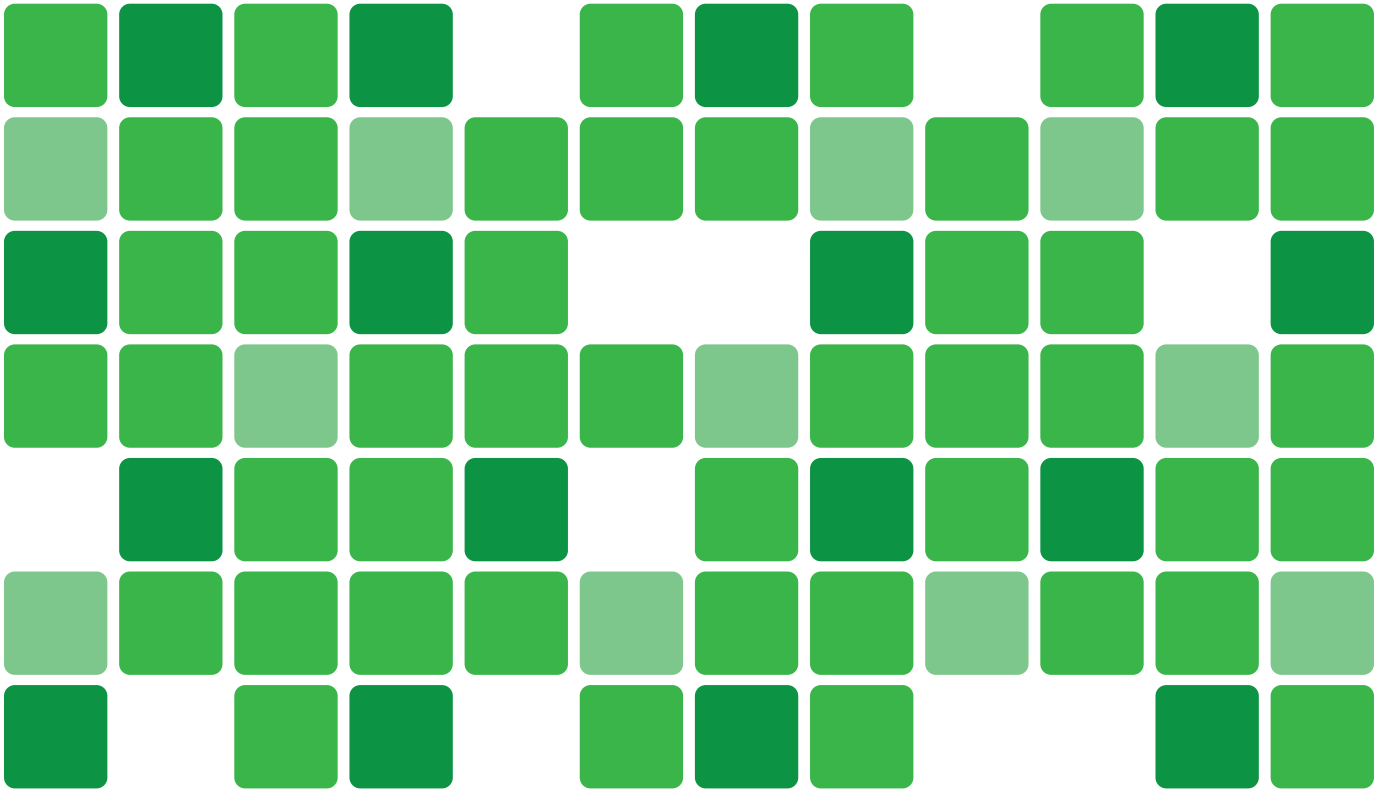
Programação para a Web

Curso Técnico em Informática

Prof. Dr. Ricardo Maroquio



Instituto Federal do Espírito Santo
Campus Cachoeiro

Disponível para download em:
http://maroquio.com

Fevereiro de 2023

Sumário

Lista de Exemplos de Código

Parte I: Python Essencial

1. Fundamentos da Linguagem

O Python é uma linguagem de programação poderosa e versátil amplamente utilizada em aplicações de computação científica, análise de dados, desenvolvimento de aplicações e muito mais. Neste primeiro capítulo, revisaremos os conceitos básicos da programação em Python e veremos como aplicá-los para resolver problemas comuns do mundo da programação.

Vale ressaltar que este capítulo tem como proposta a realização de uma rápida revisão da linguagem Python, ou seja, pressupõe-se que você já tenha tido alguma experiência com programação em Python. Ao longo do capítulo, você aprenderá sobre tipos de dados, variáveis, estruturas de controle de fluxo, funções e muito mais. Também veremos muitos exemplos práticos que irão ajudá-lo a aplicar os conceitos abordados.

Ao concluir este capítulo, você terá revisto todos os recursos necessários para escrever programas simples em Python e estará preparado para se aprofundar em recursos adicionais de programação em Python com mais confiança.

1.1 Variáveis

Variáveis são consideradas componentes fundamentais da programação. Elas permitem armazenar valores que podem ser usados posteriormente ao longo do programa. Em Python, as variáveis são criadas usando o sinal de igual (=). O exemplo 1.1 mostra a criação de duas variáveis simples.

```
1 nome = "João"
2 idade = 30
```

Exemplo de Código 1.1: Atribuição de valores a variáveis.

Neste exemplo, criamos duas variáveis: `nome` e `idade`. O nome da variável fica à esquerda do sinal de igual e o valor da variável fica à direita. O tipo da variável é determinado automaticamente com base no valor atribuído. Uma variável pode ter seu valor modificado a qualquer momento. O exemplo 1.2

```
nome = 'João'
idade = 30
print("Meu nome é", nome, "e tenho", idade, "anos.")

nome = "Maria"
idade = 35
print('Meu nome é', nome, "e tenho", idade, "anos.")
```

Exemplo de Código 1.2: Reatribuição de valores a variáveis.

Neste exemplo, alteramos o valor da variável `nome` de “João” para “Maria” e o valor da variável `idade` de 30 para 35. Quando o programa é executado, o texto “Meu nome é Maria e tenho 35 anos.” é impresso na tela.

Lembre-se de que as variáveis em Python não precisam ser declaradas com antecedência. Basta atribuir um valor a uma variável para criá-la automaticamente. Além disso, é importante escolher nomes de variáveis descritivos e significativos para facilitar a leitura e manutenção do código. As regras para nomeação de variáveis em Python são as seguintes:

1. Nomes de variáveis devem começar com uma letra ou um sublinhado (_);
2. Após o primeiro caractere, podem conter letras, números e sublinhados;
3. Nomes de variáveis não podem ser uma palavra reservada do Python, como `if`, `else`, `for` etc.;
4. Nomes de variáveis devem ser descritivos e significativos;
5. Nomes de variáveis não podem conter espaços. Em vez disso, usa-se o sublinhado para separar palavras;
6. Nomes de variáveis devem ser escritos em minúsculas, exceto em casos de convenções de nomenclatura como `camelCase`, `PascalCase` ou `snake_case`.

Agora que você já sabe como declarar variáveis em Python, vamos quais são os possíveis tipos de valores que podemos atribuir a uma variável.

1.2 Tipos de Dados

Em programação, é importante compreender os diferentes tipos de dados que você pode trabalhar. Em Python, existem vários tipos de dados básicos que você precisa conhecer, incluindo números, *strings*, listas, dicionários, conjuntos, entre outros. Vamos explorar cada um desses tipos em detalhes.

1.2.1 Números

Existem dois tipos numéricos em Python: inteiros (*int*) e de ponto flutuante (*float*). No exemplo 1.3, veja que a variável `x` recebe um valor inteiro e que a variável `y` recebe um valor real.

```
1 x = 10  # int
2 y = 3.14  # float
```

Exemplo de Código 1.3: Tipos numéricos inteiro e real (ponto flutuante).

1.2.2 Textos

Textos são uma sequência de caracteres. Em Python, são representados pelo tipo *string* e seus valores podem ter diferentes formas. O exemplo 1.4 mostra como criar *strings* usando aspas simples, duplas ou 3 aspas duplas ou simples para *strings* de múltiplas linhas.

```
1 frase1 = "Olá, Python!"  # string delimitado por aspas duplas
2 frase2 = 'Olá, Python!'  # string delimitado por aspas simples
3 frase3 = """Este é um texto com
4     múltiplas linhas. Você deve
5     usar 3 aspas para delimitar
6     o início e o fim da string""" # string delimitado pro 3 aspas duplas
```

Exemplo de Código 1.4: Criação de variáveis com valores do tipo *string*.

1.2.3 Listas

As listas são uma estrutura de dados em Python que permitem armazenar uma coleção de itens tipos iguais ou diferentes. As listas são delimitadas por colchetes [] e os itens são separados por vírgulas. O exemplo 1.5 mostra a criação de 3 listas diferentes.

Você pode acessar, alterar ou remover itens em uma lista usando seus índices (posições) e também pode realizar operações comuns em listas, como concatenação, repetição e pesquisa. Além disso, as listas são objetos mutáveis, o que significa que você pode modificar seus itens depois de criá-los. Veremos listas novamente mais adiante de forma mais aprofundada.

```
# criando uma lista de números inteiros
numeros = [1, 2, 3, 4, 5]

# criando uma lista de strings
frutas = ['maçã', 'banana', 'laranja']

# criando uma lista mista de diferentes tipos de dados
mista = [1, 'dois', 3.0, ['a', 'b', 'c']]
```

Exemplo de Código 1.5: Criação de listas de diferentes tipos.

1.2.4 Dicionários

Em Python, um dicionário é uma estrutura de dados que permite armazenar uma coleção de pares chave-valor. Ao contrário de uma lista, que armazena itens em uma ordem específica baseada em índices, um dicionário armazena itens com base nas suas chaves. As chaves são usadas para acessar seus valores correspondentes.

Os dicionários são delimitados por chaves e cada par chave-valor é separado por vírgulas. As chaves podem ser de diferentes tipos, como números, *strings* ou tuplas, mas precisam ser imutáveis. Os valores podem ser de qualquer tipo. O exemplo 1.6 mostra a criação e algumas operações básicas sobre um dicionário.

```
# criando um dicionário com pares chave-valor
pessoa = {'nome': 'João', 'idade': 30, 'cidade': 'São Paulo'}

# acessando um valor pelo seu nome de chave
print(pessoa['nome']) # Saída: "João"

# alterando o valor de uma chave
pessoa['idade'] = 31
print(pessoa['idade']) # Saída: 31

# adicionando um novo par chave-valor
pessoa['pais'] = 'Brasil'
print(pessoa)
# Saída: {'nome': 'João', 'idade': 31, 'cidade': 'São Paulo', 'pais': 'Brasil'}

# removendo um par chave-valor
del pessoa['cidade']
print(pessoa) # Saída: {'nome': 'João', 'idade': 31, 'pais': 'Brasil'}
```

Exemplo de Código 1.6: Criando dicionários de diferentes tipos.

1.2.5 Conjuntos

Em Python, conjuntos são coleções não ordenadas e não indexadas de elementos únicos. Eles são definidos usando chaves ou a função set(). O exemplo 1.7 mostra as duas formas de criação de conjuntos.

```
conjunto = {1, 2, 3, 4}
print(conjunto) # output: {1, 2, 3, 4}

conjunto = set([1, 2, 3, 4])
print(conjunto) # output: {1, 2, 3, 4}
```

Exemplo de Código 1.7: Duas formas de se criar conjuntos.

Os elementos em um conjunto não podem ser repetidos e não possuem uma ordem específica. Isso significa que você não pode acessar um elemento específico de um conjunto usando um índice, mas você pode verificar se um elemento está presente no conjunto usando o operador `in`. Você pode realizar operações comuns com conjuntos, como união, interseção e diferença. O exemplo 1.8 mostra a realização de algumas dessas operações.

```
conjunto1 = {1, 2, 3, 4}
conjunto2 = {3, 4, 5, 6}

# União
print(conjunto1 | conjunto2) # output: {1, 2, 3, 4, 5, 6}

# Interseção
print(conjunto1 & conjunto2) # output: {3, 4}

# Diferença
print(conjunto1 - conjunto2) # output: {1, 2}
```

Exemplo de Código 1.8: Operações sobre conjuntos.

1.2.6 Tuplas

As tuplas são uma das estruturas de dados básicas em Python e são semelhantes a listas. A principal diferença é que as tuplas são imutáveis, ou seja, uma vez criadas, seus elementos não podem ser alterados. As tuplas constumam ser bastante usadas em aplicações que acessam banco de dados, pois é uma boa forma de se representar um registro ou um objeto de um banco de dados. A sintaxe para criar uma tupla é colocar vários elementos separados por vírgulas, envolvidos por parênteses. O exemplo 1.9 mostra a criação de algumas tuplas.

```
# Exemplo 1: Criando uma tupla vazia
tupla_vazia = ()
print(tupla_vazia)  # Saída: ()

# Exemplo 2: Criando uma tupla com elementos
tupla = 1, 2, 3, 4
print(tupla)  # Saída: (1, 2, 3, 4)

# Exemplo 3: Criando uma tupla com elementos e parênteses
tupla = (1, 2, 3, 4)
print(tupla)  # Saída: (1, 2, 3, 4)

# Exemplo 4: Acessando elementos em uma tupla
tupla = (1, 2, 3, 4)
print(tupla[0])  # Saída: 1
print(tupla[-1])  # Saída: 4

# Exemplo 5: Tuplas com elementos de diferentes tipos
tupla = (1, 2, "três", 4.0)
print(tupla)  # Saída: (1, 2, "três", 4.0)
```

Exemplo de Código 1.9: Operações complementares sobre conjuntos.

Como as tuplas são imutáveis, não é possível adicionar, remover ou alterar seus elementos após sua criação. No entanto, é possível concatenar tuplas, criar novas tuplas a partir de tuplas existentes, combinar coleções para se criar tuplas, entre outras operações. A listagem a seguir apresenta mais algumas informações sobre outras possíveis operações que podem ser realizadas nas tuplas:

- **Índices**: assim como nas listas, é possível acessar elementos específicos de uma tupla usando seus índices. Os índices começam a partir de 0 e você também pode usar índices negativos, o que significa contar a partir do final da tupla.
- **Fatiamento**: é possível fazer o "fatiamento" (*slicing*) de tuplas da mesma forma que se faz com listas. Isso significa que você pode selecionar uma sub-tupla de uma tupla existente, especificando o índice inicial e o final da seleção.
- **Desempacotamento**: é possível "desempacotar" os elementos de uma tupla e atribuí-los a várias variáveis. Isso é útil quando você precisa extrair vários valores de uma tupla e tratá-los separadamente.
- **Funções nativas**: existem várias funções nativas em Python que são úteis para trabalhar com tuplas, como `len()`, `min()`, `max()`, `sum()`, entre outras.

O código 1.10 mostra mais alguns exemplos do uso de tuplas com os operadores citados.

```
# Exemplo 6: Fatiamento de tuplas
tupla = (1, 2, 3, 4, 5)
print(tupla[1:3])  # Saída: (2, 3)
print(tupla[:-2])  # Saída: (1, 2, 3)

# Exemplo 7: Desempacotamento de tuplas
tupla = (1, 2, 3)
a, b, c = tupla
print(a)  # Saída: 1
print(b)  # Saída: 2
print(c)  # Saída: 3

# Exemplo 8: Concatenação de tuplas
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla_concatenada = tupla1 + tupla2
print(tupla_concatenada)  # Saída: (1, 2, 3, 4, 5, 6)

# Exemplo 9: Funções nativas
tupla = (1, 2, 3, 4, 5)
print(len(tupla))  # Saída: 5
print(min(tupla))  # Saída: 1
print(max(tupla))  # Saída: 5
print(sum(tupla))  # Saída: 15
```

Exemplo de Código 1.10: Operações sobre tuplas.

1.3 Operadores Matemáticos

Em Python, existem vários operadores matemáticos que permitem realizar operações matemáticas básicas com números, como adição, subtração, multiplicação, divisão, exponenciação etc. As subseções a seguir apresentam os operadores matemáticos nativos mais comuns da linguagem Python, juntamente com exemplos uso.

1.3.1 Adição

O operador de adição, representado pelo caractere +, adiciona dois números ou concatena duas *strings*. Veja o exemplo 1.11 a seguir.

1.3.2 Subtração

O operador de subtração, representado pelo caractere -, subtrai dois números. Veja o exemplo 1.12 a seguir.

```
a = 5
b = 3
c = a + b
print(c) # Saída: 8

s1 = "Olá"
s2 = " mundo!"
s3 = s1 + s2
print(s3) # Saída: "Olá mundo!"
```

Exemplo de Código 1.11: Usos do operador de adição.

```
a = 5
b = 3
c = a - b
print(c) # Saída: 2
```

Exemplo de Código 1.12: Uso do operador de subtração.

1.3.3 Multiplicação

O operador de multiplicação, representado pelo caractere $*$, multiplica dois números ou repetidamente concatena uma *string*. Veja o exemplo 1.13 a seguir.

```
a = 5
b = 3
c = a * b
print(c) # Saída: 15

s = "Oi"
r = s * 3
print(r) # Saída: "OiOiOi"
```

Exemplo de Código 1.13: Usos do operador de multiplicação.

1.3.4 Divisão

O operador de divisão, representado pelo caractere /, divide um número por outro. Veja o exemplo 1.14 a seguir.

1.3.5 Divisão Inteira

O operador de divisão inteira, representado pelos caracteres //, divide um número por outro e retorna somente a parte inteira do resultado. Veja o exemplo 1.15 a seguir.

```
a = 6
b = 3
c = a / b
print(c) # Saída: 2.0
```

Exemplo de Código 1.14: Uso do operador de divisão.

```
a = 7
b = 3
c = a // b
print(c) # Saída: 2
```

Exemplo de Código 1.15: Uso do operador de divisão inteira.

1.3.6 Resto da Divisão Inteira

O operador de resto da divisão inteira, representado pelo caractere %, retorna o resto da divisão inteira de um número por outro. Veja o exemplo 1.16 a seguir.

```
a = 7
b = 3
c = a % b
print(c) # Saída: 1
```

Exemplo de Código 1.16: Uso do operador de resto da divisão inteira.

1.3.7 Exponenciação

O operador de exponenciação, representado pelos caracteres **, eleva um número a uma potência. Veja o exemplo 1.17 a seguir.

```
a = 2
b = 3
c = a ** b
print(c) # Saída: 8
```

Exemplo de Código 1.17: Uso do operador de exponenciação.

1.3.8 Operações Trigonométricos

As funções trigonométricas são funções matemáticas que estudam a relação entre os ângulos de um triângulo retângulo e suas medidas. Em Python, você pode usar as funções trigonométricas da biblioteca nativa `math`. O código 1.18 mostra exemplos de uso das funções trigonométricas mais comuns.

```python
import math

# Exemplo 1: seno
angulo = 30
seno = math.sin(math.radians(angulo))
print("Seno de", angulo, "graus:", seno)

# Exemplo 2: cosseno
angulo = 60
cosseno = math.cos(math.radians(angulo))
print("Cosseno de", angulo, "graus:", cosseno)

# Exemplo 3: tangente
angulo = 45
tangente = math.tan(math.radians(angulo))
print("Tangente de", angulo, "graus:", tangente)

# Exemplo 4: arco seno
seno = 0.5
angulo = math.degrees(math.asin(seno))
print("Arco seno de", seno, ":", angulo, "graus")

# Exemplo 5: arco cosseno
cosseno = 0.5
angulo = math.degrees(math.acos(cosseno))
print("Arco cosseno de", cosseno, ":", angulo, "graus")

# Exemplo 6: arco tangente
tangente = 1.0
angulo = math.degrees(math.atan(tangente))
print("Arco tangente de", tangente, ":", angulo, "graus")
```

Exemplo de Código 1.18: Uso das funções trigonométricas do módulo `math`.

Lembre-se de que, em todos os exemplos acima, a entrada para as funções trigonométricas deve ser em radianos. Portanto, é necessário converter o ângulo em graus para radianos usando a função `math.radians()` antes de passar o ângulo como argumento de uma das funções trigonométricas mostradas.

1.3.9 Conversão de Tipos Primitivos

A conversão de tipos primitivos em Python é a operação de mudar o tipo de uma variável de sua forma original para outro tipo. Veja o exemplo 1.19 a seguir, que mostra alguns exemplos de conversão entre tipos primitivos.

```python
# Exemplo 1: inteiro para string
numero = 42
string = str(numero)
print("Número:", numero, "tipo:", type(numero))
print("String:", string, "tipo:", type(string))

# Exemplo 2: string para inteiro
string = "42"
numero = int(string)
print("String:", string, "tipo:", type(string))
print("Número:", numero, "tipo:", type(numero))

# Exemplo 3: float para string
ponto_flutuante = 3.14
string = str(ponto_flutuante)
print("Ponto flutuante:", ponto_flutuante, "tipo:", type(ponto_flutuante))
print("String:", string, "tipo:", type(string))

# Exemplo 4: string para float
string = "3.14"
ponto_flutuante = float(string)
print("String:", string, "tipo:", type(string))
print("Ponto flutuante:", ponto_flutuante, "tipo:", type(ponto_flutuante))

# Exemplo 5: inteiro para float
numero = 42
ponto_flutuante = float(numero)
print("Número:", numero, "tipo:", type(numero))
print("Ponto flutuante:", ponto_flutuante, "tipo:", type(ponto_flutuante))

# Exemplo 6: float para inteiro
ponto_flutuante = 3.14
numero = int(ponto_flutuante)
print("Ponto flutuante:", ponto_flutuante, "tipo:", type(ponto_flutuante))
print("Número:", numero, "tipo:", type(numero))
```

Exemplo de Código 1.19: Conversão de tipos primitivos.

Observe que, na conversão de `float` para `int`, o valor é truncado, ou seja, a parte decimal é descartada. Além disso, na conversão de *string* para número, é necessário garantir que a *string* represente realmente um número válido, pois caso a *string* não corresponda um valor numérico válido, uma exceção do tipo *ValueError* será gerada. Esse tipo de risco ocorre principalmente quando o lidamos com valores digitados pelo usuário. Nos próximos seguintes, aprenderemos a usar recursos que nos permitirão criar programas menos suscetíveis a esses problemas.

1.4 Entrada e Saída de Dados

A linguagem Python possui comandos nativos para entrada e saída de dados em programas console. Através desses comandos, o usuário pode interagir com seu programa, digitando valores para gerar resultados. É possível também formatar de diferentes maneiras as saídas textuais geradas pelo programa. As subseções a seguir tratam desses tópicos.

1.4.1 Entrada de Dados

Em uma aplicação console em Python, a entrada de dados geralmente é feita usando a função `input()` e a saída de dados é feita usando a função `print()`. A função `input()` permite que o usuário forneça dados para o programa. Quando a função é chamada, ela exibe uma mensagem na tela e aguarda a entrada do usuário. O valor digitado pelo usuário é armazenado como uma *string*. O exemplo 1.20 ilustra o uso da função `input()`.

```
1 nome = input("Digite seu nome: ")
2 print("Olá, " + nome + "!")
```

Exemplo de Código 1.20: Entrada de dados em console

1.4.2 Saída de Dados

A função `print()` é usada para exibir resultados ou mensagens na tela console. Ela pode exibir uma *string* ou o resultado de uma expressão. O exemplo 1.21 ilustra o uso da função `print()`.

```
1 nome = "Lucas"
2 print("Olá, " + nome + "!") # Saída: Olá, Lucas!
```

Exemplo de Código 1.21: Saída de dados em interface console.

1.4.3 Saída Formatada

Para imprimir uma saída formatada em Python, você pode usar o método `format()` ou o operador de formatação de *string* %. O método `format()` é uma forma fácil de formatar *strings* e incorporar valores de variáveis nelas. O exemplo 1.22 mostra como usá-lo.

```
1 nome = "Lucas"
2 idade = 30
3 print("Meu nome é {} e tenho {} anos".format(nome, idade))
4 # Saída: Meu nome é Lucas e tenho 30 anos
```

Exemplo de Código 1.22: O método format().

O operador de formatação de *string* % é uma forma mais antiga de formatar *strings*. O exemplo 1.23 mostra como usá-lo.

```
nome = "Lucas"
idade = 30
print("Meu nome é %s e tenho %d anos" % (nome, idade))
# Saída: Meu nome é Lucas e tenho 30 anos
```

Exemplo de Código 1.23: O operador de formatação %.

O operador `f` é uma forma mais recente e conveniente de formatar *strings* em Python. Ele permite que você insira valores de variáveis diretamente na *string*, usando chaves e o prefixo `f`. O exemplo 1.24 mostra como usá-lo.

```
nome = "Lucas"
idade = 30
print(f"Meu nome é {nome} e tenho {idade} anos")
# Saída: Meu nome é Lucas e tenho 30 anos
```

Exemplo de Código 1.24: O operador de formatação `f`.

O operador `f` é uma forma mais legível e concisa de formatar *strings* em comparação com outros métodos e é amplamente utilizado na comunidade Python. Ele também permite a execução de expressões dentro das chaves ({}) e a formatação avançada de números, entre outras coisas.

1.5 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você faça uso dos recursos de formatação de *strings* para apresentar uma saída textual bem formatada.

Exercício 1.1 Armazene seu nome e idade em variáveis separadas e imprima uma saída formatada com elas. ■

Exercício 1.2 Armazene dois números em variáveis e imprima a soma, subtração, multiplicação e divisão deles. ■

Exercício 1.3 Peça ao usuário para digitar três números, armazene-os em variáveis e imprima a média aritmética dos três números. ■

Exercício 1.4 Peça ao usuário para digitar seu peso e altura, calcule o índice de massa corporal (IMC) e imprima o resultado. ■

Exercício 1.5 Peça ao usuário para digitar três números e imprima a soma deles. ■

Exercício 1.6 Peça ao usuário para digitar um número e calcule o seu dobro. ■

Exercício 1.7 Peça ao usuário para digitar o nome, o preço de custo, o preço de venda e a quantidade em estoque de determinado produto e mostre o lucro que esse estoque pode gerar se todos os produtos forem vendidos. ■

Exercício 1.8 Peça ao usuário para digitar um número e calcule a raiz quadrada desse número. ■

Exercício 1.9 Peça ao usuário para digitar um número e calcule o seno, cosseno e tangente desse número. ■

Exercício 1.10 Peça ao usuário para digitar dois números e calcule a potência do primeiro número elevado ao segundo. ■

Exercício 1.11 Peça ao usuário para digitar três números e calcule a fórmula de Bhaskara para esses números. ■

Exercício 1.12 Peça ao usuário para digitar o raio de um círculo e calcule a área e o comprimento do círculo. ■

Exercício 1.13 Peça ao usuário para digitar as dimensões de um retângulo (largura e altura) e calcule a área e o perímetro desse retângulo. ■

Exercício 1.14 Peça ao usuário para digitar a base e a altura de um triângulo e calcule a área desse triângulo. ■

Exercício 1.15 Peça ao usuário para digitar a distância e a velocidade inicial de um objeto em queda livre e calcule o tempo que leva para atingir o solo, desconsiderando a resistência do ar. ■

Exercício 1.16 Peça ao usuário para digitar o valor inicial de um investimento, a taxa de juros e o número de anos e calcule o valor final do investimento. ■

Exercício 1.17 Peça ao usuário para digitar o preço de uma mercadoria, o desconto e o imposto e calcule o preço final da mercadoria. ■

Exercício 1.18 Peça ao usuário para digitar a massa e a aceleração de um objeto e calcule a força resultante. ■

Exercício 1.19 Peça ao usuário para digitar a velocidade final, a velocidade inicial e o tempo e calcule a aceleração. ■

Exercício 1.20 Peça ao usuário para digitar o valor da medida de um ângulo em radianos e calcule o valor desse ângulo em graus. ■

Exercício 1.21 Peça ao usuário para digitar o comprimento de dois lados de um triângulo retângulo e calcule o comprimento da hipotenusa. ■

Exercício 1.22 Peça ao usuário para digitar a distância percorrida por um objeto e o tempo gasto e calcule a velocidade média do objeto. ■

Exercício 1.23 Peça ao usuário para digitar a distância percorrida, o tempo gasto e aceleração e calcule a velocidade inicial e final do objeto. ■

Exercício 1.24 Calcular o perímetro de um círculo dado o seu raio como entrada. ■

Exercício 1.25 Calcular o volume de uma esfera dado o seu raio como entrada. ■

Exercício 1.26 Calcular a área de um triângulo retângulo dado as medidas dos seus catetos como entrada. ■

Exercício 1.27 Receber o nome, o salário e o valor do imposto de uma pessoa como entrada e imprimir o salário líquido. ■

Exercício 1.28 Crie uma lista com os nomes de 5 membros da sua família. ■

Exercício 1.29 Crie um dicionário com os nomes, idades e cor dos olhos de 5 (cinco) membros da sua família. ■

Exercício 1.30 Crie dois conjuntos e combine-os usando as operações de união, interseção e diferença, apresentando os resultados de cada operação. ■

1.6 Considerações Sobre o Capítulo

Este capítulo apresentou os fundamentos da programação na linguagem Python. Foi mostrado como criar variáveis de diferentes tipos, como utilizar os operadores matemáticos básicos e como usar os comandos de entrada e saída de dados para se criar programas mais interativos. Por fim, foram propostos diversos exercícios para você colocar em prática aquilo que aprendeu ao longo do capítulo. No próximo capítulo, você verá como executar códigos baseados em condições através das estruturas condicionais.

2. Estruturas Condicionais

As estruturas condicionais são parte fundamental da programação, pois permitem que o programa tome decisões e execute ações diferentes de acordo com as condições especificadas. Isso é importante porque o programa pode se adaptar a diferentes situações e entrada de dados, tornando-se mais flexível e capaz de lidar com problemas complexos.

Por exemplo, imagine que você esteja escrevendo um programa para calcular o salário de um funcionário. Sem a utilização de estruturas condicionais, o programa sempre calcularia o salário de acordo com as mesmas regras, independentemente do funcionário em questão. No entanto, se você utilizar estruturas condicionais, poderá incluir regras específicas para cada funcionário, levando em consideração fatores como horas extras, férias etc.

De maneira geral, as estruturas condicionais tornam os programas mais inteligentes e capazes de lidar com situações variadas, ajudando a garantir que as decisões tomadas pelo programa sejam as mais adequadas para cada caso específico.

2.1 Sintaxe Básica

Seguindo essa linha de pensamento, as estruturas condicionais em Python permitem que você execute ações diferentes de acordo com a verificação de determinadas condições. A sintaxe de uso de uma estrutura condicional básica em Python é apresentada no exemplo 2.1.

Vale ressaltar que os blocos `elif` e `else` não são de uso obrigatório. Portanto, uma estrutura condicional simples poderia ter somente o bloco correspondente ao `if`, que só seria executado se a condição fosse verdadeira. O código 2.2 ilustra essa situação.

```
if condição1:
    # Execute alguma ação se a condição 1 for verdadeira
elif condição2:
    # Execute alguma ação se a condição 1 for falsa a condição 2 for verdadeira
else:
    # Execute alguma outra ação se a condição for falsa
```

Exemplo de Código 2.1: A estrutura condicional `if`

```
# Exemplo 1: verificação de idade
idade = int(input("Digite sua idade: "))
if idade >= 18:
    print("Você é maior de idade.")

# Exemplo 2: verificação de número par
numero = int(input("Digite um número: "))
if numero % 2 == 0:
    print("O número é par.")

#Exemplo 3: verificação de nota
nota = float(input("Digite sua nota: "))
if nota >= 7:
    print("Você foi aprovado.")
```

Exemplo de Código 2.2: Estruturas condicionais simples

2.1.1 Estrutura Condicional Composta

A estrutura condicional composta é uma estrutura de controle de fluxo que permite a execução de diferentes trechos de código com base em diferentes condições. Ela é composta pelo comando `if`, `elif` (abreviação de `else if`) e `else`. A sintaxe básica de uma estrutura condicional composta é mostrada nos exemplos do código 2.3.

Em síntese, quando se tem mais de uma condição a ser avaliada, como no exemplo do código 2.3, você pode utilizar a estrutura `elif` (uma ou várias) para verificar cada uma das condições, uma a uma. Se a primeira condição não for verdadeira, o programa passa para a próxima condição verificada pelo `elif`, e assim por diante, até que uma das condições seja verificada como verdadeira ou até que seja atingido o `else` (caso nenhuma das condições seja verdadeira).

2.2 Combinação de Condições com Operadores Lógicos

É possível combinar várias condições em uma única estrutura condicional. Existem várias maneiras de combinar condições, dependendo do que você deseja verificar. Esta seção abora

```python
# Exemplo 1: verificação de idade
idade = int(input("Digite sua idade: "))
if idade < 18:
    print("Você é menor de idade.")
elif idade >= 18 and idade < 60:
    print("Você é maior de idade.")
else:
    print("Você é idoso.")

# Exemplo 2: verificação de nota
nota = float(input("Digite sua nota: "))
if nota >= 9:
    print("Você foi aprovado com louvor.")
elif nota >= 7:
    print("Você foi aprovado.")
else:
    print("Você foi reprovado.")
```

Exemplo de Código 2.3: Exemplos de uso da estrutura condicional composta

os operadores lógicos usados para combinação de condições.

2.2.1 O Operador `AND`

O operador lógico AND é utilizado para combinar duas ou mais condições lógicas e verificar se todas elas são verdadeiras ao mesmo tempo. O operador AND é representado pela palavra reservada `and` na sintaxe do Python. O código 2.4 mostra alguns exemplos de uso do operador AND em Python.

2.2.2 O Operador `OR`

O operador lógico OR é utilizado para combinar duas ou mais condições lógicas e verificar se pelo menos uma delas é verdadeira. O operador OR é representado pela palavra reservada `or` na sintaxe do Python. O código 2.5 mostra alguns exemplos de uso do operador OR em Python.

Estes são alguns exemplos de como o operador lógico OR pode ser usado em Python. Em cada exemplo, o operador OR é usado para combinar duas ou mais condições lógicas e verificar se pelo menos uma delas é verdadeira, permitindo uma tomada de decisão mais avançada.

2.2.3 O Operador Lógico NOT

O operador lógico NOT em Python é usado para inverter o valor de uma condição lógica. Em outras palavras, se uma condição é verdadeira, o operador NOT fará com que ela se torne falsa

```python
#Exemplo 1: autenticação de usuário
usuario = input("Digite seu nome de usuário: ")
senha = input("Digite sua senha: ")
if usuario == "admin" and senha == "123":
    print("Você tem permissão de acesso.")
else:
    print("Você não tem permissão de acesso.")

#Exemplo 2: verificação de horário de trabalho
hora = int(input("Digite a hora atual: "))
dia = input("Digite o dia da semana: ")
if hora >= 9 and hora <= 18 and dia != "sábado" and dia != "domingo":
    print("Você está no horário de trabalho.")
else:
    print("Você não está no horário de trabalho.")
```

Exemplo de Código 2.4: Exemplos de uso do operador AND

```python
# Exemplo 1: verificação do horário de funcionamento
dia_da_semana = input("Digite o dia da semana: ")
hora = int(input("Digite a hora atual: "))
if dia_da_semana == "sábado" or dia_da_semana == "domingo" or hora < 9 or hora >= 17:
    print("Loja fechada.")
else:
    print("Loja aberta.")

# Exemplo 2: verificação de feriado
dia = int(input("Digite o dia: "))
mes = int(input("Digite o mês: "))
if dia == 1 and mes == 1 or dia == 25 and mes == 12:
    print("Hoje é feriado.")
else:
    print("Hoje não é feriado.")
```

Exemplo de Código 2.5: Exemplos de uso do operador OR

e vice-versa. O operador NOT é representado por `not` na sintaxe do Python. O código 2.6 mostra alguns exemplos de uso do operador NOT.

Estes são apenas alguns exemplos de como o operador NOT pode ser usado em Python. Em cada exemplo, o operador NOT é usado para inverter o valor de uma condição lógica, permitindo uma tomada de decisão mais avançada.

```python
# Exemplo 1: verificação de maioridade
idade = int(input("Digite sua idade: "))
if not idade >= 18:
    print("Você não tem mais de 18 anos.")
else:
    print("Você tem mais de 18 anos.")

# Exemplo 2: verificação de horário de trabalho
hora = int(input("Digite a hora atual: "))
if not (hora >= 9 and hora <= 18):
    print("Você não está no horário de trabalho.")
else:
    print("Você está no horário de trabalho.")
```

Exemplo de Código 2.6: Exemplos de uso do operador NOT

2.3 Comparação de Valores em Condições

Os operadores de comparação são utilizados para comparar valores e determinar se uma determinada condição é verdadeira ou falsa. Aqui estão alguns dos principais operadores de comparação em Python:

- == (*igual a*) : Verifica se os valores são iguais;
- != (*diferente de*) : Verifica se os valores são diferentes;
- < (*menor que*) : Verifica se o valor à esquerda é menor que o valor à direita;
- > (*maior que*) : Verifica se o valor à esquerda é maior que o valor à direita;
- <= (*menor ou igual a*) : Verifica se o valor à esquerda é menor ou igual ao valor à direita;
- >= (*maior ou igual a*) : Verifica se o valor à esquerda é maior ou igual ao valor à direita.

No código 2.7 estão alguns exemplos de como esses operadores podem ser usados.

Em geral, os operadores de comparação são uma parte fundamental da programação, pois permitem que o programa tome decisões baseadas nas condições especificadas, tornando-o mais flexível e capaz de lidar com situações variadas.

2.4 Estruturas Condicionais Aninhadas

As estruturas condicionais aninhadas são estruturas condicionais dentro de outras estruturas condicionais. Em outras palavras, você pode colocar uma estrutura condicional dentro de outra para verificar condições mais complexas. Isso pode ser útil quando você precisa verificar várias

```python
# Exemplo 1: Verificação de igualdade
a = 5
b = 5
if a == b:
    print("a é igual a b")
else:
    print("a é diferente de b")
# Saída: a é igual a b

# Exemplo 2: Verificação de desigualdade
a = 5
b = 3
if a != b:
    print("a é diferente de b")
else:
    print("a é igual a b")
# Saída: a é diferente de b

# Exemplo 3: Verificação de número par ou ímpar
numero = 4
if numero % 2 == 0:
    print(numero, "é par")
else:
    print(numero, "é ímpar")
# Saída: 4 é par
```

Exemplo de Código 2.7: Exemplos de uso dos operadores de comparação

condições em sequência ou quando precisa tomar decisões com base em várias condições diferentes. O código 2.8 mostra um exemplo de estrutura condicional aninhada.

```python
idade = 25
pais = "Brasil"
if idade >= 18:
    if pais == "Brasil":
        print("Você pode votar no Brasil.")
    else:
        print("Você pode votar em outro país.")
else:
    print("Você não pode votar.")
# Saída: Você pode votar no Brasil.
```

Exemplo de Código 2.8: Exemplo de estrutura condicional aninhada.

Nesse exemplo, a primeira estrutura condicional verifica se a idade é maior ou igual a 18. Se for, a segunda estrutura condicional verifica se o país é o Brasil. Se for, a mensagem “Você pode

votar no Brasil." é exibida. Se não for, a mensagem "Você pode votar em outro país." é exibida. Se a primeira condição não for verdadeira, a mensagem "Você não pode votar." é exibida.

É importante lembrar que as estruturas condicionais aninhadas podem tornar o código mais complexo e difícil de entender, então é importante usá-las com moderação e cuidado. Se o código estiver ficando muito complexo, pode ser uma boa ideia refatorá-lo em funções ou em classes para torná-lo mais legível e fácil de manter.

2.5 Operador de Atribuição Condicional Ternário

Existe um operador de atribuição ternário em Python. É uma forma concisa de escrever uma estrutura condicional que retorna um valor diretamente para uma atribuição de variável. O operador ternário é escrito da forma mostrada no código 2.9.

```
valor_verdadeiro if condição else valor_falso
```

Exemplo de Código 2.9: O operador ternário de atribuição condicional

O código 2.10 mostra um exemplo de como o operador ternário pode ser usado.

```
idade = 25
status = "Maior de idade" if idade >= 18 else "Menor de idade"
print(status)
# Saída: Maior de idade
```

Exemplo de Código 2.10: Exemplo de uso do operador ternário

Nesse exemplo, o operador ternário verifica se a idade é maior ou igual a 18. Se for, a *string* "Maior de idade" é atribuída à variável `status`. Se não for, a *string* "Menor de idade" é atribuída à variável `status`.

O operador ternário é uma forma concisa e eficiente de escrever estruturas condicionais simples, mas é importante lembrar que ele pode tornar o código mais difícil de entender se for usado de forma excessiva ou em casos mais complexos. Em geral, é uma boa ideia usar o operador ternário com moderação e optar por estruturas condicionais mais explícitas quando o código se torna mais complexo.

2.6 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 2.1 Escreva um programa que verifique se um número é par ou ímpar. ■

Exercício 2.2 Escreva um programa que verifique se um número digitado pelo usuário é positivo, negativo ou zero. ■

Exercício 2.3 Escreva um programa que verifique se uma letra digitada pelo usuário é uma vogal ou consoante. ■

Exercício 2.4 Escreva um programa que verifique se um ano é bissexto ou não (pesquise o que caracteriza um ano como bissexto). ■

Exercício 2.5 Escreva um programa que, a partir da idade digitada pelo usuário, verifique se ele é maior de idade ou não . ■

Exercício 2.6 Escreva um programa que verifique se uma *string* é um palíndromo ou não (não usar estrutura de repetição). ■

Exercício 2.7 Escreva um programa que verifique se uma *string* é um número inteiro ou não e mostre uma mensagem de acordo (pode usar estrutura de repetição). ■

Exercício 2.8 Escreva um programa que verifique se uma *string* é um número real ou não (pode usar estrutura de repetição). ■

Exercício 2.9 Escreva um programa que verifique se uma *string* digitada pelo usuário é uma data no formato mm/dd/aaaa ou não. ■

Exercício 2.10 Escreva um programa que calcule a média de três números e exiba uma mensagem de “Aprovado” se a média for maior ou igual a 6, ou “Reprovado” caso contrário. Se a nota for 10, exiba também a mensagem “Parabéns”. ■

Exercício 2.11 Escreva um programa que verifique se uma temperatura está acima, abaixo ou dentro da faixa normal (36°C a 37°C). ■

Exercício 2.12 Escreva um programa que verifique se uma pessoa pode votar ou não (se tem 18 anos ou mais e se é brasileira). ■

Exercício 2.13 Escreva um programa que verifique se uma pessoa é elegível para aposentadoria (se tem 60 anos ou mais para mulheres e 65 anos ou mais para homens). ■

Exercício 2.14 Escreva um programa que verifique se um número inteiro digitado pelo usuário é divisível por outro número inteiro digitado pelo usuário ou não. ■

Exercício 2.15 Escreva um programa que pergunte ao usuário seu salário e exiba uma mensagem de "Alto salário" se o salário for maior do que R$10.000,00, ou "Baixo salário" caso contrário. ■

Exercício 2.16 Escreva um programa que pergunte ao usuário seu gênero (M para masculino, F para feminino) e exiba uma mensagem de "Gênero masculino" ou "Gênero feminino". ■

Exercício 2.17 Escreva um programa que pergunte ao usuário seu peso e altura e exiba uma mensagem de "Você está abaixo do peso" se o IMC (índice de massa corporal) for menor do que 18,5, "Você está com o peso normal" se o IMC estiver entre 18,5 e 24,9, "Você está com sobrepeso" se o IMC estiver entre 25 e 29,9, ou "Você está com obesidade" caso contrário. ■

Exercício 2.18 Escreva um programa que pergunte ao usuário sua idade e exiba uma mensagem de "Você é jovem" se a idade for menor do que 30, "Você é adulto" se a idade estiver entre 30 e 60, ou "Você é idoso" caso contrário. ■

Exercício 2.19 Escreva um programa que peça ao usuário para digitar 5 números inteiros. O programa deve exibir uma mensagem informando se todos os números digitados são pares ou se há pelo menos um número ímpar. ■

2.7 Considerações Sobre o Capítulo

Este capítulo apresentou os conceitos e formas de uso da estrutura condicional `if` na linguagem Python. Foi mostrado como usar a estrutura `if`, quais são os operadores lógicos, quais são

os operadores de comparação, como usar estruturas `if` aninhadas, como usar o operador de atribuição condicional ternário e, concluindo, foram apresentados várias propostas de exercícios. No próximo capítulo, você verá como repetir a execução de um trecho de código usando as estruturas condicionais.

3. Estruturas de Repetição

As estruturas de repetição são uma das ferramentas mais importantes na programação, pois permitem que o código seja executado várias vezes, o que é útil em muitos casos. Elas são utilizadas para repetir uma sequência de instruções até que uma determinada condição seja atendida. Algumas das características das estruturas de repetição incluem:

- Controle de repetição: As estruturas de repetição permitem que o programador controle quantas vezes o código será executado, seja por meio de uma contagem fixa ou através de uma condição que deve ser atendida;
- Flexibilidade: As estruturas de repetição são muito flexíveis e permitem que o programador escolha o número de vezes que o código será executado, ou até mesmo deixar que o código seja executado indefinidamente;
- Automatização de tarefas repetitivas: As estruturas de repetição permitem que tarefas repetitivas sejam automatizadas, economizando tempo e esforço para o programador;
- Processamento de dados em massa: As estruturas de repetição permitem processar grandes quantidades de dados de uma só vez, o que é útil em aplicações como processamento de dados financeiros, análise de dados de saúde, etc.

Em síntese, as estruturas de repetição são importantes para a programação porque permitem automatizar tarefas repetitivas e a processar grandes quantidades de dados, além de fornecer flexibilidade e controle de repetição para o programador.

3.1 Sintaxe Básica

Em Python, existem duas estruturas de repetição principais: o laço `for` e o laço `while`. O laço `for` é usado para repetir uma ação um número fixo de vezes. Por exemplo, você pode usar um

laço for para imprimir todos os elementos de uma lista, como mostra o código 3.1.

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

Exemplo de Código 3.1: Exemplos de uso do laço `for`

O laço `while` é usado para repetir uma ação enquanto uma determinada condição for verdadeira. Por exemplo, você pode usar um laço `while` para ler números digitados pelo usuário até que ele digite um número negativo, como mostra o código 3.2.

```
numero = int(input("Digite um número: "))
while numero >= 0:
    numero = int(input("Digite outro número: "))
print("Você digitou um número negativo.")
```

Exemplo de Código 3.2: Exemplos de uso do laço `while`

Estas são as duas principais estruturas de repetição em Python. Cada uma delas tem suas próprias vantagens e usos, e é importante conhecer ambas para escolher a mais adequada para a tarefa em questão. As seções seguintes apresentam maiores detalhes sobre cada uma dessas duas estruturas.

3.2 A Estrutura de Repetição *for*

Como introduzido na seção anterior, a estrutura de repetição `for` em Python é usada para repetir uma ação um número fixo de vezes. Ela funciona percorrendo uma sequência de elementos, como uma lista ou uma *strings*, e executando uma ação para cada elemento da sequência. A sintaxe geral da estrutura de repetição for é apresentada no código 3.3.

```
for elemento in sequencia:
    # código a ser executado para cada elemento da sequência
```

Exemplo de Código 3.3: Sintaxe geral da estrutura `for`

No código 3.3, `elemento` é uma variável que representa cada elemento da sequencia na iteração atual e `sequencia` é uma sequência de elementos, como uma lista, *string*, *range*, entre outros. O código 3.4 mostra exemplos mais específicos de uso do `for`.

Estes são apenas alguns exemplos de como a estrutura de repetição `for` pode ser usada em Python. Em cada exemplo, a estrutura `for` é usada para repetir uma ação para cada elemento

```python
# Exemplo 1: impressão dos elementos de uma lista
frutas = ["maçã", "banana", "laranja"]
for fruta in frutas:
    print(fruta)

# Exemplo 2: soma dos elementos de uma lista
numeros = [1, 2, 3, 4, 5]
soma = 0
for numero in numeros:
    soma += numero
print("A soma dos números é", soma)

# Exemplo 3: impressão de caracteres de uma string
nome = "João"
for letra in nome:
    print(letra)

# Exemplo 4: cálculo de fatorial
numero = 5
fatorial = 1
for i in range(1, numero + 1):
    fatorial *= i
print("O fatorial de", numero, "é", fatorial)
```

Exemplo de Código 3.4: Exemplos de uso da estrutura `for`

de uma sequência, permitindo a realização de tarefas mais avançadas e eficientes.

3.3 A Estrutura de Repetição *while*

A estrutura de repetição `while` em Python é usada para repetir uma ação enquanto uma determinada condição é verdadeira. A estrutura `while` funciona verificando a condição no início de cada iteração e, se a condição for verdadeira, a ação é executada. A sintaxe geral da estrutura de repetição `while` é mostrada na figura 3.5.

```python
while condicao:
    # código a ser executado enquanto a condição for verdadeira
```

Exemplo de Código 3.5: Sintaxe geral da estrutura `while`

No código 3.5, `condicao` é uma expressão lógica que é avaliada antes de cada iteração do laço. Se a condição for verdadeira, o código dentro do laço será executado, caso contrário, o laço será encerrado. É importante lembrar de incluir código dentro do laço para alterar a condição de forma que, eventualmente, ela se torne falsa, caso contrário, o laço se tornará infinito, travando

a execução do programa. O código 3.6 mostra exemplos mais específicos de uso da estrutura `while` em Python.

```
# Exemplo 1: leitura de números positivos
numero = int(input("Digite um número: "))
while numero >= 0:
    print("Você digitou o número", numero)
    numero = int(input("Digite outro número: "))
print("Você digitou um número negativo.")

# Exemplo 2: impressão de números pares
numero = 0
while numero <= 10:
    print(numero)
    numero += 2

# Exemplo 3: cálculo da média para vários números
soma = 0
contador = 0
media = 0
numero = float(input("Digite um número: "))
while numero >= 0:
    soma += numero
    contador += 1
    media = soma / contador
    numero = float(input("Digite outro número: "))
print("A média dos números é", media)
```

Exemplo de Código 3.6: Exemplos de uso da estrutura `while`

Estes são apenas alguns exemplos de como a estrutura de repetição `while` pode ser usada em Python. Em cada exemplo, a estrutura `while` é usada para repetir uma ação enquanto uma determinada condição é verdadeira, permitindo a realização de tarefas mais avançadas e eficientes.

3.4 Controle de Fluxo com *break* e *continue*

Os comandos break e continue são usados para controlar o fluxo de execução dentro de laços em Python. Eles permitem interromper ou pular iterações de um laço, respectivamente. O comando `break` é usado para interromper um laço prematuramente. Quando o comando `break` é executado dentro de um laço, o laço é imediatamente encerrado, independentemente da condição de repetição. O código 3.7 mostra um exemplo de uso do comando `break` que pede ao usuário que digite um número até que ele digite um valor negativo.

```
numero = int(input("Digite um número: "))
while True:
    if numero < 0:
        break
    print("Você digitou o número", numero)
    numero = int(input("Digite outro número: "))
print("Você digitou um número negativo.")
```

Exemplo de Código 3.7: Exemplo de uso do comando `break`

O outro comando de controle de fluxo é o `continue`, que é usado para pular uma iteração de um laço. Quando o comando continue é executado dentro de um laço, a iteração atual é imediatamente encerrada e a próxima iteração é iniciada. O código 3.8 mostra um exemplo de uso do comando `continue` que imprime os números ímpares de 0 a 9.

```
for numero in range(10):
    if numero % 2 == 0:
        continue
    print(numero)
```

Exemplo de Código 3.8: Exemplo de uso do comando `continue`

Estes são os conceitos básicos sobre como usar os comandos `break` e `continue` para controlar o fluxo de execução dentro de laços em Python. É importante lembrar que, embora estes comandos possam ser úteis em muitas situações, eles também podem tornar o código mais difícil de ler e manter, portanto, é importante usá-los com moderação.

3.5 Estruturas de Repetição Aninhadas

A utilização de laços dentro de outros laços é uma técnica importante na programação que permite realizar tarefas mais complexas. Isso é conhecido como aninhamento de laços. Quando você aninha laços, você pode repetir ações dentro de outras ações repetidas, permitindo a realização de tarefas mais complexas. A sintaxe geral para aninhar laços em Python é mostrada no código 3.9.

```
for elemento_externo in sequencia_externa:
    # código a ser executado para cada elemento da sequência externa
    for elemento_interno in sequencia_interna:
        # código a ser executado para cada elemento da sequência interna
```

Exemplo de Código 3.9: Sintaxe geral do uso de laços aninhados

No código 3.9, as variáveis `elemento_externo` e `elemento_interno` representam os elementos de `sequencia_externa` e `sequencia_interna` nas iterações atuais, respectivamente. O código 3.10 mostra exemplos de uso de laços aninhados em Python.

```
# Exemplo 1: impressão de tabuada
for i in range(1, 11):
    for j in range(1, 11):
        print(i, "x", j, "=", i * j)

# Exemplo 2: verificação de número primo
numero = 17
e_primo = True
for i in range(2, numero):
    if numero % i == 0:
        e_primo = False
        break
if e_primo:
    print(numero, "é primo.")
else:
    print(numero, "não é primo.")
```

Exemplo de Código 3.10: Exemplos de uso de laços aninhados

Estes são apenas alguns exemplos de como laços aninhados podem ser usados em Python para realizar tarefas mais complexas. É importante lembrar que, embora laços aninhados possam ser úteis em muitas situações, eles também podem tornar o código mais difícil de ler e manter, portanto, é importante usá-los com moderação.

3.6 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 3.1 Imprima todos os números de 1 a 100. ■

Exercício 3.2 Imprima todos os números pares de 1 a 100. ■

Exercício 3.3 Imprima todos os números ímpares de 1 a 100. ■

Exercício 3.4 Soma todos os números de 1 a 100. ■

Exercício 3.5 Soma todos os números pares de 1 a 100. ■

Exercício 3.6 Soma todos os números ímpares de 1 a 100. ■

Exercício 3.7 Calcule a média dos números digitados pelo usuário. O usuário deve digitar números até digitar um número negativo. ■

Exercício 3.8 Calcule o fatorial de um número digitado pelo usuário. ■

Exercício 3.9 Imprima todos os números divisíveis por 3 ou 5 de 1 a 100. ■

Exercício 3.10 Imprima todos os números primos de 1 a 100. ■

Exercício 3.11 Soma todos os números divisíveis por 3 ou 5 de 1 a 100. ■

Exercício 3.12 Soma todos os números primos de 1 a 100. ■

Exercício 3.13 Imprima a tabuada de um número digitado pelo usuário. ■

Exercício 3.14 Verifique se um número digitado pelo usuário é primo. ■

Exercício 3.15 Verifique se um número digitado pelo usuário é perfeito. ■

Exercício 3.16 Encontre o maior e o menor número em uma lista de 10 números digitada pelo usuário. ■

Exercício 3.17 Encontre o segundo maior e o segundo menor número em uma lista de 10 números digitada pelo usuário. ■

Exercício 3.18 Imprimir a sequência de Fibonacci até o n-ésimo termo, onde n é digitado pelo usuário. ■

Exercício 3.19 Verificar se uma palavra digitada pelo usuário é um palíndromo. Se for, imprimir, ao final, “A palavra é um palíndromo”. Se não for, imprimir “A palavra não é um palíndromo”. ■

Exercício 3.20 Calcular o fatorial de todos os números de 1 a n, onde n é digitado pelo usuário. Imprima o resultado para cada número. ■

3.7 Considerações Sobre o Capítulo

Este capítulo apresentou os conceitos e formas de uso das estruturas de repetição `for` e `while` na linguagem Python. Foi mostrado como usar cada uma das estruturas, bem como os comandos de controle de fluxo `break` e `continue` e as estruturas de repetição aninhadas, concluindo, com uma série de exercícios propostos. No próximo capítulo, você verá como manipular números e textos de forma mais avançada.

4. Números e Textos

Python é uma linguagem de programação poderosa e flexível que permite manipular números, textos e coleções de maneira um pouco mais avançada. Com algumas funções e bibliotecas nativas, você pode realizar operações complexas em Python. As seções seguintes tratam individualmente da manipulação avançada de números, textos e coleções.

4.1 Operações Matemáticas Avançadas

A manipulação de números é fundamental para a programação e é essencial para uma grande variedade de campos de computação, desde o desenvolvimento de jogos e aplicativos até a ciência de dados e inteligência artificial. A programação envolve resolver problemas complexos e criar soluções eficientes e eficazes para esses problemas. A matemática fornece as ferramentas necessárias para realizar esses cálculos complexos e ajudar a automatizar tarefas que seriam impossíveis de realizar manualmente. A matemática em si pode ajudar a programar sistemas de inteligência artificial e *machine learning* que são capazes de aprender e se adaptar a novas situações. Na programação, a matemática é usada para uma variedade de tarefas, incluindo:

- Cálculos aritméticos básicos, como adição, subtração, multiplicação e divisão;
- Cálculos avançados, como álgebra, trigonometria, cálculo e estatística;
- Modelagem e simulação de fenômenos físicos, como a física de partículas e a dinâmica de fluidos;
- Desenvolvimento de algoritmos e estruturas de dados, como grafos e árvores, que são usados em uma variedade de aplicativos;
- Implementação de técnicas de criptografia e segurança, que envolvem cálculos complexos para garantir a privacidade e a segurança dos dados.

A linguagem Python possui uma vasta biblioteca de funções para operações matemáticas avançadas. Além das operações básicas de adição, subtração, multiplicação e divisão, existem muitas funções disponíveis para cálculos mais avançados. Alguns exemplos de operações matemáticas avançadas em Python são abordados nas subseções a seguir.

4.1.1 Potenciação

A potenciação é uma operação matemática que envolve a multiplicação repetida de um número por ele mesmo. Em Python, isso é feito com o operador ** ou a função `pow()`. O código 4.1 mostra alguns exemplos.

```
a = 2
b = 3
print(a ** b)  # imprime 8
print(pow(a, b))  # imprime 8
```

Exemplo de Código 4.1: Operações de potenciação.

4.1.2 Radiciação

A radiciação é a operação inversa da potenciação. É usada para encontrar a raiz de um número. Em Python, isso é feito com a função `sqrt()` da biblioteca `math`. O código 4.2 mostra alguns exemplos.

```
import math
a = 16
print(math.sqrt(a))  # imprime 4.0
```

Exemplo de Código 4.2: Operação de radiciação.

4.1.3 Trigonometria

Para realizar cálculos trigonométricos em Python, podemos usar a biblioteca `math`, que fornece funções para calcular seno, cosseno, tangente e outras funções trigonométricas. Essas funções são muito úteis em problemas de matemática, física e engenharia, que frequentemente envolvem ângulos e triângulos. Aqui estão algumas das funções trigonométricas mais comuns disponíveis na biblioteca `math` em Python:

- `math.sin(x)`: calcula o seno de um ângulo em radianos;
- `math.cos(x)`: calcula o cosseno de um ângulo em radianos;
- `math.tan(x)`: calcula a tangente de um ângulo em radianos;
- `math.asin(x)`: calcula o arco-seno de um número, com resultado em radianos;

- `math.acos(x)`: calcula o arco-cosseno de um número, com resultado em radianos;
- `math.atan(x)`: calcula o arco-tangente de um número, com resultado em radianos;
- `math.degrees(x)`: converte um ângulo em radianos para graus;
- `math.radians(x)`: converte um ângulo em graus para radianos.

O código 4.3 mostra alguns exemplos.

```
import math

# Calcula o seno de 30 graus
angulo = 30
radianos = math.radians(angulo)
seno = math.sin(radianos)
print(seno) # imprime 0.5

# Calcula o arco-cosseno de 0.5
numero = 0.5
arco_cosseno = math.acos(numero)
print(math.degrees(arco_cosseno)) # imprime 60.0
```

Exemplo de Código 4.3: Operações trigonométricas.

Neste exemplo 4.3, o primeiro bloco de código calcula o seno de um ângulo de 30 graus. Primeiro, convertemos o ângulo de graus para radianos usando a função `radians()`. Em seguida, calculamos o seno do ângulo usando a função `sin()`. O resultado é armazenado na variável `seno` e impresso na tela. O segundo bloco de código calcula o arco-cosseno de 0,5. Primeiro, o número 0,5 é passado para a função `acos()` para calcular o arco-cosseno em radianos. Em seguida, usamos a função `degrees()` para converter o resultado de radianos para graus. O resultado é armazenado na variável `arco_cosseno` e impresso na tela.

Esses são apenas alguns exemplos de como podemos usar as funções trigonométricas da biblioteca `math` para realizar cálculos trigonométricos em Python.

4.1.4 Logaritmo

O logaritmo é a operação inversa da potenciação. É usado para encontrar o expoente necessário para produzir um determinado número. Em Python, a função `log()` da biblioteca `math` pode ser usada para calcular o logaritmo natural de um número. O código 4.4 mostra alguns exemplos.

```
import math
a = 100
print(math.log(a))  # imprime 4.605170185988092
```

Exemplo de Código 4.4: Operações sobre logaritmos.

4.1.5 Fatorial

O fatorial de um número é o produto de todos os números inteiros positivos menores ou iguais a ele. Em Python, a função `factorial()` da biblioteca `math` pode ser usada para calcular o fatorial de um número. O código 4.5 mostra um exemplo.

```
import math
a = 5
print(math.factorial(a))  # imprime 120
```

Exemplo de Código 4.5: Operações de fatorial.

4.1.6 Números Complexos

Os números complexos são números que contêm uma parte real e uma parte imaginária. Em Python, os números complexos são representados pelo sufixo j após a parte imaginária. O código 4.6 mostra alguns exemplos.

```
a = 2 + 3j
b = 4 - 5j
print(a + b)  # imprime (6-2j)
print(a * b)  # imprime (23-2j)
```

Exemplo de Código 4.6: Operações Sobre Números Complexos .

Em resumo, a matemática é uma ferramenta essencial na programação e permite que os desenvolvedores criem soluções eficientes e eficazes para problemas complexos. A capacidade de aplicar conceitos matemáticos na programação é uma habilidade valiosa que pode ajudar os desenvolvedores a se destacar em suas carreiras e contribuir para o desenvolvimento de tecnologias inovadoras.

4.2 Manipulação Avançada de Textos

A manipulação de texto é uma tarefa fundamental em muitos projetos de programação, especialmente quando se trabalha com dados não estruturados, como textos de redes sociais, artigos de notícias ou e-mails. Python é uma linguagem poderosa para manipulação de texto,

com muitas bibliotecas e métodos disponíveis para realizar operações avançadas em texto. Nesta seção, veremos algumas técnicas avançadas para manipulação de texto em Python, com exemplos práticos.

4.2.1 Limpeza

Antes de começar a manipular o texto, muitas vezes é necessário limpar os dados para remover caracteres indesejados ou transformar o texto em um formato mais uniforme. Considere, como exemplo, a remoção de caracteres não alfanuméricos, como emojis, símbolos e pontuação, que podem interferir na análise de texto. Para remover esses caracteres, podemos usar expressões regulares em Python. O código 4.7 mostra um exemplo de como usar esse recurso.

```
import re

texto = "Este é um texto com @caracteres! especiais #\$"

# remover caracteres especiais
texto_sem_especiais = re.sub('[^A-Za-z0-9]+', ' ', texto)

print(texto_sem_especiais) # Saída: Este é um texto com caracteres especiais
```

Exemplo de Código 4.7: Limpeza de textos via expressão regular.

Outro tipo de tratamento comum é a transformação de letras maiúsculas em minúsculas e vice-versa. Em muitos casos, é útil transformar todo o texto em letras maiúsculas ou minúsculas para torná-lo mais uniforme. Podemos fazer isso facilmente em Python com os métodos `upper()` e `lower()`. O código 4.8 mostra um exemplo de como usar esses métodos.

```
texto = "Este é Um TexTo com leTrAs MaIúsCulas e miNúsCulAs."

# transformar em letras minúsculas
texto_min = texto.lower()

# transformar em letras maiúsculas
texto_mai = texto.upper()

print(texto_min) # Saída: este é um texto com letras maiúsculas e minúsculas.
print(texto_mai) # Saída: ESTE É UM TEXTO COM LETRAS MAIÚSCULAS E MINÚSCULAS.
```

Exemplo de Código 4.8: Transformação de texto para maiúsculas e minúsculas.

4.2.2 Tokenização

A tokenização é o processo de dividir o texto em unidades menores, como palavras, frases ou parágrafos. Em Python, podemos usar a biblioteca NLTK (*Natural Language Toolkit*) para realizar a tokenização de texto. O código 4.9 mostra exemplos de uso desse recurso.

```
import nltk
texto = """Este é um texto de exemplo para tokenização.
    Ele contém várias frases diferentes."""
# dividir o texto em frases
frases = nltk.sent_tokenize(texto)
# dividir o texto em palavras
palavras = nltk.word_tokenize(texto)
print(frases)
# Saída: ['Este é um texto de exemplo para tokenização.',
#     'Ele contém várias frases diferentes.']
print(palavras)
# Saída: ['Este', 'é', 'um', 'texto', 'de', 'exemplo', 'para', 'tokenização', '.',
#     'Ele', 'contém', 'várias', 'frases', 'diferentes', '.']
```

Exemplo de Código 4.9: Tokenização de textos.

4.2.3 Análise de Sentimento

A análise de sentimento é uma técnica que permite determinar se um texto tem uma conotação positiva, negativa ou neutra. Em Python, podemos usar a biblioteca *TextBlob* para realizar análise de sentimento em texto. Para instalar a biblioteca, basta usar `pip install textblob`. Vale observar que a biblioteca *TextBlob* só funciona para o idioma inglês, porém, existem outras bibliotecas que podem ser preparadas para funcionarem em português. O código 4.7 mostra um exemplo de como usar esse recurso.

```
from textblob import TextBlob
texto = "Brazil is an amazing country!"
blob = TextBlob(texto)
sentimento = blob.sentiment.polarity
if sentimento > 0:
    print("O texto tem uma conotação positiva.")
elif sentimento < 0:
    print("O texto tem uma conotação negativa.")
else:
    print("O texto tem uma conotação neutra.")
# Saída: O texto tem uma conotação positiva.
```

Exemplo de Código 4.10: Análise de sentimento em textos.

4.3 Métodos do Tipo *String*

As *strings* são objetos imutáveis em Python, o que significa que você não pode alterar o conteúdo de uma *string* depois de criada. No entanto, existem muitos métodos disponíveis para manipulação de *strings*, como substituição de texto, conversão de maiúsculas e minúsculas (vistos anteriormente), formatação de *strings* (visto anteriormente) e extração de *substrings*. Nesta seção, veremos alguns dos métodos complementares do tipo *string* que podem ser úteis em tarefas de manipulação de textos.

4.3.1 Substituição de Texto

O método `replace()` é usado para substituir todas as ocorrências de uma *substring* por outra em uma *string* existente. O código 4.11 mostra um exemplo de como usar esse método.

```
texto = "Python é uma linguagem de programação popular. Python é fácil de aprender e usar."
novo_texto = texto.replace("Python", "Java")
print(novo_texto)
#Saída: Java é uma linguagem de programação popular. Java é fácil de aprender e usar.
```

Exemplo de Código 4.11: Substituição de *string* por outra *string*.

4.3.2 Extração de *Substrings*

O método `split()` é usado para dividir uma *string* em *substrings* com base em um separador especificado. Por padrão, o separador é um espaço em branco. O código 4.12 mostra um exemplo de como usar esse método.

```
texto = "Este é um exemplo de texto."
palavras = texto.split()
print(palavras) # Saída: ['Este', 'é', 'um', 'exemplo', 'de', 'texto.']
```

Exemplo de Código 4.12: Divisão de uma *string* em palavras.

O método `slice()` é usado para extrair uma parte de uma *string*. O índice inicial e final da parte desejada são especificados como argumentos. O código 4.13 mostra um exemplo de como usar esse método.

```
texto = "Este é um exemplo de texto."
parte = texto[5:18]
print(parte) # Saída: é um exemplo
```

Exemplo de Código 4.13: Extração de parte de uma *string*.

4.3.3 Verificação de *Strings*

Os métodos `startswith()` e `endswith()` são usados para verificar se uma *string* começa ou termina com uma determinada *substring*, respectivamente. O código 4.14 mostra um exemplo de como usar esse método.

```
texto = "Este é um exemplo de texto."
if texto.startswith("Este"):
    print("A string começa com 'Este'.")
else:
    print("A string não começa com 'Este'.")
if texto.endswith("texto."):
    print("A string termina com 'texto.'")
else:
    print("A string não termina com 'texto.'")
# Saída:
# A string começa com 'Este'
# A string termina com 'texto.'
```

Exemplo de Código 4.14: Verificação se *string* começa ou termina com outra *string*.

4.3.4 Remoção de Espaços em Branco

O método `strip()` é usado para remover espaços em branco no início e no final de uma *string*. O código 4.15 mostra um exemplo de como usar esse método.

```
texto = "     Este é um exemplo de texto.     "
texto_sem_espaco = texto.strip()
print(texto_sem_espaco) # Saída: Este é um exemplo de texto.
```

Exemplo de Código 4.15: Remoção de espaços em branco de uma *string*.

4.3.5 Localização de Substrings

O método `find()` é usado para encontrar a posição de uma *substring* em uma *string*. Se a *substring* não for encontrada, o método retorna -1. O código 4.16 mostra um exemplo de como usar esse método.

```
texto = "Este é um exemplo de texto."
posicao = texto.find("exemplo")
print(posicao) # Saída: 10
```

Exemplo de Código 4.16: Localização de uma *substring* em uma *string*.

4.3.6 Contagem de *Substrings*

O método `count()` é usado para contar o número de ocorrências de uma *substring* em uma *string*. O código 4.17 mostra um exemplo de como usar esse método.

```
texto = "Este é um exemplo de texto. Este texto é sobre Python."
num_ocorrencias = texto.count("Este")
print(num_ocorrencias) # Saída: 2
```

Exemplo de Código 4.17: Contagem de ocorrências de uma *substring* dentro de uma *string*.

4.4 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 4.1 Crie um programa que peça ao usuário para inserir dois números e calcule a potência do primeiro número pelo segundo número. ■

Exercício 4.2 Crie um programa que peça ao usuário para inserir um número e calcule a raiz quadrada desse número. ■

Exercício 4.3 Crie um programa que peça ao usuário para inserir um ângulo em graus e calcule o seno, cosseno e tangente desse ângulo. ■

Exercício 4.4 Crie um programa que peça ao usuário para inserir um número e calcule o logaritmo natural desse número. ■

Exercício 4.5 Crie um programa que peça ao usuário para inserir um número e calcule o fatorial desse número. ■

Exercício 4.6 Crie um programa que peça ao usuário para inserir dois números complexos e calcule a soma e o produto desses números. ■

Exercício 4.7 Crie um programa que peça ao usuário para digitar um nome de usuário e uma senha contendo apenas caracteres alfanuméricos e use expressão regular para fazer uma limpeza nos valores digitados, exibindo-os novamente para o usuário os valores que forem modificados. ■

Exercício 4.8 Crie um programa que peça ao usuário para digitar uma frase com 5 palavras. Caso a frase digitada tenha uma quantidade diferente de palavras, o usuário deve digitar novamente. Ao fim, mostre uma palavra por linha. Use tokenização para extrair as palavras. ■

Exercício 4.9 Crie um programa que peça ao usuário para digitar uma frase, uma palavra presente na frase e outra palavra ausente na frase. Em seguida, substitua todas as ocorrências da palavra existente pela palavra inexistente. ■

Exercício 4.10 Crie um programa que peça ao usuário para digitar uma frase e que, em seguida, extraia todos os artigos da frase digitada e mostre-a sem os artigos. ■

Exercício 4.11 Crie um programa que peça ao usuário para digitar uma frase e que, em seguida, fatie a frase em *substrings* de 6 caracteres e mostre-as uma por linha. ■

Exercício 4.12 Crie um programa que peça ao usuário para digitar uma frase e que, em seguida, verifique quantas palavras terminam com a letra "o" e quantas terminam com a letra "a". ■

Exercício 4.13 Crie um programa que peça ao usuário para digitar uma frase, divida-a em palavras, remova todos os espaços em branco desnecessários dessas palavras, e componha a frase novamente com apenas 1 espaço entre as palavras. ■

Exercício 4.14 Crie um programa que peça ao usuário para digitar uma frase e que, em seguida, mostre a posição inicial de cada palavra contida nessa frase. ■

Exercício 4.15 Crie um programa que peça ao usuário para digitar uma frase e que, em seguida, mostre quantas vezes cada palavra aparece nessa frase. ■

4.5 Considerações Sobre o Capítulo

Este capítulo apresentou formas mais avançadas de se manipular números e textos usando a linguagem Python. Foram mostradas diversas operações matemáticas, incluindo potenciação, radiciação, operações trigonométricas, operações com números complexos, entre outras. Ainda, foram mostradas formas avançadas de manipulação de textos e, ao fim, uma série de exercícios foram propostos. No próximo capítulo, você verá como manipular coleções de forma mais avançada.

5. Coleções

A linguagem Python oferece diversas formas de manipulação de suas coleções, a saber, listas, tuplas, dicionários e conjuntos. O domínio dessas técnicas pode lhe tornar um programador diferenciado, uma vez que a manipulação correta de coleções é uma tarefa frequentemente necessária em programas que lidam com dados. Neste capítulo, são apresentadas algumas técnicas avançadas para manipulação de coleções em Python.

5.1 *List Comprehension*

A *List Comprehension* é uma construção sintática da linguagem Python que permite criar uma nova lista a partir de uma ou mais listas existentes de forma concisa e elegante. Ela é muito usada em Python e é uma das características mais distintas da linguagem. Basicamente, a técnica consiste em aplicar uma expressão a cada elemento de uma lista existente e filtrar apenas os elementos que satisfazem uma determinada condição. O código 5.1 mostra como usar esse recurso.

```
1 numeros = [1, 2, 3, 4, 5]
2 quadrados = [x**2 for x in numeros if x % 2 == 0]
3 print(quadrados)  # Saída: [4, 16]
```

Exemplo de Código 5.1: Aplicando *list comprehension* a coleções.

Nesse exemplo, a lista `quadrados` é criada a partir da lista `numeros`, elevando ao quadrado apenas os números pares.

5.2 Mapeamento e Filtragem

O `map()` e `filter()` são duas funções nativas do Python que podem ser usadas para manipular coleções de forma mais eficiente. A função `map` aplica uma função a cada elemento de uma coleção, retornando um iterador com os resultados. Já a função `filter` retorna apenas os elementos que satisfazem uma determinada condição. O código 5.2 mostra como usar esse recurso.

```
numeros = [1, 2, 3, 4, 5]
quadrados = map(lambda x: x**2, numeros)
pares = filter(lambda x: x % 2 == 0, numeros)
print(list(quadrados))  # Saída: [1, 4, 9, 16, 25]
print(list(pares))  # Saída: [2, 4]
```

Exemplo de Código 5.2: Aplicando `map` e `filter` a coleções.

Nesse exemplo, a função `map()` é usada para criar uma lista com o quadrado de cada elemento da lista `numeros`. Já a função `filter()` é usada para retornar apenas os elementos pares da lista `numeros`.

5.3 Zip

A função `zip()` é usada para combinar duas ou mais coleções em uma única coleção. Ela retorna um iterador com tuplas contendo os elementos correspondentes de cada coleção. O código 5.3 mostra como usar esse recurso.

```
nomes = ["Alice", "Bob", "Carol"]
idades = [25, 30, 35]
pessoas = zip(nomes, idades)
for nome, idade in pessoas:
    print(f"{nome} tem {idade} anos")
```

Exemplo de Código 5.3: Aplicando `zip` a coleções,

Nesse exemplo, a função `zip()` é usada para combinar a lista `nomes` e `idades` em uma única lista de tuplas. O laço `for` então é usado para iterar sobre a lista de tuplas e exibir os resultados.

5.4 *Dict Comprehension*

Assim como a *List Comprehension*, também é possível criar dicionários utilizando a sintaxe da *Dict Comprehension*. Nesse caso, é necessário definir a chave e o valor de cada item a partir

de uma expressão aplicada a cada elemento da coleção. O código 5.4 mostra como usar esse recurso.

```
numeros = [1, 2, 3, 4, 5]
quadrados = {x: x**2 for x in numeros}
print(quadrados)  # Saída: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Exemplo de Código 5.4: Aplicando *dict comprehension* a coleções

Nesse exemplo, um dicionário é criado a partir da lista `numeros`, utilizando cada elemento como chave e o seu quadrado como valor correspondente.

5.5 Conjuntos

Os conjuntos (*sets*) em Python, já apresentados no início deste material, são coleções não ordenadas de elementos únicos. Eles podem ser usados para remover elementos duplicados de uma lista ou para realizar operações de conjuntos, como união, interseção e diferença. O código 5.5 mostra como usar esse recurso.

```
numeros1 = [1, 2, 3, 4, 5]
numeros2 = [3, 4, 5, 6, 7]
conjunto1 = set(numeros1)
conjunto2 = set(numeros2)
uniao = conjunto1.union(conjunto2)
intersecao = conjunto1.intersection(conjunto2)
diferenca = conjunto1.difference(conjunto2)
print(uniao)  # Saída: {1, 2, 3, 4, 5, 6, 7}
print(intersecao)  # Saída: {3, 4, 5}
print(diferenca)  # Saída: {1, 2}
```

Exemplo de Código 5.5: Operações sobre coleções do tipo conjuntos

Nesse exemplo, os conjuntos `conjunto1` e `conjunto2` são criados a partir das listas `numeros1` e `numeros2`. As operações de união, interseção e diferença são realizadas entre os dois conjuntos.

5.6 Indexação

Em Python, a indexação é uma operação que permite acessar elementos individuais ou intervalos de uma coleção. As coleções em Python incluem listas, tuplas, strings, dicionários e conjuntos. Existem várias opções de indexação que podem ser usadas para acessar elementos em diferentes posições ou intervalos dentro de uma coleção. Essas opções são apresentadas

em detalhes a seguir.

5.6.1 Acesso a Um Elemento Único

Para acessar um elemento único em uma coleção, é necessário usar o índice numérico correspondente ao elemento. O índice numérico começa em 0 e vai até o comprimento da coleção menos 1. O código 5.6 mostra como usar esse recurso.

```
# Acessando um elemento único em uma lista
lista = [1, 2, 3, 4, 5]
primeiro_elemento = lista[0]  # Saída: 1
terceiro_elemento = lista[2]  # Saída: 3

# Acessando um caractere único em uma string
frase = "Hello, world!"
primeiro_caractere = frase[0]  # Saída: H
ultimo_caractere = frase[-1]  # Saída: !
```

Exemplo de Código 5.6: Acessando um único elemento de uma coleção.

5.6.2 Acessando um Intervalo de Elementos

Para acessar um intervalo de elementos dentro de uma coleção, é necessário usar a sintaxe de fatiamento (*slicing*). O fatiamento é feito usando dois pontos (:) entre os índices de início e fim, sendo que o índice de fim é exclusivo. O código 5.7 mostra como usar esse recurso.

```
# Acessando um intervalo de elementos em uma lista
lista = [1, 2, 3, 4, 5]
primeiros_tres_elementos = lista[0:3]  # Saída: 1, 2, 3
elementos_do_meio = lista[1:4]  # Saída: 2, 3, 4

# Acessando um intervalo de caracteres em uma string
frase = "Hello, world!"
primeiros_caracteres = frase[0:5]  # Saída: Hello
caracteres_do_meio = frase[7:12]  # Saída: world
```

Exemplo de Código 5.7: Acessando um intervalo de elementos de uma coleção.

5.6.3 Acesso a Elementos a Partir do Final da Coleção

Para acessar elementos a partir do final de uma coleção, é necessário usar índices negativos. O índice negativo -1 representa o último elemento, -2 representa o penúltimo elemento, e assim por diante. O código 5.8 mostra como usar esse recurso.

```python
# Acessando elementos a partir do final de uma lista
lista = [1, 2, 3, 4, 5]
ultimo_elemento = lista[-1]  # Saída: 5
penultimo_elemento = lista[-2]  # Saída: 4

# Acessando caracteres a partir do final de uma string
frase = "Hello, world!"
ultimo_caractere = frase[-1]  # Saída: !
penultimo_caractere = frase[-2]  # Saída: d
```

Exemplo de Código 5.8: Acessando elementos a partir do final da coleção.

5.6.4 Acessando Elementos Usando Passo

Para acessar elementos de uma coleção com um passo (intervalo entre os elementos), é necessário usar a sintaxe de fatiamento (*slicing*) com um terceiro parâmetro que indica o tamanho do passo. O código 5.9 mostra como usar esse recurso.

```python
# Acessando elementos de uma lista com passo
lista = [1, 2, 3, 4, 5]
elementos_com_passo = lista[0:5:2]  # Saída: 1, 3

# Acessando caracteres de uma string com passo
frase = "Hello, world!"
caracteres_com_passo = frase[0:12:2]  # Saída: Hlo ol
```

Exemplo de Código 5.9: Acessando intervalos de elementos usando passo.

5.6.5 Acessando Elementos de Forma Invertida

Para acessar os elementos de uma coleção de forma invertida, é necessário usar a sintaxe de fatiamento (*slicing*) com um passo negativo. O passo negativo inverte a ordem dos elementos. O código 5.10 mostra como usar esse recurso.

```python
# Acessando os elementos de uma lista de forma reversa
lista = [1, 2, 3, 4, 5]
elementos_reversos = lista[::-1]
# Saída: 5, 4, 3, 2, 1

# Acessando os caracteres de uma string de forma reversa
frase = "Hello, world!"
caracteres_reversos = frase[::-1]
# Saída: !dlrow ,olleH
```

Exemplo de Código 5.10: Invertendo uma coleção

Essas são as opções de indexação de coleções em Python. É importante lembrar que a indexação começa em 0 e que o índice de fim é exclusivo, ou seja, é igual ao valor menos 1. Além disso, as coleções em Python são objetos iteráveis que podem ser percorridos com laços `for` e outras técnicas de iteração. A escolha da opção de indexação adequada dependerá do contexto e da situação em que a coleção está sendo usada.

5.7 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 5.1 Use *list comprehension* para criar uma lista com as raízes quadradas dos números pares de 0 a 20. ■

Exercício 5.2 Use *list comprehension* para criar uma lista com as palavras que contêm a letra “a” em uma frase digitada pelo usuário, substituindo a letra por “o”. ■

Exercício 5.3 Use *list comprehension* para criar uma lista com o quadrado dos números pares entre 0 e 10. ■

Exercício 5.4 Use *list comprehension* para criar uma lista com os números divisíveis por 3 ou 5 de 0 a 30. ■

Exercício 5.5 Crie um dicionário com nomes e notas de alunos digitados pelo usuário, usando os nomes dos alunos como chave e as notas como valor. Em seguida, use *dict comprehension* para criar um dicionário com os alunos com nota igual ou superior a 7. ■

Exercício 5.6 Use *dict comprehension* para criar um dicionário com as raízes quadradas dos números de 1 a 10. Utilize os números como chave e as raízes quadradas como valor. ■

Exercício 5.7 Crie um dicionário com nomes e notas de alunos digitados pelo usuário, usando os nomes dos alunos como chave e as notas como valor. Em seguida, use *dict comprehension* para criar um dicionário com os alunos e suas notas arrendadas para o número inteiro mais próximo. ■

Exercício 5.8 Faça um programa que peça ao usuário para digitar uma lista de números e que, em seguida, retorne uma nova lista com os quadrados de cada número. Use mapeamento. ■

Exercício 5.9 Faça um programa que peça ao usuário para digitar uma lista de nomes e que, em seguida, retorne uma nova lista com os nomes em caixa alta. Use mapeamento. ■

Exercício 5.10 Faça um programa que peça ao usuário para digitar uma lista de palavras e que, em seguida, retorne uma nova lista com os comprimentos de cada palavra. ■

Exercício 5.11 Faça um programa que peça ao usuário para digitar uma lista de números e que, em seguida, retorna uma nova lista apenas com os números pares. ■

Exercício 5.12 Escreva um programa que leia duas listas, uma com as chaves e outra com os valores, e retorna um dicionário. ■

Exercício 5.13 Escreva um programa que leia duas listas de mesmo tamanho e retorna uma nova lista com a média dos elementos correspondentes. ■

Exercício 5.14 Escreva um programa que leia duas listas de mesmo tamanho, uma com os nomes dos alunos e outra com as notas, e retorna uma lista com as tuplas (nome, nota) em ordem decrescente de nota. ■

Exercício 5.15 Escreva um programa que leia uma lista de números e um número de referência e retorna a posição do primeiro elemento maior que o número de referência. ■

Exercício 5.16 Escreva um programa que leia uma lista de números e inverta a ordem dos elementos de índices pares. ■

5.8 Considerações Sobre o Capítulo

Este capítulo apresentou formas mais avançadas de se manipular coleções usando a linguagem Python. Foram mostrados recursos para combinar coleções de várias formas, para mapear e filtrar coleções, realizar operações em conjuntos e indexar coleções visando acessar um elemento único ou um intervalo de elementos. Ao fim, uma série de exercícios foram propostos. No próximo capítulo, você aprenderá a criar e usar funções.

6. Funções

Funções em Python são uma das principais ferramentas para organizar e reutilizar código. Uma função é um bloco de código que pode ser chamado várias vezes, em diferentes partes do programa, para realizar uma tarefa específica.

As funções são úteis para dividir um programa em partes menores e mais gerenciáveis. Em vez de escrever todo o código em um único bloco, podemos criar funções para realizar tarefas específicas e chamar essas funções sempre que necessário. Isso torna o código mais legível e fácil de manter, já que cada função tem um objetivo claro e pode ser compreendida e testada separadamente.

Além disso, as funções permitem a reutilização de código. Se houver uma tarefa que precisa ser realizada várias vezes em um programa, é possível criar uma função para essa tarefa e chamá-la sempre que necessário. Dessa forma, podemos economizar tempo e evitar a repetição desnecessária de código.

As funções em Python podem se tornar ainda mais flexíveis quando são criadas para receber argumentos, que são valores de entrada que a função usa para realizar sua tarefa. Esses argumentos podem ser opcionais ou obrigatórios e podem ter um valor padrão. Além disso, as funções podem retornar um único valor ou uma lista de valores, que podem ser usados em outras partes do programa.

Por fim, as funções também têm algumas desvantagens. Por exemplo, podem ser difíceis de entender e depurar em programas grandes e complexos. Além disso, o uso excessivo de funções pode levar a uma complexidade desnecessária e tornar o código mais difícil de entender. As seções seguintes mostram como usar funções de forma eficiente em Python.

6.1 Como Definir Funções

Para criar uma função simples em Python, usamos a palavra-chave `def`, seguida pelo nome da função e pelos argumentos, se houver. Em seguida, definimos as instruções que a função deve executar e, opcionalmente, o valor que a função deve retornar quando terminar de executar. O código 6.1 mostra um exemplo simples de uma função em Python.

```
def saudacao(nome):
    print("Olá, " + nome + "!")

saudacao("João")  # Saída: "Olá, João!"
saudacao("Maria")  # Saída: "Olá, Maria!"
```

Exemplo de Código 6.1: Criando e chamando uma função simples.

Neste exemplo, a função `saudacao` recebe um argumento `nome` e imprime uma saudação personalizada. Depois, a função é chamada duas vezes, uma com o argumento `João` e outra com o argumento `Maria`. O resultado é a exibição da mensagem "Olá, João!" seguida pela mensagem "Olá, Maria!".

6.2 Argumentos de Função

Argumentos de função são valores que uma função recebe quando é chamada. Os argumentos permitem que a função seja mais flexível e adaptável a diferentes situações e entradas de dados. Em Python, existem três tipos de argumentos de função: argumentos posicionais, argumentos nomeados e argumentos padrão. Vamos ver cada um deles mais detalhadamente nas subseções a seguir.

6.2.1 Argumentos Posicionais

Os argumentos posicionais são os valores que são passados na chamada da função na ordem em que são definidos na criação da função. O código 6.2 mostra um exemplo de uso.

```
def soma(a, b):
    return a + b

resultado = soma(3, 5)
print(resultado)  # Saída: 8
```

Exemplo de Código 6.2: Criando e chamando uma função com argumentos posicionais.

Neste exemplo, `a` e `b` são argumentos posicionais. Quando a função `soma` é chamada com os

argumentos 3 e 5, o valor 3 é atribuído à variável `a` e o valor 5 é atribuído à variável `b`. Em seguida, a função retorna a soma desses dois valores, que é 8.

6.2.2 Argumentos Nomeados

Os argumentos nomeados permitem que os argumentos sejam passados para a função em qualquer ordem, desde que sejam especificados pelo nome do argumento. O código 6.3 mostra um exemplo de uso.

```
def saudacao(nome, sobrenome):
    print("Olá, " + nome + " " + sobrenome + "!")

saudacao(sobrenome="Silva", nome="João")  # Saída: "Olá, João Silva!"
```

Exemplo de Código 6.3: Criando e chamando uma função com argumentos posicionais.

Neste exemplo, os argumentos `nome` e `sobrenome` são nomeados. Quando a função `saudacao` é chamada com os argumentos `sobrenome` e `nome` trocados de posição, o Python ainda sabe como atribuir os valores corretos às variáveis, porque cada argumento é nomeado explicitamente.

6.2.3 Argumentos com Valor Padrão

Os argumentos com valor padrão são argumentos que possuem valores predefinidos na definição da função, sendo que esse valores são atribuídos aos argumentos no momento da chamada da função caso nenhum valor seja passado. O código 6.4 mostra um exemplo de uso.

```
def potencia(base, expoente=2):
    return base ** expoente

resultado = potencia(3)
print(resultado)  # Saída: 9
```

Exemplo de Código 6.4: Criando e chamando uma função que possui argumentos com valor padrão.

Neste exemplo, o argumento `expoente` tem um valor padrão de 2. Quando a função `potencia` é chamada com apenas um argumento (3), o Python atribui automaticamente o valor padrão de 2 ao argumento `expoente`. Em seguida, a função retorna o resultado da potência de 3 elevado a 2, que é 9. Também é possível especificar um valor diferente para o argumento `expoente`, quando necessário. O código 6.5 mostra um exemplo de uso.

Neste exemplo, o segundo argumento é especificado como 3, substituindo o valor padrão de 2.

```
resultado = potencia(3, 3)
print(resultado)  # Saída: 27
```

Exemplo de Código 6.5: Sobrescrevendo valores de argumentos com valor padrão.

Em seguida, a função retorna o resultado da potência de 3 elevado a 3, que é 27.

6.3 Combinando Argumentos

Os argumentos de função também podem ser combinados, ou seja, é possível usar argumentos posicionais e argumentos com valor padrão na mesma função, e ainda é possível chamar essa função usando-se argumentos nomeados. O código 6.6 mostra um exemplo de uso desse recurso.

```
def calcular_imc(peso, altura, unidade='kg/m2'):
    if unidade == 'kg/m2':
        return peso / altura ** 2
    elif unidade == 'lb/in2':
        return 703 * peso / altura ** 2
    else:
        return None

resultado1 = calcular_imc(70, 1.75)
resultado2 = calcular_imc(154, 69, unidade='lb/in2')

print(resultado1)  # Saída: 22.86
print(resultado2)  # Saída: 22.78
```

Exemplo de Código 6.6: Combinando diferentes tipos de argumento.

Neste exemplo, a função `calcular_imc` recebe três argumentos: `peso`, `altura` e `unidade`, que tem uma unidade padrão igual a "kg/m2". Se a unidade for especificada como "kg/m2", a função retorna o índice de massa corporal (IMC) calculado a partir do peso e altura. Se a unidade for especificada como "lb/in2", a função retorna o IMC calculado a partir do peso e altura convertidos para as unidades americanas. No exemplo, a função é chamada duas vezes, uma vez sem especificar a unidade (usando o valor padrão) e outra especificando a unidade como "lb/in2". Em seguida, os resultados são impressos na tela.

Esses são os três tipos de argumentos de função em Python. Ao usar argumentos posicionais, nomeados e padrão, podemos criar funções mais flexíveis e adaptáveis, que podem ser usadas em diferentes situações e com diferentes entradas de dados.

6.4 Escopo de Variáveis em Funções

O escopo de uma variável é a parte do programa em que a variável pode ser acessada e usada. Em Python, o escopo de uma variável pode ser local ou global, dependendo de onde a variável é definida. Quando uma variável é definida dentro de uma função, ela é considerada uma variável local e seu escopo é limitado ao corpo da função. Isso significa que a variável só pode ser acessada e usada dentro da função em que foi definida. O código 6.7 mostra um exemplo de uso desse recurso.

```
1 def soma(a, b):
2     resultado = a + b
3     return resultado
4
5 print(soma(2, 3))  # Saída: 5
6 print(resultado)   # Saída: NameError: name 'resultado' is not defined
```

Exemplo de Código 6.7: Erro em escopo de variáveis em funções.

Neste exemplo, a variável `resultado` é definida dentro da função `soma` e seu escopo é limitado ao corpo da função. Quando a função é chamada com os argumentos 2 e 3, a variável `resultado` é criada dentro da função, recebe o valor da soma de `a` e `b` e é retornada pela função. Fora da função, a variável `resultado` não existe e, se for tentado acessá-la, o Python gerará um erro `NameError`.

Por outro lado, quando uma variável é definida fora de uma função, ela é considerada uma variável global e pode ser acessada e usada em qualquer parte do programa. O código 6.8 mostra um exemplo de uso desse recurso.

```
1 resultado = 0
2
3 def soma(a, b):
4     global resultado
5     resultado = a + b
6
7 soma(2, 3)
8 print(resultado)  # Saída: 5
```

Exemplo de Código 6.8: Variável global usada em função.

Neste exemplo, a variável `resultado` é definida fora da função `soma` e seu escopo é global. Dentro da função, é necessário usar a palavra-chave `global` para informar ao Python que a

variável `resultado` é global e deve ser acessada fora do escopo da função. Quando a função é chamada com os argumentos 2 e 3, a variável `resultado` é atualizada com o valor da soma de `a` e `b`. Fora da função, a variável `resultado` é impressa na tela com o valor 5.

É importante observar que, embora variáveis globais possam ser convenientes em alguns casos, é uma boa prática evitar o uso excessivo de variáveis globais, pois elas podem tornar o código menos legível e mais difícil de depurar.

Em resumo, o escopo de uma variável em Python pode ser local ou global, dependendo de onde a variável é definida. O escopo local é limitado ao corpo da função em que a variável é definida, enquanto o escopo global permite que a variável seja acessada e usada em qualquer parte do programa. O uso de variáveis globais deve ser evitado sempre que possível para tornar o código mais legível e fácil de depurar.

6.5 Retorno de Valores em Funções

Uma função em Python pode retornar um valor ou uma coleção de valores para o código que a chamou. Isso é útil porque permite que a função processe dados e os envie de volta para o programa principal para serem usados em outras operações. Para retornar um valor em uma função em Python, usamos a palavra-chave `return`. A instrução `return` indica ao Python que a função deve sair e retornar o valor especificado para o código que a chamou. O código 6.9 mostra um exemplo de função que retorna um valor simples.

```
1 def soma(a, b):
2     resultado = a + b
3     return resultado
4
5 print(soma(2, 3))  # Saída: 5
```

Exemplo de Código 6.9: Função com retorno simples.

Neste exemplo, a função `soma` recebe dois argumentos (a e b) e atribui a soma desses valores à variável `resultado`. Em seguida, a função usa a instrução `return` para enviar o valor de `resultado` de volta para o código que a chamou. Fora da função, o valor retornado é impresso na tela com o valor 5. Uma função também pode retornar uma coleção de valores, como uma lista, tupla ou dicionário. O código 6.10 mostra um exemplo de função que retorna dois valores.

Neste exemplo, a função `maior_e_menor` recebe uma lista de números como argumento e usa as funções `max` e `min` do Python para encontrar o maior e o menor valor na lista. Em seguida, a

```python
def maior_e_menor(numeros):
    maior = max(numeros)
    menor = min(numeros)
    return maior, menor

resultado = maior_e_menor([3, 1, 5, 2, 4])
print(resultado)  # Saída: (5, 1)
```

Exemplo de Código 6.10: Função com retorno composto.

função usa a instrução `return` para enviar uma tupla contendo os valores de `maior` e `menor` de volta para o código que a chamou. Fora da função, a tupla é armazenada na variável `resultado` e impressa na tela com o valor (5, 1). Por fim, é importante lembrar que uma função pode ter vários valores de retorno. Para fazer isso, basta separar os valores com vírgulas na instrução `return`. O código 6.11 mostra um exemplo de função que retorna múltiplos valores.

```python
def operacoes(a, b):
    soma = a + b
    subtracao = a - b
    multiplicacao = a * b
    divisao = a / b
    return soma, subtracao, multiplicacao, divisao

resultado = operacoes(6, 2)
print(resultado)  # Output: (8, 4, 12, 3.0)
```

Exemplo de Código 6.11: Função com múltiplos retornos.

Neste exemplo, a função `operacoes` recebe dois argumentos (a e b) e realiza quatro operações matemáticas com esses valores. Em seguida, a função usa a instrução `return` para enviar uma tupla contendo os valores de `soma`, `subtracao`, `multiplicacao` e `divisao` de volta para o código que a chamou. Fora da função, a tupla é armazenada na variável `resultado` e impressa na tela com o valor (8, 4, 12, 3.0).

6.6 Funções Recursivas

Uma função recursiva é uma função que chama a si mesma durante sua execução. Em outras palavras, no corpo da função há uma chamada para a própria função. Ao atingir a **condição de parada**, a função retorna o resultado de volta para o programa principal. A cada nova chamada, novos valores são passados como argumentos, sendo que a pilha de chamadas termina quando a condição de parada, geralmente posicionada no início do corpo da função, é satisfeita.

A recursividade pode ser útil para resolver problemas que podem ser divididos em problemas menores e iguais. Por exemplo, calcular o fatorial de um número pode ser resolvido com uma função recursiva. Para calcular o fatorial de um número *n*, basta multiplicá-lo pelo fatorial de *n-1*. Se *n* for igual a 1, o fatorial de *n* é 1. O código 6.12 mostra um exemplo de função recursiva para calcular o fatorial de um número.

```
def fatorial(n):
    if n == 1:
        return 1
    else:
        return n * fatorial(n-1)
resultado = fatorial(5)
print(resultado)  # Saída: 120
```

Exemplo de Código 6.12: Função com múltiplos retornos.

Neste exemplo, a função `fatorial` recebe um número *n* como argumento. Se *n* for igual a 1, a função retorna 1. Caso contrário, a função chama a si mesma, passando o valor de *n-1* como argumento, e multiplica o resultado pelo valor de *n*. O processo continua até que *n* seja igual a 1 e, em seguida, a função retorna o valor final para o código que a chamou.

Em síntese, as funções recursivas são úteis para resolver problemas que podem ser divididos em problemas menores e iguais. Elas podem ser mais eficientes e fáceis de entender em alguns casos, mas também podem consumir muita memória e serem mais difíceis de depurar, portanto, seu uso depende do problema e das limitações de desempenho e memória a considerar.

6.7 Funções Lambda

Funções lambda, também conhecidas como funções anônimas, são funções pequenas e temporárias que são criadas em tempo de execução. Elas são úteis para realizar operações simples e rápidas em uma única linha de código. A sintaxe para definir uma função lambda é mostrada no exemplo 6.13.

```
lambda argumentos: expressão
```

Exemplo de Código 6.13: Sintexe básica de uma função lambda.

Aqui, `argumentos` é uma lista de argumentos separada por vírgulas que a função recebe e `expressão` é a operação que a função executa. Por exemplo, a função lambda mostrada no código 6.14 calcula o dobro de um número.

```
dobro = lambda x: x * 2
resultado = dobro(5)
print(resultado)  # Saída: 10
```

Exemplo de Código 6.14: Função lambda que calcula o dobro de um número.

Neste exemplo, a função lambda `dobro` recebe um argumento `x` e retorna o dobro do valor de `x`. A função é atribuída à variável `dobro` e é chamada com o argumento 5. O resultado é o valor 10, que é impresso na tela. As funções lambda também podem receber vários argumentos e realizar operações mais complexas. Por exemplo, a função lambda do código 6.15 calcula a média ponderada de duas notas.

```
media_ponderada = lambda nota1, peso1, nota2, peso2:
    (nota1 * peso1 + nota2 * peso2) / (peso1 + peso2)
resultado = media_ponderada(8, 3, 9, 2)
print(resultado)  # Saída: 8.3
```

Exemplo de Código 6.15: Função lambda para calcular média ponderada de duas notas.

Neste exemplo, a função lambda `media_ponderada` recebe quatro argumentos: `nota1`, `peso1`, `nota2` e `peso2`. A função retorna a média ponderada das duas notas, usando as fórmulas apropriadas. A função é atribuída à variável `media_ponderada` e é chamada com os argumentos 8, 3, 9 e 2. O resultado é o valor 8.3, que é impresso na tela.

As funções lambda também podem ser usadas em conjunto com funções como `map`, `filter` e `reduce` do Python para processar sequências de dados de forma mais concisa. O código 6.16 mostra um exemplo em que a função lambda e a função `filter` do Python filtram os números pares em uma lista.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Saída: [2, 4, 6, 8, 10]
```

Exemplo de Código 6.16: Função lambda usada com a função `filter` do Python.

Neste exemplo, a função lambda é usada como argumento da função `filter`. A função lambda testa se um número é par e retorna `True` ou `False` de acordo com o resultado. A função `filter` filtra a lista `numeros` com base na condição especificada pela função lambda e retorna uma nova lista com apenas os números pares. A nova lista é armazenada na variável `pares` e impressa na tela.

Em resumo, as funções lambda em Python são úteis para funções pequenas e temporárias que podem ser criadas em tempo de execução. Elas são uma maneira rápida e fácil de definir funções em uma única linha de código, sem a necessidade de criar uma função completa com nome e argumentos. Além disso, as funções lambda podem ser usadas em conjunto com outras funções do Python, como `map`, `filter` e `reduce`, para processar sequências de dados de forma mais concisa e eficiente. Saber usar funções lambda pode ser muito útil para escrever código mais limpo e eficiente em Python.

6.8 Funções de Ordem Superior

Funções de ordem superior são funções que recebem outras funções como argumentos e/ou retornam funções como resultado. Essas funções são úteis para criar funções genéricas que podem ser usadas com diferentes tipos de dados e operações.

Um exemplo simples de função de ordem superior é a função `map` do Python. A função `map` recebe uma função e uma sequência de dados como argumentos e aplica a função a cada elemento da sequência, retornando uma nova sequência com os resultados. No código do exemplo 6.17, função `map` aplica a função `dobro` a cada elemento da lista `numeros` e uma nova lista é criada.

```
1  def dobro(x):
2      return x * 2
3
4  numeros = [1, 2, 3, 4, 5]
5  resultado = list(map(dobro, numeros))
6  print(resultado)  # Saída: [2, 4, 6, 8, 10]
```

Exemplo de Código 6.17: Passagem de função como argumento de outra função.

Neste exemplo 6.17, a função `dobro` é definida para multiplicar um número por 2. A lista `numeros` contém os números de 1 a 5. A função `map` é usada para aplicar a função `dobro` a cada elemento da lista `numeros` e retornar uma nova lista com os resultados. O resultado é uma lista com os números dobrados de 2 a 10.

Outro exemplo de função de ordem superior é a função `filter` do Python. A função `filter` recebe uma função e uma sequência de dados como argumentos e retorna uma nova sequência com os elementos da sequência original que satisfazem a função. No exemplo de código 6.18, a função `filter` filtra os números pares em uma lista.

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares)  # Saída: [2, 4, 6, 8, 10]
```

Exemplo de Código 6.18: Função `filter` recebendo uma função lambda como argumento.

Neste exemplo, a função lambda é usada como argumento da função `filter`. A função lambda testa se um número é par e retorna `True` ou `False` de acordo com o resultado. A função `filter` filtra a lista `numeros` com base na condição especificada pela função lambda e retorna uma nova lista apenas com os números pares. A nova lista é armazenada na variável `pares` e impressa na tela.

As funções de ordem superior também podem ser usadas para criar funções mais genéricas que podem ser reutilizadas em diferentes partes de um programa. No exemplo de código 6.19, a função `aplicar_operacao` recebe uma função e uma sequência de dados como argumentos e aplica a função a cada elemento da sequência.

```
def aplicar_operacao(operacao, sequencia):
    resultado = []
    for elemento in sequencia:
        resultado.append(operacao(elemento))
    return resultado

def dobro(x):
    return x * 2

numeros = [1, 2, 3, 4, 5]
resultado = aplicar_operacao(dobro, numeros)
print(resultado)  # Saída: [2, 4, 6, 8, 10]
```

Exemplo de Código 6.19: Função de ordem superior mais genérica.

Neste exemplo, a função `aplicar_operacao` recebe um argumento chamado `operacao`, que é uma função, e uma sequência de dados chamada `sequencia`. A função `aplicar_operacao` cria uma lista vazia chamada `resultado`, percorre cada elemento da sequência e aplica a função `operacao` a cada elemento. O resultado de cada operação é adicionado à lista `resultado`, que é retornada no final. A função `dobro` é definida anteriormente para multiplicar um número por 2. A lista `numeros` contém os números de 1 a 5. A função `aplicar_operacao` é usada para aplicar a função `dobro` a cada elemento da lista `numeros` e retornar uma nova lista com os resultados. O resultado é uma lista com os números dobrados de 2 a 10.

Em resumo, as funções de ordem superior em Python são úteis para criar funções genéricas que podem ser usadas com diferentes tipos de dados e operações. Elas permitem que as funções sejam passadas como argumentos e/ou retornadas como resultados, o que facilita a reutilização de código e a criação de funções mais flexíveis e poderosas. Saber usar funções de ordem superior pode ser muito útil para escrever código mais limpo e eficiente em Python.

6.9 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 6.1 Crie uma função que receba dois números como argumentos e retorne a soma dos dois números. ■

Exercício 6.2 Crie uma função que receba um número como argumento e retorne `True` se o número for par e `False` se o número for ímpar. ■

Exercício 6.3 Crie uma função que receba três argumentos posicionais: um número inteiro, um número flutuante e uma *string*. A função deve imprimir os valores dos argumentos na tela. ■

Exercício 6.4 Crie uma função que receba dois argumentos nomeados: `nome` e `idade`. A função deve imprimir na tela uma mensagem a seu gosto contendo o nome e a idade da pessoa. ■

Exercício 6.5 Crie uma função que defina uma variável `x` dentro da função e imprima o valor de `x` na tela. Em seguida, chame a função e verifique se a variável `x` está acessível fora da função. ■

Exercício 6.6 Crie uma função que receba uma lista como argumento e adicione um elemento à lista dentro da função. Em seguida, imprima a lista fora da função para verificar se o elemento foi adicionado corretamente. ■

Exercício 6.7 Crie uma função que receba uma lista como argumento e retorne o maior valor da lista. ■

Exercício 6.8 Crie uma função que receba uma *string* como argumento e retorne a *string* invertida. ■

Exercício 6.9 Crie uma função recursiva que calcule o fatorial de um número inteiro. ■

Exercício 6.10 Crie uma função recursiva que calcule o N-ésimo termo da sequência de Fibonacci. ■

Exercício 6.11 Crie uma função lambda que receba um número como argumento e retorne o quadrado desse número. ■

Exercício 6.12 Crie uma função lambda que receba duas listas como argumentos e retorne uma lista que contenha apenas os elementos que estão nas duas listas. ■

Exercício 6.13 Crie uma função que receba uma lista e uma função como argumentos e retorne uma nova lista com os elementos da lista original que satisfazem a função. ■

Exercício 6.14 Crie uma função que receba duas funções como argumentos e retorne uma nova função que é a composição das duas funções. ■

6.10 Considerações Sobre o Capítulo

Este capítulo mostrou como criar e usar funções em Python. Foram abordados os tipos de argumento que uma função pode receber, o escopo de variáveis em funções, o retorno de valores simples e compostos, funções recursivas, funções lambda e funções de ordem superior. Ao fim, foram propostos diversos exercícios para colocar em prática o que o capítulo abordou. No próximo capítulo, você aprenderá a tratar exceções e a criar exceções personalizadas em Python.

7. Arquivos e Módulos

Python é uma linguagem de programação que oferece muitas funcionalidades para lidar com arquivos e módulos. Arquivos são essenciais para a leitura e gravação de dados em um sistema de arquivos, enquanto módulos são usados para armazenar e reutilizar código em um programa.

Neste capítulo, exploraremos como trabalhar com arquivos e módulos em Python. Você aprenderá como abrir, ler e escrever em arquivos, bem como usar os recursos de gerenciamento de arquivos para manipulá-los em um sistema de arquivos. Além disso, você aprenderá a importar e a utilizar módulos em Python, e também aprenderá a criar seus próprios módulos e pacotes. Ao final deste capítulo, você terá uma compreensão sólida dos conceitos de arquivos e módulos em Python, e estará pronto para usar essas funcionalidades em seus próprios projetos de programação.

7.1 Trabalhando com Arquivos

Uma das principais funcionalidades da linguagem Python é a capacidade de ler e escrever arquivos de forma simples e eficiente. Arquivos são usados para armazenar dados em um sistema de arquivos e podem ser lidos e manipulados por programas. Nesta seção, exploraremos como trabalhar com arquivos em Python, desde a abertura e fechamento de arquivos, até a manipulação de arquivos e o uso da instrução `with`.

7.1.1 Abrindo e Fechando Arquivos

Antes de poder ler ou escrever em um arquivo em Python, é necessário abrir o arquivo. A função `open()` é usada para abrir arquivos em Python e pode receber dois parâmetros: o nome do arquivo e o modo de abertura. O modo de abertura pode ser `r` (leitura), `w` (escrita) ou `a` (acréscimo). O código 7.1 mostra um exemplo de uso desse recurso.

```
arquivo = open('meuarquivo.txt', 'r')
```

Exemplo de Código 7.1: Abrindo um arquivo em modo de leitura.

Neste exemplo 7.1, o arquivo `meuarquivo.txt` é aberto em modo de leitura. Para fechar o arquivo, é preciso chamar a função `close()` a partir da variável `arquivo`, isto é, `arquivo.close`, como veremos em exemplos mais adiante. É importante sempre fechar o arquivo após seu uso para garantir que o arquivo seja salvo corretamente e para liberar recursos do sistema. Existem formas mais adequadas de se fazer isso, conforme veremos mais adiante.

7.1.2 Lendo e Escrevendo Arquivos

Depois de abrir um arquivo em Python, é possível ler ou escrever no arquivo. A função `read()` é usada para ler todo o conteúdo de um arquivo, enquanto a função `write()` é usada para escrever no arquivo. O código 7.2 mostra um exemplo de abertura de um arquivo em modo de escrita.

```
arquivo = open('meuarquivo.txt', 'w')
arquivo.write('Este é o conteúdo do meu arquivo.')
arquivo.close()
```

Exemplo de Código 7.2: Abrindo um arquivo em modo de escrita e escrevendo nele.

Neste exemplo 7.2, o arquivo `meuarquivo.txt` é aberto em modo de escrita, e a *string* "Este é o conteúdo do meu arquivo." é escrita no arquivo. Outras funções também estão disponíveis para escrita e leitura em arquivos. A função `readline()` é usada para ler uma linha de cada vez de um arquivo, enquanto a função `writelines()` é usada para escrever várias linhas em um arquivo. O exemplo 7.3 mostra como usar o método `readline()` para ler um arquivo linha a linha.

```
arquivo = open('arquivo.txt', 'r')
linha = arquivo.readline()
while linha:
    print(linha)
    linha = arquivo.readline()
arquivo.close()
```

Exemplo de Código 7.3: Leitura de um arquivo linha por linha.

No exemplo 7.3, enquanto a leitura de uma linha com `readline()` retornar algo, o programa repetirá a tentativa de leitura da linha seguinte.

7.1.3 Manipulando Arquivos

Além de ler e escrever em um arquivo, é possível usar outras funções nativas do Python úteis para manipulação de arquivos, como `seek()`, que é usada para mover o ponteiro de leitura ou gravação para uma posição específica no arquivo, ou `tell()`, que é usada para obter a posição atual do ponteiro. Também é possível usar funções dos módulos `os` e `shutil` para manipular arquivos em um sistema de arquivos, incluindo a criação, exclusão, renomeação e cópia de arquivos, como veremos adiante.

7.1.4 Utilizando a Instrução `with`

A instrução `with` em Python é uma forma conveniente de abrir um arquivo e garantir que ele seja fechado após o uso. A instrução `with` também ajuda a evitar erros de programação, pois o arquivo é automaticamente fechado no final do bloco `with`. O código 7.4 mostra um exemplo de uso desse recurso.

```
1 with open('meuarquivo.txt', 'r') as arquivo:
2     for linha in arquivo:
3         print(linha)
```

Exemplo de Código 7.4: Abrindo um arquivo usando o bloco `with`

Neste exemplo 7.4, o arquivo `meuarquivo.txt` é aberto em modo de leitura usando a instrução `with`. Em seguida, um bloco `for` percorre o arquivo (`arquivo`) linha a linha e, por fim, a instrução `with` garante que o arquivo seja fechado automaticamente após o uso.

7.2 Gerenciamento de Arquivos

Além de ler e escrever arquivos em Python, é possível manipular arquivos em um sistema de arquivos usando várias funções e módulos integrados do Python. Nesta seção, veremos como trabalhar com diretórios em Python, como renomear e excluir arquivos, como copiar e mover arquivos e como compactar e descompactar arquivos.

7.2.1 Trabalhando com Diretórios

Antes de manipular arquivos em um sistema de arquivos, é importante saber como trabalhar com diretórios em Python. O módulo `os` é usado para trabalhar com diretórios em Python, e inclui várias funções úteis, como `mkdir()`, que é usada para criar um novo diretório, e `rmdir()`, que é usada para excluir um diretório. O código 7.5 mostra um exemplo de uso desse recurso.

```
import os

os.mkdir('meudiretorio')
os.rmdir('meudiretorio')
```

Exemplo de Código 7.5: Criando e removendo um diretório.

Neste exemplo 7.5, o diretório `meudiretorio` é criado e, em seguida, excluído.

7.2.2 Renomeando e Excluindo Arquivos

A função `os.rename()` é usada para **renomear** um arquivo em Python, enquanto a função `os.remove()`é usada para **excluir** um arquivo. O código 7.6 mostra um exemplo de uso desse recurso.

```
import os

os.rename('meuarquivo.txt', 'meuarquivo_novo.txt')
os.remove('meuarquivo_novo.txt')
```

Exemplo de Código 7.6: Renomeando um arquivo.

Neste exemplo 7.6, o arquivo `meuarquivo.txt` é renomeado para `meuarquivo_novo.txt` pela função `os.rename()` e em seguida é excluído pela função `os.remove()`.

7.2.3 Copiando e Movendo Arquivos

O módulo `shutil` é usado para copiar e mover arquivos em Python. A função `shutil.copy()` é usada para copiar um arquivo, enquanto a função `shutil.move()` é usada para mover um arquivo. O código 7.7 mostra um exemplo de uso desses recursos.

```
import shutil

shutil.copy('meuarquivo.txt', 'diretorio_novo/meuarquivo.txt')
shutil.move('meuarquivo.txt', 'diretorio_novo/meuarquivo_novo.txt')
```

Exemplo de Código 7.7: Copiando e movendo um arquivo.

Neste exemplo 7.7, o arquivo `meuarquivo.txt` é copiado para o diretório `diretorio_novo` e, em seguida, é movido para o mesmo diretório, recebendo um novo nome.

7.2.4 Compactando e Descompactando Arquivos

O módulo `zipfile` é usado para compactar e descompactar arquivos em Python. A função `ZipFile()` é usada para criar um novo arquivo compactado, enquanto a função `extractall()` é usada para descompactar um arquivo. O código 7.8 mostra um exemplo de uso desse recurso.

```
1 import zipfile
2
3 with zipfile.ZipFile('arquivo.zip', 'w') as meu_zip:
4     meu_zip.write('meuarquivo.txt')
5
6 with zipfile.ZipFile('arquivo.zip', 'r') as meu_zip:
7     meu_zip.extractall('meudiretorio')
```

Exemplo de Código 7.8: Compactando um arquivo.

Neste exemplo 7.8, o arquivo `meuarquivo.txt` é compactado em um novo arquivo chamado `arquivo.zip`, e em seguida é descompactado no diretório `meudiretorio`.

7.3 Importando e Utilizando Módulos

Um dos recursos mais poderosos da linguagem Python é a capacidade de importar e usar módulos. Um módulo é um arquivo Python que pode conter funções, classes e outros objetos reutilizáveis. Nesta seção, veremos como importar e utilizar módulos em Python. Veremos também como criar seus próprios módulos e como utilizar pacotes.

7.3.1 Importando Módulos

Para importar um módulo em Python, é necessário usar a instrução `import`. O código 7.9 mostra um exemplo de como importar o módulo `math`.

```
1 import math
```

Exemplo de Código 7.9: Importando um módulo.

Neste exemplo 7.9, o módulo `math` é importado em um programa Python. Também é possível apelidar um módulo ao importá-lo, usando a instrução `as`. O código 7.10 mostra como importar o módulo `math` aplicando um *alias* (apelido).

```
1 import math as m
```

Exemplo de Código 7.10: Importando um módulo com *alias*.

Neste exemplo 7.10, o módulo `math` é importado e apelidado de `m`. Isso permite que você use um nome de módulo mais curto e mais fácil de se lembrar ao longo do código que importou e fará uso do módulo.

7.3.2 Utilizando Recursos Presentes em Um Módulo

Depois de importar um módulo em Python, é possível utilizar seus recursos ao longo do arquivo que o importou. O exemplo 7.11 mostra como usar a função `sqrt()` do módulo `math`.

```
1 import math
2
3 resultado = math.sqrt(25)
```

Exemplo de Código 7.11: Usando função de um módulo importado.

Neste exemplo 7.11, a função `sqrt()` do módulo `math` é usada para calcular a raiz quadrada de 25 e o resultado é armazenado na variável `resultado`. Também é possível importar conteúdos específicos de um módulo usando a instrução `from`. O exemplo 7.12 mostra como importar apenas a função `sqrt()` do módulo `math` e chamá-lo no código que a importou. Neste caso, o nome do módulo pode ser ignorado na chamada.

```
1 from math import sqrt
2
3 resultado = sqrt(25)
```

Exemplo de Código 7.12: Importando uma função de um módulo.

Neste exemplo 7.12, apenas a função `sqrt()` é importada do módulo `math`, e não é necessário usar o nome do módulo ao chamar a função. Aqui também seria possível usar o operador `as` para dar um apelido amigável à função `sqrt()`.

7.3.3 Criando Seus Próprios Módulos

Além de usar módulos existentes em Python, também é possível criar seus próprios módulos para reutilização em seus programas. Para criar um módulo em Python, basta criar um arquivo Python com as funções, classes e objetos que deseja incluir. O exemplo 7.13 mostra o arquivo `meumodulo.py` com a função `minha_funcao()`.

```
1 def minha_funcao():
2     print("Esta é minha função.")
```

Exemplo de Código 7.13: Código do arquivo `meumodulo.py` contendo uma função.

Depois de criar o arquivo Python com as funções e objetos desejados, é possível importar o módulo em outro programa Python usando a instrução `import`. O exemplo 7.14 mostra como usar, em outro módulo do programa, uma função de um módulo criado pelo programador.

```
1 import meumodulo
2
3 meumodulo.minha_funcao()
```

Exemplo de Código 7.14: Usando uma função de um módulo criado.

Neste exemplo 7.14, o módulo `meumodulo` é importado em um programa Python e a função `minha_funcao()` é chamada.

7.3.4 Utilizando Pacotes

Um pacote é um diretório que contém um ou mais módulos Python relacionados. Para importar um pacote em Python, é necessário usar a instrução `import`. O exemplo 7.15 mostra como importar o pacote `numpy`.

```
1 import numpy
```

Exemplo de Código 7.15: Importando um pacote.

Neste exemplo 7.15, o pacote `numpy` é importado em um programa Python. Depois de importar o pacote, é possível utilizar seus módulos, funções e objetos em seu código. Também é possível importar um módulo específico de um pacote usando a instrução `from`. O exemplo 7.16 mostra como importar apenas o módulo `array` do pacote `numpy`.

```
1 from numpy import array
```

Exemplo de Código 7.16: Importando um módulo de um pacote.

Neste exemplo 7.16, apenas o módulo `array` é importado do pacote `numpy`, e não é necessário usar o nome do pacote ao chamar o módulo. Além disso, é possível criar seus próprios pacotes em Python, que consistem em diretórios com arquivos Python. Para criar um pacote em Python, basta criar um diretório com o nome do pacote e adicionar arquivos Python com os módulos e objetos que deseja incluir.

O exemplo 7.17 mostra um pacote chamado `meupacote` contendo um arquivo especial chamado `__init__.py` e o arquivo `meumodulo.py` correspondente a um módulo chamado `meumodulo`.

```
meupacote/
    __init__.py
    meumodulo.py
```

Exemplo de Código 7.17: Diretório com estrutura de arquivos de um pacote.

Depois de criar o pacote com os módulos e objetos desejados, é possível importar o pacote em outro programa Python usando a instrução `import`. Como mostrado anteriormente, você pode importar o pacote inteiro ou somente um módulo do pacote. Você também pode usar o operador `as` para apelidar o recurso importado, visando facilitar sua utilização ao longo do código. Isso é algo muito comum em programas escritos na linguagem Python. O exemplo 7.18 mostra como realizar essa importação.

```
import meupacote.meumodulo

meupacote.meumodulo.minha_funcao()
```

Exemplo de Código 7.18: Importando e usando pacote criado pelo programador.

Neste exemplo 7.18, o pacote `meupacote` é importado em um programa Python e o módulo `meumodulo` é chamado com a função `minha_funcao()`. Neste exemplo, a chamada da função `minha_funcao()`, inclui o nome do pacote (`meupacote`) e o nome do módulo (`meumodulo`) como prefixos da função.

7.4 Documentando Módulos

A documentação de código é uma parte importante do desenvolvimento de software, ajudando a garantir que o código seja fácil de entender e de usar. Em Python, é possível documentar módulos e seus recursos usando *docstring* e outras ferramentas de documentação, conforme veremos nesta seção.

7.4.1 Documentando Funções e Classes

Para documentar funções e classes em Python, é possível usar *docstring*, que é um comentário de texto colocado no início da definição de uma função ou de uma classe. O texto no formato *docstring* deve descrever o que a função ou classe faz e como ela deve ser usada. Basicamente, para usar a documentação *docstring* envolvemos o texto correspondente à documentação entre 3 aspas duplas, sendo que cada trio de aspas duplas fica isolado em sua própria linha. O exemplo 7.19 mostra um *docstring* criado para uma função que retorna o dobro de um número.

```
def dobro(numero):
    """
    Retorna o dobro de um número.

    Parâmetros:
    numero (int ou float): O número a ser dobrado.

    Retorno:
    int ou float: O dobro do número.
    """
    return numero * 2
```

Exemplo de Código 7.19: Documentação de uma função usando *docstring*.

Neste exemplo 7.19, o texto *docstring* descreve a função `dobro()` e inclui informações sobre seus parâmetros e retorno.

7.4.2 Utilizando o *docstring*

O *docstring* é usado por várias ferramentas do ambiente Python, incluindo a função `help()` e a ferramenta de geração de documentação `pydoc`. Para visualizar o *docstring* de uma função ou classe em Python, é possível usar a função `help()`. O exemplo 7.20 mostra o comando usado para visualizar o *docstring* da função `dobro()` definida anteriormente.

```
help(dobro)
```

Exemplo de Código 7.20: Acessando a documentação.

Neste exemplo 7.20, a função `help()` é usada para exibir o *docstring* da função `dobro()`.

7.4.3 Gerando Documentação de Módulos

Além de usar o *docstring* para documentar funções e classes em Python, também é possível gerar a documentação completa de um módulo Python usando a ferramenta `pydoc`. Para gerar a documentação de um módulo Python, basta executar o comando `pydoc` seguido pelo nome do módulo. O exemplo 7.21 mostra como gerar a documentação para o módulo `meumodulo.py`.

```
python -m pydoc meumodulo
```

Exemplo de Código 7.21: Gerando a documentação de um módulo usando `pydoc`.

Neste exemplo 7.21, o comando `pydoc` é usado para gerar a documentação para o módulo `meumodulo.py`.

7.5 Gerenciando Dependências de Módulos

Ao desenvolver projetos em Python, é comum depender de módulos externos para fornecer funcionalidades adicionais. Nesta seção, veremos como gerenciar dependências de módulos em Python, incluindo como utilizar gerenciadores de pacotes para instalar e atualizar pacotes e como gerenciar versões de pacotes em seus projetos.

7.5.1 Utilizando Gerenciadores de Pacotes

Um gerenciador de pacotes é uma ferramenta que ajuda a gerenciar as dependências de um projeto e que facilita a instalação e atualização de pacotes. Em Python, o gerenciador de pacotes mais popular é o `pip`. Para utilizar o `pip`, basta ter o Python instalado no seu sistema.

7.5.2 Instalando e Atualizando Pacotes

Para instalar um pacote em Python usando o `pip`, basta executar o comando `pip install` seguido do nome do pacote. O exemplo 7.22 mostra como instalar o pacote `requests`.

```
pip install requests
```

Exemplo de Código 7.22: Instalando um pacote usando `pip`.

Neste exemplo 7.22, o pacote `requests` é instalado usando o `pip`. De forma bastante semelhante, para atualizar um pacote em Python usando o `pip` basta executar o comando `pip install` seguido do nome do pacote e da opção `--upgrade`. O exemplo 7.23 mostra como atualizar o pacote `requests`.

```
pip install requests --upgrade
```

Exemplo de Código 7.23: Atualizando um pacote com `pip`.

Neste exemplo 7.23, o pacote `requests` é atualizado para a versão mais recente usando o `pip`.

7.5.3 Gerenciando Versões de Pacotes

Além de instalar e atualizar pacotes, também é importante gerenciar as versões de pacotes em seus projetos. Para fazer isso, é possível usar um arquivo `requirements.txt`, que lista todas as dependências do projeto, incluindo as versões específicas de cada pacote. O exemplo 7.24 mostra uma linha existente no arquivo `requirements.txt` de um projeto qualquer que depende do pacote `requests` na versão 2.25.1.

```
requests==2.25.1
```

Exemplo de Código 7.24: Informando versão de pacote no arquivo de dependências.

Neste exemplo 7.24, o arquivo `requirements.txt` mostra que o programa em questão depende do pacote `requests` na versão 2.25.1. Para instalar todas as dependências de um projeto presentes em um arquivo `requirements.txt` basta executar o comando `pip install` seguido da opção `-r` e do nome do arquivo. O exemplo 7.25 mostra como instalar todas as dependências do projeto a partir do arquivo `requirements.txt`.

```
pip install -r requirements.txt
```

Exemplo de Código 7.25: Instalando todos os pacotes de um arquivo de dependências.

Neste exemplo 7.25, todas as dependências do projeto listadas no arquivo `requirements.txt` são instaladas usando o `pip`.

7.6 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 7.1 Crie um programa que solicite ao usuário um nome de arquivo e exiba seu conteúdo na tela. ■

Exercício 7.2 Crie um programa que leia um arquivo texto e exiba quantas linhas ele possui. ■

Exercício 7.3 Crie um programa que leia um arquivo texto, inverta o conteúdo de cada linha e salve o resultado em um novo arquivo. ■

Exercício 7.4 Crie um programa que solicite ao usuário o nome de um arquivo e que renomeie esse arquivo adicionando a palavra “renomeado” ao nome existente e mantendo sua extensão. ■

Exercício 7.5 Crie um programa que solicite ao usuário o nome de um arquivo e exclua esse arquivo. ■

Exercício 7.6 Crie um programa que solicite ao usuário um nome de arquivo, copie-o para um novo arquivo mudando a extensão para ".copy" e exiba o resultado na tela. ■

Exercício 7.7 Crie um programa que crie um diretório chamado `temp` e, dentro desse diretório, crie também um arquivo chamado "temp.txt". ■

Exercício 7.8 Crie um programa que exclua o diretório criado no exercício anterior com todo o seu conteúdo (cuidado para não excluir a pasta errada). ■

Exercício 7.9 Crie um programa que utilize o módulo `random` para gerar um número aleatório entre 1 e 10. ■

Exercício 7.10 Crie um programa que utilize um módulo personalizado para exibir a data e hora atuais do sistema. ■

Exercício 7.11 Crie um programa que utilize o `pydoc` para gerar a documentação de um módulo criado por você. ■

Exercício 7.12 Crie um programa que utilize o `pydoc` para gerar a documentação de uma função criada por você. ■

Exercício 7.13 Utilize o `pip` para instalar um pacote Python chamado `fastapi`. ■

Exercício 7.14 Utilize o `pip` para atualizar um pacote Python chamado `fastapi`. ■

Exercício 7.15 Utilize o `pip` para listar os pacotes Python instalados no sistema. ■

7.7 Considerações Sobre o Capítulo

Neste capítulo, vimos como trabalhar com arquivos e módulos em Python, incluindo como abrir, ler e escrever arquivos, manipular arquivos e diretórios, importar e utilizar módulos, gerenciar dependências de módulos e seguir boas práticas de programação em Python. Ao dominar esses conceitos, você poderá criar aplicativos Python mais sofisticados e eficientes. No próximo capítulo, veremos como criar e manipular classes e objetos.

8. Classes e Objetos

A programação orientada a objetos (POO) é um paradigma fundamental de programação em Python e em muitas outras linguagens de programação modernas. A POO é uma abordagem modular para a criação de software, que se concentra na criação de objetos que contêm dados e funcionalidades relacionadas. Em Python, a POO é suportada por meio de classes e objetos, que são construções fundamentais para modelar e desenvolver sistemas de software nos dias atuais.

Este capítulo tem como objetivo fornecer uma introdução completa ao uso de classes e objetos em Python. Começaremos explicando o que é a programação orientada a objetos e suas vantagens. Em seguida, iremos apresentar os conceitos de classe e objeto, explicando como as classes são definidas, como os objetos são criados e como eles interagem entre si. Também exploraremos recursos avançados de POO, como encapsulamento, herança e polimorfismo.

8.1 Classes

As classes são a base da programação orientada a objetos em Python. Elas são usadas para definir o comportamento e as características dos objetos que serão criados a partir delas. Uma classe é uma estrutura de dados que pode conter dados (atributos) e funções (métodos) que operam nesses dados.

Nesta seção, vamos explorar em detalhes o que é uma classe em Python e como ela é usada para criar objetos. Também vamos abordar os conceitos de atributos e métodos, que são as principais características de uma classe. Além disso, veremos os conceitos de encapsulamento, herança e polimorfismo, que são recursos avançados de POO em Python. Por fim, veremos alguns exemplos práticos de criação de classes em Python.

8.1.1 Definição de Classe

Uma classe é uma estrutura de dados que define um novo tipo de objeto em Python. Ela contém atributos e métodos que descrevem as características e o comportamento desse objeto, respectivamente. A definição de uma classe começa com a palavra-chave `class` seguida do nome da classe, que deve ser única dentro do programa.

8.1.2 Atributos e Métodos

Os atributos de uma classe são variáveis que contêm os dados que descrevem o objeto. Eles são acessados por meio de objetos da classe. Por outro lado, os métodos de uma classe são funções que operam sobre os dados da classe e podem ser acessados por meio de objetos da classe.

8.1.3 Encapsulamento

O encapsulamento é um conceito importante da POO que permite que os dados e o comportamento de uma classe sejam protegidos contra acessos não autorizados. Em Python, isso é feito usando os modificadores de acesso, que são `public`, `private` e `protected`.

8.1.4 Herança

A herança é um recurso avançado da POO em que uma classe é derivada de outra classe, chamada classe pai ou superclasse. A classe derivada, chamada classe filha ou subclasse, herda os atributos e métodos da classe pai e pode adicionar seus próprios atributos e métodos.

8.1.5 Polimorfismo

O polimorfismo é a capacidade de usar uma única interface para representar objetos de diferentes classes. Isso permite que objetos de diferentes tipos sejam usados de forma intercambiável em um programa.

8.1.6 Exemplos de Classes

Para ilustrar os conceitos apresentados nesta seção, veremos agora alguns exemplos de classes em Python. Esses exemplos incluem classes simples, que possuem apenas alguns atributos e métodos, e classes mais complexas, que podem incluir herança e polimorfismo.

Classe Pessoa

A classe `Pessoa` é um exemplo simples de uma classe em Python. Ela contém dois atributos

(`nome` e `idade`) e um método que permite que a pessoa diga uma mensagem (`falar()`). O código 8.1 mostra um exemplo de uma classe em Python.

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def falar(self, mensagem):
        print(f"{self.nome} diz: {mensagem}")
```

Exemplo de Código 8.1: Código de uma classe em Python.

Neste exemplo 8.1, o método especial `__init__()` é usado para inicializar os atributos nome e idade da pessoa. O método `falar()` é usado para permitir que a pessoa diga uma mensagem.

Classe Animal

A classe `Animal` é um exemplo mais complexo de uma classe em Python. Ela contém vários atributos (`nome`, `idade`, `som` e `cor`) e vários métodos (`dormir()`, `comer()` e `fazer_barulho()`) que permitem que o animal realize diferentes ações. O de código abaixo:

```
class Animal:
    def __init__(self, nome, idade, som, cor):
        self.nome = nome
        self.idade = idade
        self.som = som
        self.cor = cor

    def dormir(self):
        print(f"{self.nome} está dormindo")

    def comer(self):
        print(f"{self.nome} está comendo")

    def fazer_barulho(self):
        print(f"{self.nome} faz {self.som}")
```

Exemplo de Código 8.2: Classe com atributos e métodos.

Neste exemplo, a classe `Animal` contém os atributos `nome`, `idade`, `som` e `cor`. Também inclui os métodos `dormir()`, `comer()` e `fazer_barulho()`, que permitem que o animal realize diferentes ações.

Esses exemplos de classe são apenas o começo do que é possível fazer com classes em Python. As classes podem ser usadas para modelar qualquer tipo de objeto ou sistema que você deseja criar. Com a prática, você poderá criar classes mais complexas e poderosas que atendam às necessidades do seu programa.

8.2 Objetos

Em programação orientada a objetos, os objetos são as instâncias das classes. Eles representam entidades do mundo real e contêm dados e métodos relacionados. Nesta seção, vamos explorar em detalhes o que é um objeto em Python e como ele é criado a partir de uma classe. Também vamos abordar os conceitos de atributos e métodos de objetos, que são os principais componentes de um objeto. Além disso, veremos como referenciar objetos em Python.

8.2.1 Definição de Objeto

Um objeto é uma instância de uma classe. Ele contém os atributos (com valores) e métodos definidos na classe e pode ser criado a partir da classe. Enquanto a classe é uma representação abstrata de algo do mundo real dentro do programa, um objeto é a representação concreta.

8.2.2 Instanciação de Objetos

A instanciação de objetos é o processo de criar um objeto a partir de uma classe. Para criar um objeto, você usa a palavra-chave `new` seguida do nome da classe e seus parâmetros, se houver. Aqui está um exemplo:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

pessoa1 = Pessoa("João", 25)
pessoa2 = Pessoa("Maria", 30)
```

Exemplo de Código 8.3: .

Neste exemplo, criamos uma classe chamada `Pessoa` com um construtor que recebe dois parâmetros, `nome` e `idade`. Em seguida, criamos dois objetos, `pessoa1` e `pessoa2`, a partir da classe `Pessoa`, passando seus parâmetros correspondentes.

8.2.3 Atributos de Objeto

Os atributos de um objeto são variáveis que contêm os dados que descrevem o objeto. Eles são acessados usando a notação de ponto (`.`) com o nome do objeto e do atributo. Aqui está um exemplo:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

pessoa1 = Pessoa("João", 25)
print(pessoa1.nome)
print(pessoa1.idade)
```

Exemplo de Código 8.4: .

Neste exemplo, criamos um objeto `pessoa1` a partir da classe `Pessoa` e acessamos seus atributos `nome` e `idade` usando a notação de ponto.

8.2.4 Métodos de Objeto

Os métodos de um objeto são funções que operam sobre os dados do objeto. Eles são acessados usando a notação de ponto (`.`) com o nome do objeto e do método. Aqui está um exemplo:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def dizer_idade(self):
        print(f"{self.nome} tem {self.idade} anos")

pessoa1 = Pessoa("João", 25)
pessoa1.dizer_idade()
```

Exemplo de Código 8.5: .

Neste exemplo, criamos um objeto `pessoa1` a partir da classe `Pessoa` e chamamos o método `dizer_idade()` usando a notação de ponto.

8.2.5 Referenciando Objetos em Python

Os objetos em Python são referenciados por variáveis. Quando você cria um objeto, ele é armazenado em memória e uma referência a esse objeto é armazenada na variável. Aqui está

um exemplo:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

pessoa1 = Pessoa("João", 25)
pessoa2 = pessoa1
pessoa1.nome = "Pedro"
print(pessoa2.nome)
```

Exemplo de Código 8.6: .

Neste exemplo, criamos dois objetos, `pessoa1` e `pessoa2`, a partir da classe `Pessoa`. Em seguida, criamos uma referência ao objeto `pessoa1` na variável `pessoa2`. Quando alteramos o valor do atributo nome do objeto `pessoa1`, isso afeta a referência do objeto em `pessoa2`, que aponta para o mesmo objeto na memória. É importante entender como as referências de objetos funcionam em Python para evitar erros de programação e vazamentos de memória.

Compreender como criar objetos, acessar seus atributos e métodos e referenciá-los em Python é fundamental para o desenvolvimento de programas orientados a objetos em Python. Com esses conceitos em mente, você estará pronto para criar classes e objetos poderosos e flexíveis em seus projetos.

8.3 Criando Classes e Objetos

A criação de classes e objetos em Python é uma das principais características da programação orientada a objetos em Python. Nesta seção, vamos explorar em detalhes como criar classes e objetos em Python. Vamos abordar a definição de uma classe, a criação de um objeto, atributos e métodos de classe, construtores e destrutores. Além disso, veremos exemplos práticos de criação de classes e objetos em Python.

8.3.1 Definindo uma Classe

Para definir uma classe em Python, usamos a palavra-chave `class` seguida do nome da classe. Veja um exemplo abaixo:

Neste exemplo, criamos uma classe chamada `Pessoa` usando a palavra-chave `class` e o nome da classe. A instrução `pass` indica que a classe não possui nenhum conteúdo até o momento.

```
class Pessoa:
    pass
```

Exemplo de Código 8.7: .

8.3.2 Criando um Objeto

Para criar um objeto em Python, usamos a sintaxe a seguir:

```
nome_do_objeto = nome_da_classe()
```

Exemplo de Código 8.8: .

Veja um exemplo abaixo:

```
class Pessoa:
    pass

pessoa1 = Pessoa()
```

Exemplo de Código 8.9: .

Neste exemplo, criamos uma classe chamada `Pessoa` e, em seguida, criamos um objeto chamado `pessoa1` a partir da classe `Pessoa`.

8.3.3 Atributos e Métodos de Classe

Os atributos e métodos são as principais características de uma classe em Python. Os atributos são variáveis que descrevem as características do objeto, enquanto os métodos são funções que operam sobre esses dados. Aqui está um exemplo de uma classe com atributos e métodos:

Neste exemplo, criamos uma classe chamada `Pessoa` com dois atributos (`nome` e `idade`) e dois métodos (`dizer_nome` e `dizer_idade`). Em seguida, criamos um objeto `pessoa1` a partir da classe `Pessoa` e chamamos os métodos `dizer_nome` e `dizer_idade` para exibir o nome e a idade da pessoa.

8.3.4 Construtores e Destrutores

Os construtores e destrutores são métodos especiais em uma classe Python que são usados para inicializar e destruir um objeto. O construtor é chamado automaticamente quando um objeto é criado, enquanto o destrutor é chamado automaticamente quando o objeto é destruído. Aqui está um exemplo de uma classe com construtor e destrutor:

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def dizer_nome(self):
        print(f"Meu nome é {self.nome}")

    def dizer_idade(self):
        print(f"Tenho {self.idade} anos")

pessoa1 = Pessoa("João", 25)
pessoa1.dizer_nome()
pessoa1.dizer_idade()
```

Exemplo de Código 8.10: .

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
        print("Construindo objeto...")

    def __del__(self):
        print("Destruindo objeto...")

    def dizer_nome(self):
        print(f"Meu nome é {self.nome}")

    def dizer_idade(self):
        print(f"Tenho {self.idade} anos")

pessoa1 = Pessoa("João", 25)
pessoa1.dizer_nome()
pessoa1.dizer_idade()
del pessoa1
```

Exemplo de Código 8.11: .

Neste exemplo, criamos uma classe chamada `Pessoa` com um construtor `__init__()` e um destrutor `__del__()`. Em seguida, criamos um objeto `pessoa1` a partir da classe `Pessoa` e chamamos os métodos `dizer_nome` e `dizer_idade` para exibir o nome e a idade da pessoa. Por fim, usamos a instrução `del` para destruir o objeto `pessoa1`, que chama o destrutor automaticamente.

8.3.5 Exemplos de Criação de Classes e Objetos

Aqui estão alguns exemplos de criação de classes e objetos em Python:

```
class Retangulo:
    def __init__(self, altura, largura):
        self.altura = altura
        self.largura = largura

    def area(self):
        return self.altura * self.largura

retangulo1 = Retangulo(10, 20)
print(retangulo1.area())
```

Exemplo de Código 8.12: .

Neste exemplo, criamos uma classe chamada `Retangulo` com um construtor `__init__()` e um método `area()`. Em seguida, criamos um objeto `retangulo1` a partir da classe `Retangulo` e chamamos o método `area()` para calcular a área do retângulo.

```
class ContaBancaria:
    def __init__(self, nome, saldo):
        self.nome = nome
        self.saldo = saldo

    def depositar(self, valor):
        self.saldo += valor

    def sacar(self, valor):
        if self.saldo < valor:
            print("Saldo insuficiente")
        else:
            self.saldo -= valor

    def imprimir_saldo(self):
        print(f"O saldo da conta é R${self.saldo:.2f}")

conta1 = ContaBancaria("João", 1000)
conta1.depositar(500)
conta1.sacar(300)
conta1.imprimir_saldo()
```

Exemplo de Código 8.13: .

Neste exemplo, criamos uma classe chamada `ContaBancaria` com um construtor `__init__()`

e métodos para depositar, sacar e imprimir o saldo da conta. Em seguida, criamos um objeto `conta1` a partir da classe `ContaBancaria` e realizamos algumas operações de depósito e saque na conta, antes de imprimir o saldo final.

A criação de classes e objetos em Python é uma das principais características da programação orientada a objetos em Python. Com esses conceitos em mente, você pode criar programas mais flexíveis e poderosos, com código mais organizado e fácil de manter. Com as informações apresentadas nesta seção, você pode começar a criar suas próprias classes e objetos em Python e implementar projetos orientados a objetos mais complexos e funcionais.

8.4 Exemplos Práticos

Nesta seção, veremos alguns exemplos práticos de como usar classes e objetos em Python. Vamos abordar a modelagem de objetos, a criação de jogos com POO, o desenvolvimento de software com POO e outros exemplos práticos.

8.4.1 Modelagem de Objetos

A modelagem de objetos é uma das principais áreas de aplicação da programação orientada a objetos. Nesta área, podemos modelar objetos do mundo real e criar classes e objetos em Python que representam esses objetos. Aqui está um exemplo de modelagem de objetos em Python:

Neste exemplo, modelamos um objeto `Carro` com vários atributos, incluindo marca, modelo, cor, ano, velocidade máxima e velocidade atual. Em seguida, criamos métodos para acelerar e frear o carro, bem como imprimir o status do carro. Por fim, criamos um objeto `carro1` a partir da classe `Carro` e realizamos algumas operações, antes de imprimir o status do carro.

8.4.2 Criação de Jogos com POO

A criação de jogos é outra área em que a programação orientada a objetos é amplamente utilizada. Nesta área, podemos criar classes e objetos em Python para representar personagens, cenários, itens etc. Aqui está um exemplo de criação de um jogo com POO em Python:

8.4.3 Desenvolvimento de Software com POO

O desenvolvimento de software é outra área em que a programação orientada a objetos é muito utilizada. Nesta área, podemos criar classes e objetos em Python para representar diferentes módulos, funções e interfaces do software. Aqui está um exemplo de como usar POO para

```
class Carro:
    def __init__(self, marca, modelo, cor, ano, velocidade_maxima):
        self.marca = marca
        self.modelo = modelo
        self.cor = cor
        self.ano = ano
        self.velocidade_maxima = velocidade_maxima
        self.velocidade_atual = 0

    def acelerar(self, quantidade):
        if self.velocidade_atual + quantidade > self.velocidade_maxima:
            self.velocidade_atual = self.velocidade_maxima
        else:
            self.velocidade_atual += quantidade

    def frear(self, quantidade):
        if self.velocidade_atual - quantidade < 0:
            self.velocidade_atual = 0
        else:
            self.velocidade_atual -= quantidade

    def imprimir_status(self):
        print(f"Marca: {self.marca}\nModelo: {self.modelo}\nCor: {self.cor}\nAno: {self.ano}\nVe

carro1 = Carro("Fiat", "Uno", "Branco", 2010, 180)
carro1.acelerar(100)
carro1.frear(50)
carro1.imprimir_status()
```

Exemplo de Código 8.14: .

desenvolver um programa simples de gerenciamento de estoque:

Neste exemplo, criamos uma classe `Produto` com atributos para nome, preço e quantidade. Em seguida, criamos métodos para adicionar e remover a quantidade do produto, bem como imprimir o status do produto. Usando essa classe, podemos criar um programa simples de gerenciamento de estoque:

Neste exemplo, criamos dois objetos, `produto1` e `produto2` a partir da classe `Produto` e realizamos algumas operações, antes de imprimir o status dos produtos.

```
class Jogador:
    def __init__(self, nome, vida, ataque, defesa):
        self.nome = nome
        self.vida = vida
        self.ataque = ataque
        self.defesa = defesa

    def atacar(self, inimigo):
        dano = self.ataque - inimigo.defesa
        if dano > 0:
            inimigo.vida -= dano

    def imprimir_status(self):
        print(f"Nome: {self.nome}\nVida: {self.vida}\nAtaque: {self.ataque}\
```

Exemplo de Código 8.15: .

```
class Produto:
    def __init__(self, nome, preco, quantidade):
        self.nome = nome
        self.preco = preco
        self.quantidade = quantidade

    def adicionar_quantidade(self, quantidade):
        self.quantidade += quantidade

    def remover_quantidade(self, quantidade):
        if quantidade > self.quantidade:
            print("Quantidade inválida")
        else:
            self.quantidade -= quantidade

    def imprimir_status(self):
        print(f"Nome: {self.nome}\nPreço: R\${self.preco:.2f}\nQuantidade: {self.quantidade}")
```

Exemplo de Código 8.16: .

8.4.4 Outros exemplos práticos

Além dos exemplos apresentados acima, existem muitas outras áreas em que a programação orientada a objetos é amplamente utilizada, incluindo o desenvolvimento de aplicativos móveis, a criação de sistemas de gerenciamento de banco de dados, a criação de robôs e muito mais. A partir dos conceitos apresentados neste capítulo, você pode explorar outras áreas de aplicação da POO e criar seus próprios projetos em Python.

```
produto1 = Produto("Notebook", 3000, 10)
produto2 = Produto("Impressora", 500, 5)

produto1.adicionar_quantidade(5)
produto1.remover_quantidade(2)
produto2.remover_quantidade(10)

produto1.imprimir_status()
produto2.imprimir_status()
```

Exemplo de Código 8.17: .

8.5 Exercícios Propostos

A fazer.

8.6 Considerações Sobre o Capítulo

Com a programação orientada a objetos em Python, podemos criar programas mais flexíveis, organizados e fáceis de manter. Com os conceitos de classes e objetos em mente, podemos modelar objetos do mundo real e criar programas mais poderosos e funcionais. Neste capítulo, vimos como criar classes e objetos em Python, bem como exemplos práticos de modelagem de objetos, criação de jogos e desenvolvimento de software. Com essas informações, você pode começar a criar seus próprios projetos orientados a objetos em Python e explorar outras áreas de aplicação da POO. O próximo capítulo mostra como usar funções assíncronas em Python, um recurso importantíssimo para se aproveitar melhor os processadores atuais.

9. Tratamento de Exceções

As exceções em Python são erros que ocorrem durante a execução de um programa. Esses erros interrompem o fluxo normal de execução e podem ser causados por diferentes fatores, como entradas inválidas do usuário, falhas na conexão de rede, erros de digitação, entre outros.

Quando ocorre uma exceção, Python gera uma mensagem de erro que descreve o problema encontrado. Essas mensagens de erro são chamadas de "rastreamento de pilha" (*traceback*) e fornecem informações úteis para depurar o código.

Tratar exceções em Python significa lidar com essas situações de erro de forma apropriada, evitando que o programa pare de funcionar inesperadamente. Um tratamento adequado de exceções pode ajudar a manter o programa em execução, mesmo que ocorram erros, permitindo que o usuário possa entender o que aconteceu e/ou corrigir o problema.

Ao lidar com exceções em Python, é importante identificar o tipo de exceção que ocorreu e tratar cada tipo de erro de forma apropriada. Por exemplo, se uma exceção de divisão por zero for lançada, o programa deve tratar essa exceção de forma diferente de uma exceção de falha de conexão de rede.

Além disso, é importante usar o tratamento de exceções de forma criteriosa, evitando capturar exceções genéricas que possam ocultar erros importantes. O tratamento adequado de exceções pode melhorar a qualidade do código e torná-lo mais robusto, aumentando a confiabilidade do programa em geral. Este capítulo aborda esse assunto.

9.1 Tratando Exceções

Uma das formas mais comuns de lidar com exceções em Python é usando os blocos `try/except`. Um bloco `try/except` permite que o programa tente executar um trecho de código e, se uma exceção ocorrer, o programa poderá tratá-la de forma apropriada, sem interromper a execução do programa.

O bloco `try/except` é composto por duas partes principais: o bloco `try`, que contém o código a ser executado, e o bloco `except`, que contém o tratamento de possíveis exceções. O exemplo 9.1 mostra um exemplo simples de bloco `try/except`.

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Erro: divisão por zero!")
```

Exemplo de Código 9.1: Bloco simples de tratamento de exceções.

Neste exemplo 9.1, o bloco `try` tenta executar uma divisão por zero, o que causaria uma exceção `ZeroDivisionError`. O bloco `except` captura essa exceção e exibe uma mensagem de erro. Vamos a mais um exemplo. O código 9.2 mostra um bloco `try/except` utilizado para tratar exceções possíveis de ocorrer ao tentar abrir um arquivo.

```
try:
    with open("arquivo.txt", "r") as arquivo:
        conteudo = arquivo.read()
except FileNotFoundError:
    print("Erro: arquivo não encontrado!")
```

Exemplo de Código 9.2: Bloco de tratamento de exceções de abertura de arquivo.

Neste exemplo 9.2, o bloco `try` tenta abrir o arquivo "arquivo.txt"para leitura. Se o arquivo não existir, uma exceção `FileNotFoundError` será lançada. O bloco `except` captura essa exceção e exibe uma mensagem de erro. É possível utilizar vários blocos `except` para capturar diferentes tipos de exceções. O exemplo 9.3 mostra como tratar múltiplas exceções no mesmo bloco.

Neste exemplo 9.3, o bloco `try` tenta ler um número inteiro do usuário e fazer uma divisão por esse número. Se o usuário digitar zero, uma exceção `ZeroDivisionError` será lançada. Se o usuário digitar um valor não numérico, uma exceção `ValueError` será lançada. Cada bloco `except` captura a exceção correspondente e exibe uma mensagem de erro apropriada.

```
try:
    x = int(input("Digite um número inteiro: "))
    resultado = 10 / x
except ZeroDivisionError:
    print("Erro: divisão por zero!")
except ValueError:
    print("Erro: valor inválido!")
```

Exemplo de Código 9.3: Tratando múltiplas exceções no mesmo bloco.

Quando lidamos com exceções em Python, é sempre importante identificar o tipo de exceção que ocorreu e tratar cada tipo de erro de forma apropriada e individualizada. Para isso, inclusive, é que existe a possibilidade de capturar exceções específicas usando vários blocos `except`.

Cada tipo de exceção é representado por uma classe. Por exemplo, a exceção de divisão por zero é representada pela classe `ZeroDivisionError`, enquanto a exceção de erro de conversão de um valor entrado pelo usuário é da classe `ValueError` e a exceção lançada quando um arquivo não é encontrado é `FileNotFoundError`. Resumindo, para capturar uma exceção específica, basta utilizar um bloco `except` seguido do nome da classe da exceção.

Outra informação muito importante a considerar é que a classe ancestral de todas as classes de exceção é a classe `Exception`. Portanto, se um bloco `except Exception` for o primeiro tratador de um bloco `try`, qualquer exceção que ocorra neste bloco `try` será sempre tratada por ele, o que acarreta na perda da especificidade do tratamento da exceção. Em alguns casos, é interessante ter um bloco `except Exception` no fim do bloco de tratamento para capturar exceções imprevistas, mas isso não dispensa o tratamento individualizado das exceções.

Em síntese, o bloco `try/except` é uma ferramenta importante para lidar com exceções em Python, mas é importante utilizá-lo com moderação e de forma correta, evitando capturar exceções genéricas no início do bloco. Além disso, é importante sempre exibir mensagens de erro claras, informativas e o mais específicas possível, pois isso vai ajudar o usuário a entender melhor o que aconteceu para tentar corrigir o problema.

9.2 Protegendo Recursos

Em Python, é possível utilizar o bloco `finally` em conjunto com o bloco `try/except` para garantir a execução de um determinado trecho de código, independentemente de ocorrer uma exceção ou não.

O bloco `finally` é executado sempre que um bloco `try` é executado, independentemente de ocorrer ou não uma exceção. Isso significa que o bloco `finally` é executado mesmo que uma exceção tenha sido lançada e não tenha sido capturada por um bloco `except`.

Um exemplo de uso do bloco `finally` é a garantia de que um recurso seja fechado, como um arquivo aberto em modo de leitura. Mesmo que ocorra uma exceção ao ler o arquivo, é importante garantir que o arquivo seja fechado após a sua utilização. O bloco `finally` pode ser utilizado para garantir que o arquivo seja fechado, independentemente de ocorrer ou não uma exceção. O exemplo 9.4 mostra como fazer isso.

```
try:
    arquivo = open("arquivo.txt", "r")
    conteudo = arquivo.read()
except FileNotFoundError:
    print("Erro: arquivo não encontrado!")
finally:
    arquivo.close()
```

Exemplo de Código 9.4: Protegendo recursos com bloco `finally`.

Neste exemplo 9.4, o bloco `try` tenta abrir o arquivo "arquivo.txt" para leitura e ler o seu conteúdo. Se o arquivo não existir, uma exceção `FileNotFoundError` será lançada. O bloco `finally` garante que o arquivo seja fechado após a sua utilização, independentemente de ocorrer ou não uma exceção.

Outro exemplo de uso do bloco `finally` pode ser para liberar recursos como conexões de banco de dados, *sockets* de rede, entre outros. É importante destacar que o bloco `finally` é executado sempre que um bloco `try` é executado, inclusive em casos onde uma exceção não é capturada. Isso significa que o bloco `finally` pode ser utilizado para garantir a execução de um trecho de código importante, mesmo em situações de erro.

9.3 Exceções Personalizadas

Em Python, é possível criar suas próprias exceções para lidar com erros específicos do seu programa. Para criar uma exceção personalizada, basta criar uma nova classe que herde da classe base `Exception` ou de uma de suas subclasses. Ao criar uma exceção personalizada, é importante definir uma mensagem de erro clara e informativa, que ajude o usuário a entender o que causou o problema. Além disso, é importante definir um nome descritivo para a exceção, que reflita o tipo de erro que está sendo tratado. O exemplo 9.5 mostra uma exceção personalizada.

```
class ValorInvalido(Exception):
    def __init__(self, parametro, valor):
        self.parametro = parametro
        self.valor = valor
        self.mensagem = f"O valor '{valor}' é inválido para o parâmetro '{parametro}'"
        super().__init__(self.mensagem)
```

Exemplo de Código 9.5: Classe de exceção personalizada.

Neste exemplo 9.5, a classe `ValorInvalido` herda da classe base `Exception` e define três atributos: o nome do parâmetro (`parametro`), o valor inválido (`valor`) e a mensagem de erro (`mensagem`). Vale lembrar que o método `__init__` é chamado sempre que uma nova instância da classe é criada, e, neste caso, é nele que definimos a mensagem de erro a ser exibida quando a exceção for capturada. Para lançar a exceção personalizada, basta criar uma nova instância da classe e lançá-la usando a palavra-chave `raise`, como no exemplo 9.6.

```
def calcula_idade(ano_nascimento):
    if ano_nascimento < 1900 or ano_nascimento > 2022:
        raise ValorInvalido("ano_nascimento", ano_nascimento)
    else:
        return 2022 - ano_nascimento
```

Exemplo de Código 9.6: Lançando a exceção personalizada.

Neste exemplo 9.6, a função `calcula_idade` recebe um ano de nascimento como parâmetro e verifica se ele está dentro do intervalo válido. Se o ano de nascimento for inválido, a função lança a exceção `ValorInvalido`, informando o nome do parâmetro e o valor inválido. Para capturar a exceção personalizada, basta utilizar um bloco `try/except` com o nome da classe da exceção, como mostra o exemplo 9.7.

```
try:
    idade = calcula_idade(1880)
except ValorInvalido as erro:
    print(erro.mensagem)
# Saída: O valor 1880 é inválido para o parâmetro 'ano_nascimento'
```

Exemplo de Código 9.7: Tratando a exceção personalizada.

Neste exemplo 9.7, o bloco `try` tenta chamar a função `calcula_idade` com um ano de nascimento inválido. O bloco `except` captura a exceção `ValorInvalido`, apelida ela de `erro`, e exibe a mensagem de erro personalizada.

Em resumo, criar suas próprias exceções personalizadas em Python pode ajudar a lidar com erros específicos no seu programa e exibir mensagens de erro mais informativas para o usuário. Ao criar uma exceção personalizada, é importante definir uma mensagem de erro clara e informativa, lembrando sempre de utilizar o comando `raise` nos pontos em que for necessário lançar a exceção.

9.4 Encadeamento de Exceções

O encadeamento de exceções, também conhecido como *exception chaining* em inglês, é uma técnica em Python que permite capturar uma exceção e lançar uma nova exceção com informações adicionais, mantendo o rastreamento de pilha da exceção originalmente lançada.

Essa técnica é útil em situações onde uma exceção é lançada por uma biblioteca ou módulo que não possui informações suficientes para identificar o erro. Ao encadear a exceção, é possível adicionar informações ao rastreamento de pilha, facilitando a identificação e correção do erro. Em Python, é possível encadear exceções utilizando a palavra-chave `from`. O exemplo 9.8 ilustra essa situação.

```
1 try:
2     arquivo = open("arquivo.txt", "r")
3     conteudo = arquivo.read()
4 except FileNotFoundError as erro:
5     raise ValueError("Erro ao ler arquivo") from erro
```

Exemplo de Código 9.8: Encadeando uma exceção.

Neste exemplo 9.8, o bloco `try` tenta abrir o arquivo "arquivo.txt" para leitura e ler o seu conteúdo. Se o arquivo não existir, uma exceção `FileNotFoundError` será lançada. O bloco `except` captura essa exceção e lança uma nova exceção `ValueError`, encadeando a exceção original com a palavra-chave `from`.

Ao encadear a exceção, a nova exceção `ValueError` mantém o rastreamento de pilha da exceção original `FileNotFoundError`, facilitando a identificação do erro. Isso significa que, ao capturar a exceção `ValueError`, é possível acessar as informações da exceção original usando a palavra-chave `__cause__` do objeto de exceção capturado. O exemplo 9.9 mostra como usar esse recurso.

Neste exemplo 9.9, o bloco `try` tenta abrir o arquivo "arquivo.txt" para leitura e ler o seu conteúdo. Se ocorrer uma exceção, o bloco `except` captura a exceção `ValueError` e exibe uma mensagem

```
try:
    arquivo = open("arquivo.txt", "r")
    conteudo = arquivo.read()
except ValueError as erro2:
    print("Erro: " + str(erro2))
    if erro2.__cause__:
        print("Causa do erro: " + str(erro2.__cause__))
```

Exemplo de Código 9.9: Acessando exceção original após encadeamento.

de erro. Em seguida, o bloco `if` verifica se a exceção possui uma causa (`cause`), e exibe a mensagem de erro da causa, se existir.

O acesso à exceção original encadeada é útil para identificar e corrigir erros em bibliotecas e módulos que não possuem informações suficientes para lidar com exceções específicas. Em resumo, a palavra-chave cause permite acessar a exceção original que foi encadeada em uma nova exceção, facilitando a identificação e correção de erros em bibliotecas e módulos.

Outro exemplo de encadeamento de exceções pode ser utilizado para tratar exceções assíncronas em Python, que são aquelas exceções lançadas no momento da chamada a um método assíncrono qualquer. O exemplo 9.10 mostra como lidar com essa situação.

```
async def exemplo_assincrono():
    try:
        await alguma_funcao()
    except Exception as erro:
        raise MinhaExcecao("Erro assíncrono") from erro
```

Exemplo de Código 9.10: Encadeando uma exceção oriunda de chamada assíncrona.

Neste exemplo 9.10, a função `exemplo_assincrono` tenta chamar a função `alguma_funcao` de forma assíncrona. Se ocorrer uma exceção durante a execução da função `alguma_funcao`, o bloco `except` captura a exceção e lança uma nova exceção `MinhaExcecao`, encadeando a exceção original com a palavra-chave `from`.

Em resumo, o encadeamento de exceções em Python permite capturar uma exceção e lançar uma nova exceção com informações adicionais, mantendo o rastreamento de pilha da exceção original. Essa técnica é útil para identificar e corrigir erros em bibliotecas e módulos que não possuem informações suficientes para lidar com exceções específicas.

9.5 Exceções em Módulos e Pacotes

Em Python, é possível tratar exceções em módulos e pacotes utilizando a palavra-chave `try/except` em conjunto com as funções `import` e `from`. Essa técnica é útil para lidar com exceções específicas que na importação/uso de módulos e pacotes utilizados em um programa. Para lidar com exceções em um módulo ou pacote, é possível utilizar a palavra-chave `try/except` de acordo com o ilustrado no exemplo 9.11.

```
try:
    import meu_modulo
except ModuleNotFoundError:
    print("Erro: módulo não encontrado")
```

Exemplo de Código 9.11: Tratando exceções em importação de módulo.

Neste exemplo, o bloco `try` tenta importar o módulo `meu_modulo`. Se o módulo não existir, o bloco `except` captura a exceção `ModuleNotFoundError` e exibe uma mensagem de erro. Também é possível utilizar a palavra-chave `try/except` com a função `from`, que permite importar apenas uma parte de um módulo ou pacote em Python. O exemplo 9.12 ilustra essa situação.

```
try:
    from meu_pacote import minha_funcao
except ImportError:
    print("Erro: função não encontrada")
```

Exemplo de Código 9.12: Tratando exceções na importação de parte de um módulo.

Neste exemplo, o bloco `try` importa apenas a função `minha_funcao` do pacote `meu_pacote`. Se a função não existir, o bloco `except` captura a exceção `ImportError` e exibe uma mensagem de erro. Outra técnica para lidar com exceções em módulos e pacotes é criar uma função empacotadora (*wrapper*), que envolve a chamada de uma função do módulo em um bloco `try/except`. O exemplo 9.13 ilustra essa situação.

```
import meu_modulo
def minha_funcao():
    try:
        meu_modulo.funcao_do_modulo()
    except Exception as e:
        print("Erro: " + str(e))
```

Exemplo de Código 9.13: Tratando exceções usando uma função empacotadora.

Neste exemplo 9.13, a função `minha_funcao` envolve a chamada da função `funcao_do_modulo` do módulo `meu_modulo` em um bloco `try/except`. Se ocorrer uma exceção durante a execução da função, o bloco `except` captura a exceção e exibe uma mensagem de erro.

Em resumo, lidar com exceções em módulos e pacotes em Python requer o uso da palavra-chave `try/except` em conjunto com as funções `import` e `from`. Também é possível criar uma função *wrapper* para envolver a chamada de uma função do módulo em um bloco `try/except`. Lidar com exceções em módulos e pacotes pode evitar falhas no código e garantir a robustez do programa como um todo.

9.6 Exercícios Propostos

Essa série de exercícios envolve os conceitos abordados neste capítulo. É importante que, além de resolver o problema, você tente fazer uso de todos os recursos estudados ao longo do capítulo.

Exercício 9.1 Explique o conceito de exceções em Python. ■

Exercício 9.2 Por que é importante tratar exceções em um programa? ■

Exercício 9.3 Dê um exemplo de exceção comum em Python. ■

Exercício 9.4 Explique a sintaxe do bloco `try/except` em Python. ■

Exercício 9.5 Como capturar todas as exceções em um bloco `try/except` usando apenas um bloco `except`? ■

Exercício 9.6 Dê um exemplo de um programa em que o bloco `try/except` é indispensável para a garantia de robustez do programa. ■

Exercício 9.7 Explique a função do bloco `finally` em Python. ■

Exercício 9.8 Como o bloco `finally` pode ser utilizado em conjunto com o bloco `try/except`? ■

Exercício 9.9 Dê um exemplo de um programa em o bloco `finally` seria indispensável para garantir a execução de código visando proteger algum recurso. ■

Exercício 9.10 Como criar uma exceção personalizada em Python? ■

Exercício 9.11 Por que é útil criar exceções personalizadas em um programa? ■

Exercício 9.12 Dê um exemplo de uma exceção personalizada em Python. ■

Exercício 9.13 Explique o conceito de encadeamento de exceções em Python. ■

Exercício 9.14 Como encadear exceções em Python? ■

Exercício 9.15 Dê um exemplo de um programa que utilize o encadeamento de exceções. ■

Exercício 9.16 Explique o conceito de exceções assíncronas em Python. ■

Exercício 9.17 Como lidar com exceções assíncronas em um programa? ■

Exercício 9.18 Por que é importante lidar com exceções em módulos e pacotes em Python? ■

Exercício 9.19 Como lidar com exceções em módulos e pacotes em Python? ■

Exercício 9.20 Crie um programa que solicite ao usuário que digite um número inteiro. O programa deve exibir a raiz quadrada desse número. Se o usuário digitar um número negativo, o programa deve exibir uma mensagem de erro apropriada e solicitar que o usuário tente novamente. ■

Exercício 9.21 Crie um programa que leia um arquivo de texto e exiba o seu conteúdo na tela. Se o arquivo não existir, o programa deve exibir uma mensagem de erro apropriada. ■

Exercício 9.22 Crie uma função que receba uma lista de números inteiros e calcule a média dos valores. Se a lista estiver vazia, a função deve lançar uma exceção personalizada com uma mensagem de erro apropriada. ■

Exercício 9.23 Crie um programa que leia uma lista de números inteiros a partir de um arquivo texto e exiba a soma dos valores na tela. Se ocorrer um erro durante a leitura do arquivo, o programa deve exibir uma mensagem de erro apropriada e encerrar a execução. ■

Exercício 9.24 Crie um programa que solicite ao usuário que digite um nome de arquivo. O programa deve tentar abrir o arquivo e exibir seu conteúdo na tela. Se ocorrer um erro durante a abertura do arquivo, o programa deve exibir uma mensagem de erro apropriada e solicitar que o usuário tente novamente. ■

Exercício 9.25 Crie um programa que utilize o encadeamento de exceções para lidar com um erro que pode ocorrer durante a execução de uma função específica. A primeira exceção deve ser uma exceção personalizada com uma mensagem de erro apropriada, e a segunda exceção deve ser uma exceção genérica que capture qualquer erro não tratado pela exceção personalizada. ■

Exercício 9.26 Crie um programa que utilize o tratamento de exceções em um módulo ou pacote Python. O programa deve importar uma função de um módulo ou pacote e exibir o resultado da função na tela. Se ocorrer um erro durante a execução da função, o programa deve exibir uma mensagem de erro apropriada. ■

9.7 Considerações Sobre o Capítulo

Este capítulo mostrou como usar técnicas de tratamento de exceção para se criar programas mais robustos. O capítulo abordou conceitos importantes sobre tratamento de exceções, proteção de recursos, criação de exceções personalizadas, encadeamento de exceções e exceções em módulos e pacotes. Ao fim, foram propostos diversos exercícios para colocar em prática o que o capítulo abordou. No capítulo seguinte, você aprenderá manipular bancos de dados em Python.