letra da string e a string invertida.
3. Declare uma variável do tipo bool e atribuir a ela um valor. Em seguida, imprimir o resultado da operação de negação sobre a variável.
4. Faça um programa que calcule o Índice de Massa Corporal (IMC) de uma pessoa. O IMC é calculado dividindo-se o peso da pessoa pela sua altura ao quadrado. O IMC é uma medida da relação entre o peso e a altura de uma pessoa. O programa deve imprimir o IMC da pessoa, classificando-o de acordo com a tabela abaixo:

IMC	Classificação
< 18,5	Abaixo do peso
18,5 - 24,9	Saudável
25,0 - 29,9	Sobrepeso
30,0 - 34,9	Obesidade grau I
35,0 - 39,9	Obesidade grau II

   = 40,0 | Obesidade grau III

5. Crie uma lista chamada `frutas` com as seguintes frutas: maçã, banana, laranja, pera e melancia. Em seguida, imprima a lista.
6. Criar uma tupla chamada `coordenadas` com as coordenadas (2, 3). Em seguida, imprima as coordenadas.
7. Crie um dicionário chamado contato. Peça ao usuário para fornecer os dados para as chaves “nome”, “telefone” e “endereço”. Em seguida, imprima o conteúdo do dicionário.

Exemplo de execução:

```
Digite o nome do contato: Python da Silva
Digite o telefone do contato: (84) 9999-9999
Digite o endereço do contato: Rua da Programação, 123

Nome: Python da Silva, Telefone: (84) 9999-9999, Endereço: Rua da Programação, 123.
```

8. Crie um conjunto chamado `numeros` com os números 1, 2, 3, 4 e 5. Imprima o conjunto.
9. Criar dois conjuntos, `conjunto_a` e `conjunto_b`, com alguns números. Realizar as seguintes operações e imprimir os resultados:
   - União: Combinar os elementos únicos de ambos os conjuntos.
   - Interseção: Encontrar os elementos que estão presentes em ambos os conjuntos.
   - Diferença: Identificar os elementos que estão em `conjunto_a` mas não em `conjunto_b`.
10. Criar uma string original, por exemplo, “Python é uma linguagem de programação poderosa!”. Realizar as seguintes operações e imprimir os resultados:

- Transformar em maiúsculas: Converter toda a string para letras maiúsculas.
- Transformar em minúsculas: Converter toda a string para letras minúsculas.

- Substituir parte da string: Substituir uma parte específica da string por outra (por exemplo, substituir “poderosa” por “versátil”).

CHAPTER

THREE

CAPÍTULO 3: OPERADORES E EXPRESSÕES

Compreender os operadores e expressões em Python é essencial para capacitar programadores na manipulação eficiente de dados e na execução de operações fundamentais. Esses operadores, que incluem aritméticos, relacionais, lógicos e de atribuição, desempenham papéis específicos, permitindo desde cálculos matemáticos até avaliações condicionais. O domínio desses conceitos não apenas facilita a escrita de códigos mais eficazes, mas também é vital para a resolução de problemas complexos e a compreensão de códigos existentes. Em resumo, o conhecimento aprofundado dos operadores e expressões em Python é uma habilidade fundamental para o desenvolvimento eficiente de algoritmos e lógica de programação.

3.1 Operadores Aritméticos

- + (adição): Soma dois valores.
- − (subtração): Subtrai o operando direito do operando esquerdo.
- * (multiplicação): Multiplica dois valores.
- / (divisão): Divide o operando esquerdo pelo operando direito.
- % (módulo): Retorna o resto da divisão do operando esquerdo pelo operando direito.
- // (divisão de piso): Retorna a parte inteira da divisão do operando esquerdo pelo operando direito.
- ** (potência): Eleva o operando esquerdo à potência do operando direito.

```
# Operadores Aritméticos em Python

# Adição (+): Soma dois valores.
soma = 5 + 3
print("Adição:", soma)

# Subtração (-): Subtrai o operando direito do operando esquerdo.
subtracao = 7 - 4
print("Subtração:", subtracao)

# Multiplicação (*): Multiplica dois valores.
multiplicacao = 6 * 2
print("Multiplicação:", multiplicacao)

# Divisão (/): Divide o operando esquerdo pelo operando direito.
divisao = 10 / 3
print("Divisão:", divisao)

# Módulo (%): Retorna o resto da divisão do operando esquerdo pelo operando direito.
```

(continues on next page)

(continued from previous page)

```
modulo = 15 % 4
print("Módulo:", modulo)

# Divisão de Piso (//): Retorna a parte inteira da divisão do operando esquerdo pelo␣
↪operando direito.
divisao_piso = 20 // 3
print("Divisão de Piso:", divisao_piso)

# Potência (**): Eleva o operando esquerdo à potência do operando direito.
potencia = 2 ** 4
print("Potência:", potencia)
```

```
Adição: 8
Subtração: 3
Multiplicação: 12
Divisão: 3.3333333333333335
Módulo: 3
Divisão de Piso: 6
Potência: 16
```

3.2 Operadores Relacionais

- == (igual a): Retorna True se os operandos forem iguais.
- != (diferente de): Retorna True se os operandos não forem iguais.
- < (menor que): Retorna True se o operando esquerdo for menor que o operando direito.
- > (maior que): Retorna True se o operando esquerdo for maior que o operando direito.
- <= (menor ou igual a): Retorna True se o operando esquerdo for menor ou igual ao operando direito.
- >= (maior ou igual a): Retorna True se o operando esquerdo for maior ou igual ao operando direito.

```
# Operadores Relacionais em Python

# Igual a (==): Retorna True se os operandos forem iguais.
igual_a = 5 == 5
print("Igual a:", igual_a)

# Diferente de (!=): Retorna True se os operandos não forem iguais.
diferente_de = 6 != 5
print("Diferente de:", diferente_de)

# Menor que (<): Retorna True se o operando esquerdo for menor que o operando direito.
menor_que = 4 < 7
print("Menor que:", menor_que)

# Maior que (>): Retorna True se o operando esquerdo for maior que o operando direito.
maior_que = 8 > 5
print("Maior que:", maior_que)

# Menor ou igual a (<=): Retorna True se o operando esquerdo for menor ou igual ao␣
↪operando direito.
```

(continues on next page)

(continued from previous page)

```
menor_ou_igual_a = 5 <= 5
print("Menor ou igual a:", menor_ou_igual_a)

# Maior ou igual a (>=): Retorna True se o operando esquerdo for maior ou igual ao␣
↪operando direito.
maior_ou_igual_a = 7 >= 6
print("Maior ou igual a:", maior_ou_igual_a)
```

```
Igual a: True
Diferente de: True
Menor que: True
Maior que: True
Menor ou igual a: True
Maior ou igual a: True
```

3.3 Operadores Lógicos

- `and` (e lógico): Retorna True se ambos os operandos forem True.
- `or` (ou lógico): Retorna True se pelo menos um dos operandos for True.
- `not` (negação lógica): Inverte o valor do operando.

```
# Operadores Lógicos em Python

# E lógico (and): Retorna True se ambos os operandos forem True.
resultado_and_1 = True and True
resultado_and_2 = True and False
resultado_and_3 = False and False

# Ou lógico (or): Retorna True se pelo menos um dos operandos for True.
resultado_or_1 = True or True
resultado_or_2 = True or False
resultado_or_3 = False or False

# Negação lógica (not): Inverte o valor do operando.
resultado_not_1 = not True
resultado_not_2 = not False

# Exibição dos resultados
print("E lógico (and):", resultado_and_1, resultado_and_2, resultado_and_3)
print("Ou lógico (or):", resultado_or_1, resultado_or_2, resultado_or_3)
print("Negação lógica (not):", resultado_not_1, resultado_not_2)
```

```
E lógico (and): True False False
Ou lógico (or): True True False
Negação lógica (not): False True
```

3.4 Operadores de Atribuição

- = (atribuição): Atribui o valor do operando direito ao operando esquerdo.
- +=, -=, *=, /=, %= (atribuições com operações aritméticas): Realiza a operação aritmética indicada e atribui o resultado à variável à esquerda.

```
# Operadores de Atribuição em Python

# Atribuição (=): Atribui o valor do operando direito ao operando esquerdo.
x = 5
y = x

# Atribuições com operações aritméticas: Realiza a operação aritmética e atribui o␣
↪resultado à variável à esquerda.
a = 10
a += 2  # Equivalente a: a = a + 2

b = 7
b -= 3  # Equivalente a: b = b - 3

c = 3
c *= 5  # Equivalente a: c = c * 5

d = 8
d /= 2  # Equivalente a: d = d / 2

e = 15
e %= 4  # Equivalente a: e = e % 4

# Exibição dos valores após as operações
print("Atribuição (=):", x, y)
print("Atribuição com soma (+=):", a)
print("Atribuição com subtração (-=):", b)
print("Atribuição com multiplicação (*=):", c)
print("Atribuição com divisão (/=):", d)
print("Atribuição com módulo (%=):", e)
```

```
Atribuição (=): 5 5
Atribuição com soma (+=): 12
Atribuição com subtração (-=): 4
Atribuição com multiplicação (*=): 15
Atribuição com divisão (/=): 4.0
Atribuição com módulo (%=): 3
```

3.5 Operadores de Incremento e Decremento

Em Python, ao contrário de algumas outras linguagens de programação, não há operadores de incremento (++) e decremento (−−) específicos. No entanto, você pode alcançar o mesmo efeito usando operadores de atribuição e aritméticos. Vamos ver como isso pode ser feito:

3.5.1 Operador de Incremento

Para incrementar o valor de uma variável em 1, você pode usar o operador de adição (+=):

```
# Exemplo de incremento
contador = 5
contador += 1  # Incrementa o valor de contador em 1
print(contador)
```

```
6
```

3.5.2 Operador de Decremento

Para decrementar o valor de uma variável em 1, você pode usar o operador de subtração (−=):

```
# Exemplo de decremento
contador = 8
contador -= 1  # Decrementa o valor de contador em 1
print(contador)
```

```
7
```

Esses exemplos ilustram como realizar operações de incremento e decremento em Python. A sintaxe += é uma forma concisa de escrever "atribua à variável o valor atual da variável mais alguma quantidade". O mesmo princípio se aplica ao operador −= para decremento.

3.6 Operadores Unários

Os operadores unários atuam em um único operando, ou seja, em um único valor. Eles realizam operações sobre esse valor. Vamos ver alguns exemplos:

3.6.1 Operador Negativo (–):

Muda o sinal do número para negativo.

```
x = 5
resultado = -x
```

```
-5
```

3.6.2 Operador Positivo (+):

Mantém o sinal do número (raramente usado, já que os números são positivos por padrão).

```
y = -3
resultado = +y  # O valor de resultado é sem mudança de sinal.
```

```
-3
```

3.6.3 Operador de Inversão Bit a Bit (~):

Inverte cada bit do número.

```
z = 7
resultado = ~z
```

```
-8
```

O operador negativo e o operador positivo lidam com números inteiros e de ponto flutuante, enquanto o operador de inversão bit a bit opera em valores inteiros.

3.7 Exercícios

1. Dada a expressão matemática: $(a = 4 \times (2 + 3))$, crie uma variável chamada `a` e atribua a ela o resultado dessa expressão. Imprima o valor de `a`.
2. Escreva um programa em Python que recebe dois números do usuário, realiza a soma desses números e exibe o resultado.
3. Calcule o resto da divisão de 17 por 5 e armazene o resultado em uma variável chamada `resto`. Imprima o valor de `resto`.
4. Crie uma expressão lógica que seja verdadeira se um número for par e maior que 10. Teste a expressão com diferentes valores e imprima os resultados.
5. Dada a variável `preco_produto` com o valor 150, aplique um desconto de 20% utilizando operadores aritméticos e de atribuição. Imprima o novo valor.
6. Implemente um programa que recebe a idade de uma pessoa e verifica se ela é maior de idade (idade maior ou igual a 18). Exiba a mensagem adequada.
7. Crie uma expressão lógica que seja verdadeira apenas se um número for ímpar ou menor que 5. Teste a expressão com diferentes valores e imprima os resultados.
8. Dado o raio de um círculo, calcule a área utilizando o operador de potência (**) para elevar o raio ao quadrado. Imprima o resultado.
9. Escreva um programa que converta uma temperatura de Celsius para Fahrenheit. Utilize a fórmula $(F = \frac{9}{5}C+32)$.
10. Dada a expressão $(x = 3)$ e $(y = 5)$, crie uma variável chamada `resultado` que armazene o valor da expressão $(x^2 + y^2)$ e imprima o resultado.

CHAPTER

FOUR

CAPÍTULO 4: CONTROLE DE FLUXO

4.1 Estruturas condicionais

As estruturas condicionais permitem ao programador tomar decisões sobre o fluxo de execução do programa. As estruturas condicionais mais comuns em Python são:

- **`if`:** Testa uma condição e executa um bloco de código se a condição for verdadeira.
- **`elif`:** Testa uma condição e executa um bloco de código se a condição for verdadeira, mas apenas se a condição anterior for falsa.
- **`else`:** Executa um bloco de código se nenhuma das condições anteriores for verdadeira.

As estruturas condicionais em Python, especificamente as instruções `if`, `elif`, e `else`, são essenciais para tomar decisões em um programa com base em diferentes condições.

4.1.1 A Instrução `if`

A instrução `if` é utilizada para executar um bloco de código se uma condição especificada for avaliada como verdadeira. Vejamos um exemplo prático:

```
idade = 20

if idade >= 18:
    print("Você é maior de idade.")
```

Neste exemplo, o bloco de código dentro do `if` só será executado se a variável `idade` for maior ou igual a 18. Caso contrário, o bloco será ignorado.

4.1.2 Instruções `elif` e `else`

Quando lidamos com múltiplas condições, as instruções `elif` (abreviação de “else if”) e `else` podem ser empregadas.

Exemplo com `elif`:

```
idade = 16

if idade < 18:
    print("Você é menor de idade.")
elif idade == 18:
    print("Você acabou de atingir a maioridade.")
else:
    print("Você é maior de idade.")
```

Neste exemplo, o programa verifica a idade e imprime uma mensagem apropriada com base nas condições. Se a idade for menor que 18, imprime "Você é menor de idade". Se a idade for exatamente 18, imprime "Você acabou de atingir a maioridade". Caso contrário, o bloco dentro do `else` é executado, imprimindo "Você é maior de idade".

4.1.3 Exemplo Prático: Verificação de Números Pares e Ímpares

Vamos criar um exemplo mais prático usando estruturas condicionais para verificar se um número é par ou ímpar:

```
numero = 15

if numero % 2 == 0:
    print(f"{numero} é um número par.")
else:
    print(f"{numero} é um número ímpar.")
```

```
15 é um número ímpar.
```

Neste exemplo, o operador `%` calcula o resto da divisão por 2. Se o resto for zero, o número é par; caso contrário, é ímpar.

Observação: O trecho `print(f"{numero} é um número ímpar.")` utiliza uma f-string para criar uma string formatada, onde `{numero}` é substituído pelo valor atual da variável `numero`.

4.1.4 Aninhamento de Estruturas Condicionais

É possível aninhar instruções `if` dentro de outras instruções `if`, `elif`, ou `else`. Isso permite lidar com condições mais complexas. No entanto, deve-se ter cuidado para não tornar o código muito complexo.

```
idade = 25
sexo = "Feminino"

if idade >= 18:
    print("Você é maior de idade.")

    if sexo == "Feminino":
        print("E também do sexo feminino.")
else:
    print("Você é menor de idade.")
```

Analisando o exemplo:

Avaliação Externa (`if idade >= 18`):

- Se a idade for maior ou igual a 18, o bloco interno é executado.

- A mensagem “Você é maior de idade.” será impressa.

Bloco Interno (`if sexo == "Feminino"`):

- Este bloco só será executado se a condição externa (idade >= 18) for verdadeira.
- Se o sexo for “Feminino”, a mensagem “E também do sexo feminino.” será impressa.

`else` Externo:

- Se a condição externa não for atendida (idade < 18), a mensagem “Você é menor de idade.” será impressa.

Agora, como segundo exemplo, considere uma situação em que você está classificando alunos com base em suas notas em uma disciplina:

```
nota = 75

if nota >= 90:
    print("Parabéns! Você obteve uma nota A.")
elif nota >= 80:
    print("Ótimo! Sua nota é B.")
elif nota >= 70:
    print("Bom trabalho! Sua nota é C.")
else:
    print("Infelizmente, você não atingiu a nota mínima. Sua nota é D.")
```

Avaliação da Nota (`if nota >= 90`):

- Se a nota for 90 ou superior, o aluno recebe uma nota A.

`elif nota >= 80:`

- Se a condição anterior não for atendida, mas a nota for 80 ou superior, o aluno recebe uma nota B.

`elif nota >= 70:`

- Se a condição anterior não for atendida, mas a nota for 70 ou superior, o aluno recebe uma nota C.

`else:`

- Se nenhuma das condições anteriores for atendida, o aluno recebe uma nota D.

Esses exemplos mostram como você pode aninhar estruturas condicionais para lidar com várias situações e condições em seu código de maneira organizada.

4.2 Estruturas de repetição

As estruturas de repetição em Python, também conhecidas como loops, são utilizadas para executar um bloco de código várias vezes. Python possui duas principais estruturas de repetição: `for` e `while`. Vamos explorar cada uma delas com teoria e exemplos práticos.

4.2.1 Estrutura de Repetição `for`

A estrutura `for` é uma ferramenta poderosa em Python, projetada para iterar sobre sequências, como listas, tuplas, strings e outros objetos iteráveis. Ela proporciona uma maneira elegante e eficiente de processar cada elemento de uma sequência, executando um bloco de código associado a cada iteração.

Sintaxe

```
for variavel in sequencia:
    # Bloco de código a ser repetido
```

Aqui, `variavel` é uma variável que assume o valor de cada elemento da `sequencia` durante cada iteração do loop. O bloco de código associado é executado para cada valor da sequência.

Exemplo Prático: Iterando sobre uma Lista de Frutas

```
frutas = ["maçã", "banana", "uva"]
for fruta in frutas:
    print(fruta)
```

Saída:

```
maçã
banana
uva
```

Neste exemplo, o loop `for` percorre a lista `frutas` e imprime cada elemento da lista. Isso é particularmente útil ao lidar com conjuntos de dados, como uma lista de itens a serem processados.

Casos de Utilização do `for`

Números em um Intervalo:

```
for i in range(1, 6):
    print(i)
```

Saída:

```
1
2
3
4
5
```

O `range(1, 6)` cria uma sequência de números de 1 a 5, e o `for` itera sobre esses valores.

Iteração sobre uma String:

```
palavra = "Python"
for letra in palavra:
    print(letra)
```

Saída:

```
P
y
t
h
o
n
```

O `for` percorre cada caractere na string `palavra` e imprime-os individualmente.

Iterando sobre Dicionários:

```
aluno_notas = {"Alice": 90, "Bob": 80, "Charlie": 95}
for aluno, nota in aluno_notas.items():
    print(f"{aluno}: {nota}")
```

Saída:

```
Alice: 90
Bob: 80
Charlie: 95
```

Aqui, o `for` itera sobre os itens do dicionário, permitindo o acesso tanto ao nome do aluno quanto à sua nota.

Esses exemplos demonstram a versatilidade do `for` em Python, tornando-o uma ferramenta valiosa para uma variedade de situações, desde a iteração básica sobre listas até a manipulação de estruturas de dados mais complexas.

4.2.2 Estrutura de Repetição `while`

A estrutura `while` é um componente fundamental em Python, possibilitando a execução repetida de um bloco de código enquanto uma condição especificada permanece verdadeira. Isso é particularmente útil quando o número exato de iterações não é conhecido antecipadamente.

Sintaxe

```
while condicao:
    # Bloco de código a ser repetido
```

Neste contexto, o bloco de código é executado repetidamente enquanto a `condicao` é verdadeira. A condição é avaliada antes de cada iteração, e o loop continua até que a condição se torne falsa.

Exemplo Prático: Contagem até 5 usando `while`

```
contador = 1
while contador <= 5:
    print(contador)
    contador += 1
```

Saída:

```
1
2
3
4
5
```

Neste exemplo, o `while` é utilizado para contar até 5, incrementando o contador a cada iteração.

Casos de Utilização do `while`

1. **Entrada do Usuário:**

```
resposta = ""
while resposta.lower() != "sair":
    resposta = input("Digite 'sair' para encerrar: ")
```

   Aqui, o loop `while` permite que o programa continue solicitando entrada do usuário até que a resposta seja "sair".

2. **Execução Baseada em Condição Dinâmica:**

```
limite = 100
soma = 0
contador = 1
while soma < limite:
    soma += contador
    contador += 1
```

   Este exemplo ilustra como o `while` pode ser usado para realizar iterações com base em condições que podem ser determinadas dinamicamente.

3. **Validação de Entrada:**

```
nota = -1
while nota < 0 or nota > 100:
    nota = int(input("Digite uma nota entre 0 e 100: "))
```

   O loop `while` garante que o programa só prossiga quando uma entrada válida é fornecida pelo usuário.

A estrutura `while` é uma ferramenta poderosa quando o número de iterações não é fixo antecipadamente, proporcionando flexibilidade na execução de código em situações dinâmicas.

4.3 Exercícios

1. **Verificação de Idade**: Crie um programa que, ao receber a idade do usuário, imprima "Você é maior de idade" se a idade for 18 anos ou mais; caso contrário, imprima "Você é menor de idade".
2. **Calculadora de Bônus**: Faça um programa que, ao solicitar o salário de um funcionário, calcule e imprima um bônus de 10% se o salário for inferior a R$ 2000.
3. **Média de Notas**: Desenvolva um programa que, ao receber três notas do usuário, calcule a média. Em seguida, imprima se o aluno foi aprovado (média maior ou igual a 7) ou reprovado.
4. **Contagem Regressiva**: Escreva um programa que imprima uma contagem regressiva de 10 a 1 usando um loop while. Por exemplo, "Contagem regressiva: 10, 9, 8, …, 1".
5. **Verificação de Palíndromo**: Crie um programa que, ao solicitar ao usuário uma palavra, verifique se é um palíndromo (pode ser lida da mesma forma de trás para frente) e imprima o resultado.
6. **Tabuada Personalizada**: Peça ao usuário para fornecer um número. O programa deve imprimir a tabuada desse número de 1 a 10. Por exemplo, se o usuário inserir 5, o programa imprimirá "5 x 1 = 5", "5 x 2 = 10", …, "5 x 10 = 50".
7. **Validação de Senha**: Desenvolva um programa que, ao solicitar ao usuário criar uma senha, valide se ela tem pelo menos 8 caracteres e inclui pelo menos um número. Dê feedback ao usuário sobre a validação.

8. **Jogo da Adivinhação**: Crie um jogo em que o programa gere um número aleatório entre 1 e 100. O usuário deve tentar adivinhar o número, e o programa deve fornecer dicas sobre se o número é maior ou menor após cada tentativa.
9. **Fatorial**: Implemente um programa que, ao solicitar ao usuário inserir um número, calcule e imprima o fatorial desse número. O fatorial de um número é o produto de todos os números inteiros de 1 até o próprio número.
10. **Classificação de Triângulos**: Solicite ao usuário os comprimentos dos lados de um triângulo. O programa deve classificar o triângulo como equilátero (todos os lados iguais), isósceles (dois lados iguais) ou escaleno (nenhum lado igual).

CHAPTER

FIVE

CAPÍTULO 5: FUNÇÕES

As funções desempenham um papel fundamental na estruturação de código em Python. Elas permitem que você divida seu programa em partes gerenciáveis e reutilizáveis, facilitando a manutenção e compreensão do código. Neste capítulo, exploraremos os conceitos essenciais relacionados a funções em Python.

5.1 Definindo Funções

Uma função é um bloco de código reutilizável que executa uma tarefa específica. Ela é definida usando a palavra-chave `def`, seguida do nome da função e parênteses contendo os parâmetros. O bloco de código da função é indentado.

Exemplo:

```
def saudacao(nome_pessoa):
    """
    Esta função imprime uma saudação personalizada.

    :param nome_pessoa: Uma string representando o nome da pessoa a ser saudada.
    """
    print(f"Olá, {nome_pessoa}!")

# Chamando a função
saudacao("Alice")
```

```
Olá, Alice!
```

A função `saudacao` imprime uma saudação personalizada com base no nome fornecido como argumento. No exemplo, ao chamar a função com “Alice”, ela exibe “Olá, Alice!” na tela. Essa função realiza uma saudação personalizada.

Ao definir funções em Python, além de compreender sua estrutura básica, é fundamental adotar boas práticas para a nomenclatura de funções e parâmetros. Seguir diretrizes de nomenclatura aprimora a legibilidade do código e facilita a colaboração em projetos. Algumas regras essenciais incluem:

5.1.1 Nomes de Funções:

- **Convenção Snake Case:** Utilize letras minúsculas e underline para separar palavras. Exemplo: `calcular_media`, `processar_dados`.
- **Clareza e Descritividade:** Escolha nomes que claramente descrevam a função desempenhada, proporcionando entendimento imediato do propósito da função.
- **Evite Palavras Reservadas:** Não utilize nomes que são palavras reservadas em Python, como `print`, `if`, `else`, entre outras.
- **Convenção de Documentação:** Siga a PEP 257 para docstrings. Forneça uma breve descrição do propósito da função, especificando tipos de parâmetros e de retorno, quando aplicável.

5.1.2 Nomes de Parâmetros:

- **Convenção Snake Case:** Mantenha a consistência na nomenclatura, utilizando letras minúsculas e underline para separar palavras. Exemplo: `nome_completo`, `valor_total`.
- **Descrição Significativa:** Escolha nomes de parâmetros que indiquem claramente sua função na execução da função.

5.2 Argumentos de Função

Os argumentos são valores fornecidos a uma função quando ela é chamada. Uma função pode ter parâmetros, que são variáveis usadas para receber esses argumentos.

Exemplo:

```
def soma(a, b):
    """Esta função retorna a soma de dois números."""
    resultado = a + b
    return resultado

# Chamando a função com argumentos
resultado_soma = soma(3, 5)
print(f"A soma é: {resultado_soma}")
```

```
A soma é: 8
```

A função `soma` ilustra o conceito de parâmetros posicionais, onde os argumentos são correspondidos à ordem dos parâmetros na função. No exemplo, o argumento `3` corresponde a `a` e o argumento `5` corresponde a `b`.

Além dos parâmetros posicionais, as funções em Python podem ter parâmetros chave-valor, também conhecidos como argumentos nomeados. Isso permite que você especifique explicitamente para qual parâmetro está passando um determinado valor, tornando a chamada da função mais explícita. Exemplo:

```
def saudacao(nome, mensagem="Olá"):
    """Esta função exibe uma saudação com uma mensagem opcional."""
    print(f"{mensagem}, {nome}!")

# Chamando a função com argumentos nomeados
saudacao(nome="Alice", mensagem="Bom dia")
```

```
Bom dia, Alice!
```

Neste exemplo, `nome` é um parâmetro posicional, enquanto `mensagem` é um parâmetro chave-valor com um valor padrão de "Olá". Ao chamar a função, podemos especificar `mensagem` de forma explícita, como em `mensagem="Bom dia"`. Isso oferece flexibilidade e clareza na passagem de argumentos para funções.

5.3 Retorno de Valor

O retorno de valor permite que uma função envie um resultado de volta ao ponto de chamada. Isso é feito usando a palavra-chave `return`.

Exemplo:

```
def quadrado(x):
    """Esta função retorna o quadrado de um número."""
    return x ** 2

# Chamando a função e usando o valor retornado
valor_quadrado = quadrado(4)
print(f"O quadrado é: {valor_quadrado}")
```

```
O quadrado é: 16
```

O código define uma função chamada `quadrado` que calcula o quadrado de um número. Em seguida, a função é chamada com o argumento 4, e o resultado é armazenado na variável `valor_quadrado`, que é então impressa, mostrando o quadrado do número 4.

5.4 Funções Anônimas (Lambda)

As funções anônimas, ou lambdas, são funções pequenas e temporárias definidas sem um nome formal. Elas são criadas usando a palavra-chave `lambda`.

Exemplo:

```
dobro = lambda x: x * 2
print(f"O dobro de 3 é: {dobro(3)}")
```

```
O dobro de 3 é: 6
```

O código utilizou uma função lambda para calcular o dobro de um número. Na última linha, o código aplica essa função ao número 3 e imprime o resultado, demonstrando como calcular e exibir o dobro de um valor específico.

5.5 Funções Recursivas

Funções recursivas são aquelas que chamam a si mesmas durante a execução. Isso é útil para resolver problemas que podem ser quebrados em casos menores do mesmo problema.

Exemplo:

```
def fatorial(n):
    """Esta função retorna o fatorial de um número."""
    if n == 0 or n == 1:
        return 1
    else:
        return n * fatorial(n - 1)

# Chamando a função recursiva
resultado_fatorial = fatorial(5)
print(f"O fatorial é: {resultado_fatorial}")
```

```
O fatorial é: 120
```

A função `fatorial` é uma função recursiva que calcula o fatorial de um número. Se o número for 0 ou 1, ela retorna 1; caso contrário, ela multiplica o número pelo fatorial do número anterior, chamando a si mesma de forma recursiva. No exemplo, a função é chamada com o argumento 5, resultando no cálculo do fatorial, que é impresso como "O fatorial é: 120".

5.6 Funções *built-in*

As funções *built-in* do Python são funções que estão na própria linguagem e estão sempre disponíveis para uso, sem a necessidade de importar módulos específicos. Essas funções fornecem funcionalidades essenciais que são amplamente utilizadas. Algumas delas incluem:

- **`print()`:** Imprime mensagens ou valores na saída padrão.

```
print("Olá, mundo!")
```

```
Olá, mundo!
```

- **`len()`:** Retorna o número de elementos em uma sequência (como uma lista, tupla ou string).

```
tamanho = len("Python")
print(f"O tamanho da string é: {tamanho}")
```

```
O tamanho da string é: 6
```

- **`type()`:** Retorna o tipo de um objeto.

```
tipo = type(42)
print(f"O tipo do objeto é: {tipo}")
```

```
O tipo do objeto é: <class 'int'>
```

5.6.1 Funções para Conversão de Tipos

- **`int()`**, **`float()`**, **`str()`**, **`list()`**, **`tuple()`**, **`dict()`**: Convertem valores entre diferentes tipos.

```
numero_texto = "123"
numero_inteiro = int(numero_texto)
print(f"O tipo agora é: {type(numero_inteiro)}")
```

```
O tipo agora é: <class 'int'>
```

5.6.2 Funções para Manipulação de Sequências

- **`max()`**, **`min()`**: Encontram o valor máximo e mínimo em uma sequência.

```
numeros = [2, 8, 5, 10, 3]
maximo = max(numeros)
print(f"O valor máximo é: {maximo}")
```

```
O valor máximo é: 10
```

- **`sum()`**: Retorna a soma dos elementos em uma sequência numérica.

```
total = sum(numeros)
print(f"A soma dos elementos é: {total}")
```

```
A soma dos elementos é: 28
```

5.6.3 Funções para Interagir com o Usuário

- **`input()`**: Solicita entrada do usuário.

```
nome_usuario = input("Digite seu nome: ")
print(f"Olá, {nome_usuario}!")
```

```
Digite seu nome: Ana
Olá, Ana!
```

5.6.4 Funções Matemáticas

- **`abs()`**: Retorna o valor absoluto de um número.

```
numero_negativo = -5
absoluto = abs(numero_negativo)
print(f"O valor absoluto é: {absoluto}")
```

```
O valor absoluto é: 5
```

- **pow()**, **sqrt()**: Potenciação e raiz quadrada.

```
quadrado = pow(3, 2)
raiz_quadrada = sqrt(9)
```

```
O quadrado é: 9
```

Ao compreender e utilizar essas funções incorporadas, você poderá tornar seu código mais eficiente e conciso, aproveitando as capacidades que o Python oferece por padrão.

Observação: Para utilizar a função `sqrt()` (raiz quadrada) mencionada no exemplo, é necessário importar o módulo `math` em seu código Python. A função `sqrt()` faz parte do módulo `math` e não está disponível por padrão. Portanto, antes de usar a função `sqrt()`, inclua a seguinte linha de importação no início código:

```
from math import sqrt
```

5.7 Exercícios

1. **Máximo da Lista**: Crie uma lista de números inteiros. Utilize a função `max()` para encontrar o valor máximo e imprima o resultado.
2. **Comprimento da String**: Defina uma string, por exemplo, "Python". Utilize a função `len()` para calcular o número de caracteres na string e exiba o resultado.
3. **Conversão de Entrada**: Solicite ao usuário que insira um número utilizando `input()`. Converta o input para um número inteiro com `int()` e imprima o resultado.
4. **Soma dos Primeiros Números**: Crie uma lista com os primeiros cinco números naturais. Calcule a soma usando a função `sum()` e exiba o resultado.
5. **Mínimo da Lista**: Tenha uma lista de números. Encontre o valor mínimo utilizando `min()` e imprima o resultado.
6. **Conversão de String para Inteiro**: Dada a string "42", converta-a para um número inteiro utilizando `int()` e exiba o resultado.
7. **Conversão de Decimal**: Peça ao usuário que insira um número decimal utilizando `input()`. Converta o input para um número de ponto flutuante usando `float()` e imprima o resultado.
8. **Valor Absoluto**: Defina um número negativo. Utilize a função `abs()` para encontrar o valor absoluto e imprima-o.
9. **Raiz Quadrada com Math**: Importe o módulo `math`. Calcule a raiz quadrada de 16 usando `sqrt()` e imprima o resultado.
10. **Conversão de Lista para Caracteres**: Crie uma lista com três strings. Converta essa lista em uma lista de caracteres utilizando `list()` e exiba o resultado.

CHAPTER

SIX

CAPÍTULO 6: MANIPULAÇÃO DE STRINGS

6.1 Introdução a Strings

As strings em Python são sequências de caracteres, o que significa que são compostas por letras, números, símbolos ou espaços. Essas estruturas de dados permitem uma variedade de manipulações para atender às necessidades do desenvolvedor. Neste contexto, abordaremos operações fundamentais, métodos e técnicas essenciais de manipulação de strings.

6.1.1 Operações com Strings

Concatenação

A concatenação de strings é uma operação que combina duas ou mais strings em uma única string. Isso é feito utilizando o operador +.

Exemplo:

```
string1 = "Olá, "
string2 = "mundo!"
concatenacao = string1 + string2
print(concatenacao)
```

```
Olá, mundo!
```

Repetição

A repetição de strings envolve duplicar ou triplicar o conteúdo de uma string. Isso é alcançado pelo operador *.

Exemplo:

```
repeticao = "abc" * 3
print(repeticao)
```

```
abcabcabc
```

Indexação e Fatiamento (*Slicing*)

As strings são indexadas, o que significa que cada caractere tem uma posição única. A indexação permite acessar caracteres específicos. Além disso, o fatiamento (*slicing*) possibilita extrair partes específicas da string.

Exemplo:

```
fruta = "banana"
primeira_letra = fruta[0]
substr = fruta[1:4]
print(primeira_letra)
print(substr)
```

```
b
ana
```

6.1.2 Métodos de Strings

`upper()` e `lower()`

Os métodos `upper()` e `lower()` alteram o caso da string para maiúsculas e minúsculas, respectivamente.

Exemplo:

```
fruta = "maçã"
maiuscula = fruta.upper()
print(maiuscula)
```

```
MAÇÃ
```

`replace()`

O método `replace()` substitui parte da string por outra.

Exemplo:

```
mensagem = "Olá, mundo!"
nova_mensagem = mensagem.replace("mundo", "Python")
print(nova_mensagem)
```

```
Olá, Python!
```

`split()`

O método `split()` divide uma string em uma lista de substrings com base em um delimitador.

Exemplo:

```
frase = "Python é uma linguagem poderosa"
palavras = frase.split(" ")
print(palavras)
```

```
['Python', 'é', 'uma', 'linguagem', 'poderosa']
```

6.1.3 Explorando mais Conceitos

Formatação de Strings (f-strings)

As f-strings são uma forma eficiente e legível de formatar strings com valores de variáveis.

Exemplo:

```
nome = "Alice"
idade = 25
mensagem = f"Olá, meu nome é {nome} e tenho {idade} anos."
print(mensagem)
```

```
Olá, meu nome é Alice e tenho 25 anos.
```

Métodos `startswith()` e `endswith()`

Esses métodos verificam se uma string começa ou termina com uma determinada substring.

Exemplo:

```
frase = "Python é incrível!"
print(frase.startswith("Python"))
print(frase.endswith("incrível"))
```

```
True
False
```

6.2 Expressões Regulares em Python

As expressões regulares, conhecidas como regex, são uma poderosa ferramenta para manipulação e busca em strings no Python, proporcionando flexibilidade e eficiência. O módulo `re` oferece suporte para trabalhar com expressões regulares.

6.2.1 Sintaxe Básica e Principais Funções

Para começar, importamos o módulo `re`. A seguir, vamos analisar o exemplo para entender como as funções `re.match` e `re.search` funcionam:

Exemplo

```
import re

# Exemplo 1: Utilizando re.match para encontrar um padrão no início da string
padrao_match = r'\d+'
texto_match = "123 Python"
resultado_match = re.match(padrao_match, texto_match)
print(f"Exemplo 1 - Buscando padrão {padrao_match} no início do texto '{texto_match}':
↪")
print("Resultado:", resultado_match.group() if resultado_match else "Sem␣
↪correspondência")
print()

# Exemplo 2: Utilizando re.search para encontrar um padrão em qualquer lugar da string
padrao_search = r'\w+'
texto_search = "Python é uma linguagem poderosa"
string_busca = "linguagem"
resultado_search = re.search(padrao_search, texto_search)
print(f"Exemplo 2 - Buscando padrão {padrao_search} em qualquer lugar do texto '
↪{texto_search}':")
print(f"Buscando a string '{string_busca}':")
print("Resultado:", resultado_search.group() if resultado_search else "Sem␣
↪correspondência")
```

```
Exemplo 1 - Buscando padrão \d+ no início do texto '123 Python':
Resultado: 123

Exemplo 2 - Buscando padrão \w+ em qualquer lugar do texto 'Python é uma linguagem␣
↪poderosa':
Buscando a string 'linguagem':
Resultado: linguagem
```

Explicação:

- **Exemplo 1 - `re.match` com padrão `\d+`:**
  - O padrão `\d+` busca por um ou mais dígitos no início da string.
  - A string de texto é "123 Python".
  - `re.match` verifica se o padrão ocorre no início da string. Neste caso, o padrão é encontrado no início com os dígitos "123".
  - A saída será "Resultado: 123", indicando que houve correspondência no início da string.
- **Exemplo 2 - `re.search` com padrão `\w+`:**
  - O padrão `\w+` busca por uma ou mais letras, dígitos ou underscores em qualquer lugar da string.
  - A string de texto é "Python é uma linguagem poderosa".
  - `re.search` procura por uma correspondência em qualquer lugar da string. Neste caso, o padrão é encontrado no início com a palavra "Python".

  - A saída será “Resultado: linguagem”, indicando que houve correspondência em qualquer lugar da string com a palavra “linguagem”.

Esses exemplos ilustram o uso das funções `re.match` e `re.search` para encontrar padrões específicos em strings.

`re.findall(pattern, string)` e `re.finditer(pattern, string)`:

- `re.findall`: Encontra todas as correspondências e retorna como uma lista.
- `re.finditer`: Retorna um iterador que produz objetos de correspondência.

Exemplo:

```
import re

padrao = r'\w+'
texto = "Python é uma linguagem poderosa"

todas_correspondencias = re.findall(padrao, texto)
print("Todas as correspondências:", todas_correspondencias)

iterador_correspondencias = re.finditer(padrao, texto)
print("Iterador de correspondências:", [match.group() for match in iterador_
↪correspondencias])
```

```
Todas as correspondências: ['Python', 'é', 'uma', 'linguagem', 'poderosa']
Iterador de correspondências: ['Python', 'é', 'uma', 'linguagem', 'poderosa']
```

`re.sub(pattern, replacement, string)`:

Substitui todas as ocorrências do padrão por uma string de substituição.

Exemplo:

```
import re

padrao = r'\d+'
texto = "Python 2 e Python 3"
substituido = re.sub(padrao, "X", texto)
print("Texto original:", texto)
print("Texto substituído:", substituido)
```

```
Texto original: Python 2 e Python 3
Texto substituído: Python X e Python X
```

As expressões regulares podem ser aplicadas em diversas tarefas de manipulação de strings, incluindo busca, extração e transformação de padrões específicos. Para uma compreensão mais profunda, consulte a documentação oficial sobre expressões regulares em Python: Expressões Regulares (re).

6.2.2 Exercícios

1. Concatenação e Repetição: Escreva um programa que solicite ao usuário dois nomes de obras filosóficas de Friedrich Nietzsche e os concatene em uma única string. Em seguida, repita a string resultante três vezes e exiba o resultado.

   Testes:

```
# Teste 1
entrada: "Assim Falou Zaratustra", "Além do Bem e do Mal"
saída: "Assim Falou ZaratustraAlém do Bem e do MalAssim Falou ZaratustraAlém do␣
↪Bem e do MalAssim Falou ZaratustraAlém do Bem e do Mal"

# Teste 2
entrada: "Ecce Homo", "A Gaia Ciência"
saída: "Ecce HomoA Gaia CiênciaEcce HomoA Gaia CiênciaEcce HomoA Gaia Ciência"
```

2. Indexação e Fatiamento: Crie uma função que receba o título de uma obra filosófica de Nietzsche como entrada e retorne a primeira e última letra do título, separadas por um hífen.

   Testes:

```
# Teste 1
entrada: "Assim Falou Zaratustra"
saída: "A-a"

# Teste 2
entrada: "Para Além do Bem e do Mal"
saída: "P-l"
```

3. Métodos `upper()` e `lower()`: Peça ao usuário para fornecer um trecho de uma obra de Friedrich Nietzsche. Converta todas as letras para maiúsculas e, em seguida, para minúsculas. Exiba ambas as versões.

   Testes:

```
# Teste 1
entrada: "O eterno retorno é a chave para a compreensão do Übermensch."
saída: "O ETERNO RETORNO É A CHAVE PARA A COMPREENSÃO DO ÜBERMENSCH.", "o eterno␣
↪retorno é a chave para a compreensão do übermensch."

# Teste 2
entrada: "A vontade de poder é o motor da existência humana."
saída: "A VONTADE DE PODER É O MOTOR DA EXISTÊNCIA HUMANA.", "a vontade de poder␣
↪é o motor da existência humana."
```

4. Método `replace()`: Crie um programa que substitua todas as ocorrências da palavra "nada" por "eterno retorno" em uma passagem de uma obra de Nietzsche fornecida pelo usuário.

   Testes:

```
# Teste 1
entrada: "O homem não é nada além de seu próprio eterno retorno."
saída: "O homem não é eterno retorno além de seu próprio eterno retorno."

# Teste 2
entrada: "Aquele que olha para o abismo percebe que o abismo olha para ele e␣
↪retorna eternamente."
```

(continues on next page)

(continued from previous page)

```
saída: "Aquele que olha para o abismo percebe que o abismo olha para ele e␣
↪retorna eternamente."
```

5. Método `split()`: Escreva um programa que solicite ao usuário uma citação de uma obra de Nietzsche e divida a citação em palavras. Em seguida, exiba a contagem de palavras e as próprias palavras em uma lista.

   Testes:

```
# Teste 1
entrada: "Aquele que tem um porquê para viver pode suportar quase qualquer como."
saída: "Número de palavras: 10", "['Aquele', 'que', 'tem', 'um', 'porquê', 'para',
↪ 'viver', 'pode', 'suportar', 'quase', 'qualquer', 'como.']"

# Teste 2
entrada: "O indivíduo torna-se quem é ao enfrentar seus próprios demônios."
saída: "Número de palavras: 9", "['O', 'indivíduo', 'torna-se', 'quem', 'é', 'ao',
↪ 'enfrentar', 'seus', 'próprios', 'demônios.']"
```

6. Formatação de Strings (f-strings): Crie uma função que receba o título de uma obra, o ano de publicação e o tema de uma obra de Nietzsche como parâmetros e retorne uma mensagem formatada usando f-strings.

   Testes:

```
# Teste 1
entrada: "Assim Falou Zaratustra", 1883, "Filosofia Existencial"
saída: "A obra 'Assim Falou Zaratustra', publicada em 1883, aborda o tema da␣
↪Filosofia Existencial."

# Teste 2
entrada: "Ecce Homo", 1908, "Vontade de Poder"
saída: "A obra 'Ecce Homo', publicada em 1908, explora o tema da Vontade de Poder.
↪"
```

7. Métodos `startswith()` e `endswith()`: Escreva uma função que receba uma lista de títulos de obras de Nietzsche e uma letra como parâmetros. A função deve retornar uma lista com os títulos que começam com a letra fornecida.

   Testes:

```
# Teste 1
entrada: ["Assim Falou Zaratustra", "Além do Bem e do Mal", "Ecce Homo"], "A"
saída: "['Assim Falou Zaratustra', 'Além do Bem e do Mal']"

# Teste 2
entrada: ["Para Além do Bem e do Mal", "Ecce Homo", "A Gaia Ciência"], "E"
saída: "['Ecce Homo', 'A Gaia Ciência']"
```

8. Expressões Regulares - `re.match`: Crie uma função que utilize `re.match` para verificar se um título de obra de Nietzsche atende aos seguintes critérios: deve começar com uma letra maiúscula, conter pelo menos um número e ter no mínimo 10 caracteres.

   Testes:

```
# Teste 1
entrada: "Assim Falou Zaratustra"
```

(continues on next page)

(continued from previous page)

```
saída: True

# Teste 2
entrada: "ecce homo"
saída: False
```

9. Expressões Regulares - `re.search`: Implemente uma função que utilize `re.search` para encontrar todos os anos em que Nietzsche escreveu obras e retorne uma lista com esses anos.

   Testes:

```
# Teste 1
entrada: "Nietzsche publicou 'Assim Falou Zaratustra' em 1883 e 'Além do Bem e do␣
↪Mal' em 1886."
saída: "['1883', '1886']"

# Teste 2
entrada: "As obras de Nietzsche influenciaram o século XIX e o início do século␣
↪XX."
saída: "[]"
```

10. Expressões Regulares - `re.sub`: Escreva uma função que use `re.sub` para substituir todas as ocorrências de palavras que terminam com "mente" por "eternamente" em uma passagem filosófica de Nietzsche.

Testes:

```
# Teste 1
entrada: "A sabedoria é adquirida lentamente e aplicada conscientemente."
saída: "A sabedoria é adquirida eternamente e aplicada conscientemente."

# Teste 2
entrada: "A verdade é percebida somente individualmente e compreendida coletivamente."
saída: "A verdade é percebida somente individualmente e compreendida coletivamente."
```

CHAPTER

SEVEN

CAPÍTULO 7: NUMPY E MATPLOTLIB

7.1 Biblioteca NumPy

7.1.1 O que é o NumPy?

O NumPy é uma biblioteca de código aberto para Python, especializada em manipulação eficiente de arrays multidimensionais. Essa biblioteca é fundamental para a programação científica e de engenharia em Python devido à sua capacidade de realizar operações sofisticadas em grandes conjuntos de dados de forma rápida e eficiente.

7.1.2 Importância do NumPy

O NumPy desempenha um papel crucial em várias disciplinas, incluindo ciência da computação, matemática, estatística, engenharia, processamento de sinais e visão computacional. Suas principais características incluem:

1. **Arrays Multidimensionais:** Oferece a capacidade de criar e manipular arrays N-dimensionais, fornecendo uma estrutura eficiente para armazenar e operar em dados.
2. **Operações Aritméticas e Lógicas:** Permite realizar operações aritméticas e lógicas em arrays, facilitando cálculos complexos.
3. **Estatísticas e Álgebra Linear:** Oferece funções integradas para realizar operações estatísticas e de álgebra linear em arrays, como cálculos de média, desvio padrão, multiplicação de matrizes, entre outros.

7.1.3 Exemplos de Uso do NumPy

Vamos explorar alguns exemplos práticos para entender como o NumPy pode ser utilizado:

1. Criação de Arrays Multidimensionais

```
import numpy as np

# Cria um array bidimensional
array = np.array([[1, 2, 3], [4, 5, 6]])

# Imprime o array
print(array)
```

Saída:

```
[[1 2 3]
 [4 5 6]]
```

2. Operações Aritméticas em Arrays

```
import numpy as np

# Cria dois arrays
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Soma os dois arrays
array_soma = array1 + array2

# Imprime o array soma
print(array_soma)
```

Saída:

```
[5 7 9]
```

3. Operações Lógicas em Arrays

```
import numpy as np

# Cria um array
array = np.array([1, 2, 3, 4, 5])

# Verifica se todos os elementos do array são maiores que 2
todos_maiores_que_2 = array > 2

# Imprime o resultado da verificação
print(todos_maiores_que_2)
```

Saída:

```
[True True True True True]
```

4. Estatísticas em Arrays

```
import numpy as np

# Cria um array
array = np.array([1, 2, 3, 4, 5])

# Calcula a média do array
media = np.mean(array)

# Imprime a média
print(media)
```

Saída:

```
3.0
```

5. Álgebra Linear em Arrays

```
import numpy as np

# Cria duas matrizes
matriz1 = np.array([[1, 2], [3, 4]])
matriz2 = np.array([[5, 6], [7, 8]])

# Multiplica as duas matrizes
matriz_multiplicacao = np.matmul(matriz1, matriz2)

# Imprime a matriz multiplicação
print(matriz_multiplicacao)
```

Saída:

```
[[23 26]
 [47 52]]
```

7.2 Visualização de Dados com o Matplotlib

7.2.1 O que é o Matplotlib?

O Matplotlib é uma biblioteca de visualização de dados em Python que permite criar uma ampla variedade de gráficos, plots, mapas de calor e visualizações personalizadas. Essa biblioteca oferece flexibilidade e controle total sobre o design das visualizações, tornando-se uma ferramenta poderosa para apresentar dados de maneira eficaz.

7.2.2 Importância do Matplotlib

A visualização de dados desempenha um papel fundamental na análise e interpretação de informações. O Matplotlib simplifica a criação de gráficos informativos e esteticamente agradáveis, sendo amplamente utilizada em diversas áreas, como:

- Ciência de dados
- Pesquisa acadêmica
- Engenharia
- Finanças
- Biologia
- Geociências

7.2.3 Principais Características do Matplotlib

1. **Simplicidade de Uso:** Permite criar gráficos com poucas linhas de código, sendo acessível para iniciantes e flexível para usuários avançados.
2. **Suporte a Diversos Tipos de Gráficos:** Oferece suporte a uma variedade de gráficos, incluindo linha, barra, dispersão, histograma, pizza, entre outros.
3. **Personalização:** Permite personalizar cada aspecto dos gráficos, desde cores e estilos de linha até títulos e rótulos.
4. **Integração com NumPy:** Funciona perfeitamente com arrays NumPy, facilitando a visualização de dados armazenados em arrays multidimensionais.

7.2.4 Exemplos de Uso do Matplotlib

Vamos explorar alguns exemplos práticos de visualizações criadas com o Matplotlib:

1. Gráfico de Linha Simples

```
import matplotlib.pyplot as plt

# Dados
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Criando o gráfico de linha
plt.plot(x, y)

# Adicionando título e rótulos aos eixos
plt.title("Gráfico de Linha Simples")
plt.xlabel("Eixo X")
plt.ylabel("Eixo Y")

# Exibindo o gráfico
plt.show()
```

2. Gráfico de Dispersão

```
import matplotlib.pyplot as plt

# Dados
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

# Criando o gráfico de dispersão
plt.scatter(x, y, color='red', marker='o')

# Adicionando título e rótulos aos eixos
plt.title("Gráfico de Dispersão")
plt.xlabel("Eixo X")
plt.ylabel("Eixo Y")
```

(continues on next page)

(continued from previous page)

```
# Exibindo o gráfico
plt.show()
```

3. Histograma

```
import matplotlib.pyplot as plt
import numpy as np

# Dados aleatórios
dados = np.random.randn(1000)

# Criando o histograma
plt.hist(dados, bins=30, color='green', alpha=0.7)

# Adicionando título e rótulos aos eixos
plt.title("Histograma")
plt.xlabel("Valores")
plt.ylabel("Frequência")

# Exibindo o gráfico
plt.show()
```

7.2.5 Conclusão

O Matplotlib é uma ferramenta poderosa e versátil para visualização de dados em Python. Com uma variedade de gráficos disponíveis e recursos de personalização, ela se destaca na criação de representações visuais impactantes. Ao combinar o Matplotlib com outras bibliotecas, como NumPy e Pandas, os desenvolvedores podem explorar e comunicar insights valiosos a partir de conjuntos de dados complexos. Entender e dominar o Matplotlib é uma habilidade valiosa para profissionais que buscam criar visualizações envolventes e informativas.

7.3 Exercícios