FUNDAMENTOS DA CIÊNCIA DOS DADOS E LÓGICA DE PROGRAMAÇÃO

sagah

Tipos de dados básicos e estruturados em Python

Rafael Albuquerque Pinto

OBJETIVOS DE APRENDIZAGEM

- Reconhecer os tipos de dados básicos em Python.
- Entender as listas em Python e suas operações.
- Comparar as estruturas `list` e `tuple` em Python.

Introdução

A manipulação de dados é uma das principais tarefas na programação em Python. Para isso, é fundamental reconhecer os tipos de dados básicos disponíveis na linguagem, compreender as listas e suas operações e entender as diferenças entre as estruturas de listas e tuplas.

Em Python, existem diversos tipos de dados básicos que desempenham um papel fundamental na programação, como números inteiros, números de ponto flutuante, booleanos, números complexos e cadeias de caracteres. Cada um deles tem características específicas e é utilizado para representar diferentes tipos de informações. Compreender como esses tipos de dados são representados e utilizados é essencial para criar algoritmos eficientes e realizar manipulações de informações. Por exemplo, ao realizar operações matemáticas,

é importante utilizar o tipo de dado adequado para obter resultados precisos. Além disso, ao manipular textos ou realizar comparações lógicas, é necessário utilizar os tipos de dados apropriados.

Além dos tipos de dados básicos, o Python oferece uma poderosa estrutura de dados chamada de lista. As listas são utilizadas para armazenar coleções de elementos, permitindo que você agrupe diferentes valores em uma única variável. Essa flexibilidade torna as listas extremamente úteis em diversas situações, desde o armazenamento de números e textos até a manipulação de conjuntos de dados mais complexos. Outra estrutura de dados em Python é a tupla. As tuplas são semelhantes às listas, pois também permitem armazenar múltiplos elementos. No entanto, a principal diferença entre as duas está na mutabilidade. Enquanto as listas são mutáveis (ou seja, é possível modificar seus elementos após sua criação), as tuplas são imutáveis.

Neste capítulo, você vai conhecer os tipos de dados básicos em Python. Além disso, vai estudar operações com listas e ver as diferenças entre listas e tuplas.

Tipos de dados básicos em Python

Em Python, assim como em qualquer linguagem de programação, os tipos de dados desempenham um papel fundamental na manipulação e no armazenamento de informações (SEBESTA, 2018). Eles são os blocos de construção essenciais para a criação de algoritmos e a resolução de problemas. Os principais tipos de dados em Python incluem os números inteiros, os números de ponto flutuante, os booleanos, os números complexos e as cadeias de caracteres. Cada um desses tipos tem características únicas e é amplamente utilizado em algoritmos estruturados (AMARAL, 2018; MENEZES, 2019).

A seguir, você vai ver os principais tipos de dados em Python, além de exemplos de uso em variáveis em algoritmos estruturados.

Números inteiros

Os números inteiros representam valores numéricos sem casas decimais. Eles podem ser positivos, negativos ou zero. Para armazenar um número inteiro em uma variável, basta atribuir um valor a ela. Veja o exemplo:

```
# Declaração de variável com um número inteiro
idade = 25

# Operações matemáticas com números inteiros
soma = idade + 10
subtracao = idade - 5
multiplicacao = idade * 2
divisao = idade / 3

# Saída de dados
print("Idade:", idade)
print("Soma:", soma)
print("Subtração:", subtracao)
print("Multiplicação:", multiplicacao)
print("Divisão:", divisao)
```

Nesse código, temos a declaração de uma variável denominada `idade` com o valor inteiro 25. Em seguida, realizamos algumas operações matemáticas com essa variável, como soma, subtração, multiplicação e divisão. Por fim, utilizamos o comando `print` para exibir os valores da idade e dos resultados das operações.

Números de ponto flutuante

Os números de ponto flutuante são utilizados para representar valores numéricos com casas decimais. Eles podem ser usados para representar números reais. Para atribuir um número de ponto flutuante a uma variável, utilize a notação com ponto decimal. Veja o exemplo:

```
# Declaração de variável com um número de ponto flutuante
altura = 1.75

# Operações matemáticas com números de ponto flutuante
metade _ da _ altura = altura / 2
quadrado _ da _ altura = altura ** 2
```

```
# Saída de dados
print("Altura:", altura)
print("Metade da altura:", metade_da_altura)
print("Altura ao quadrado:", quadrado_da_altura)
```

Nesse código, a variável `altura` é declarada com o valor de 1.75, que é um número de ponto flutuante. Em seguida, realizamos algumas operações matemáticas, como dividir a altura por 2 e elevar a altura ao quadrado. Utilizamos o comando `print` para exibir os valores da altura e dos resultados das operações.

Booleanos

Os valores booleanos representam a ideia de verdadeiro ou falso. Em Python, os booleanos são escritos como `True` (verdadeiro) ou `False` (falso). Eles são frequentemente utilizados para controle de fluxo em estruturas condicionais. Veja o exemplo:

```
# Declaração de variável booleana
temperatura_alta = True

# Estruturas condicionais com valores booleanos
if temperatura_alta:
    print("A temperatura está alta.")
else:
    print("A temperatura está normal.")
```

Nesse código, a variável booleana `temperatura_alta` é declarada com o valor `True`, indicando que a temperatura está alta. Em seguida, utilizamos uma estrutura condicional (`if`) para verificar se a temperatura está alta. Se a condição for verdadeira, o programa vai imprimir "A temperatura está alta". Caso contrário, ele vai imprimir "A temperatura está normal".

Números complexos

Os números complexos são utilizados para representar valores com partes reais e imaginárias. Eles são escritos na forma "parte real + parte imaginária j", onde *j* é a unidade imaginária. Para atribuir um número complexo a uma variável, utilize a seguinte sintaxe:

```
# Declaração de variável com um número complexo
numero_complexo = 2 + 3j

# Acesso às partes real e imaginária
parte_real = numero_complexo.real
parte_imaginaria = numero_complexo.imag

# Saída de dados
print("Número complexo:", numero_complexo)
print("Parte real:", parte_real)
print("Parte imaginária:", parte_imaginaria)
```

Nesse código, a variável `numero_complexo` é declarada com o valor 2 + 3j, representando um número complexo com parte real igual a 2 e parte imaginária igual a 3. Em seguida, utilizamos os atributos `.real` e `.imag` para acessar a parte real e a parte imaginária do número complexo, respectivamente. Utilizamos o comando `print` para exibir o número complexo, a parte real e a parte imaginária.

Cadeias de caracteres

As cadeias de caracteres (*strings*) são sequências de caracteres alfanuméricos. Elas são utilizadas para representar texto em Python e são delimitadas por aspas simples ou duplas. Veja o exemplo:

```
# Declaração de variável com uma cadeia de caracteres
nome = "João"

# Concatenação de cadeias de caracteres
saudacao = "Olá, " + nome + "!"

# Saída de dados
print("Nome:", nome)
print("Saudação:", saudacao)
```

Nesse código, a variável `nome` é declarada com o valor "João", que é uma cadeia de caracteres (*string*). Em seguida, realizamos a concatenação de cadeias de caracteres para formar a variável `saudacao`, que contém a mensagem "Olá, João!". Utilizamos o comando `print` para exibir o valor da variável `nome` e o da variável `saudacao`.

Agora que você conhece os tipos de dados básicos em Python, pode utilizá-los para criar algoritmos estruturados. Por exemplo, você pode realizar operações matemáticas com números inteiros e de ponto flutuante, utilizar valores booleanos em estruturas condicionais e armazenar e manipular texto com cadeias de caracteres:

```
# Exemplo de uso de tipos de dados em um algoritmo estruturado

# Números inteiros
idade = 25
anos_futuros = idade + 10
anos_passados = idade - 5

# Números de ponto flutuante
altura = 1.75
peso = 68.5
imc = peso / (altura ** 2)

# Booleanos
temperatura_alta = True
if temperatura_alta:
    print("A temperatura está alta.")
else:
    print("A temperatura está normal.")

# Números complexos
numero_complexo = 2 + 3j
parte_real = numero_complexo.real
parte_imaginaria = numero_complexo.imag

# Cadeias de caracteres
nome = "João"
sobrenome = "Silva"
nome_completo = nome + " " + sobrenome
```

```
# Saída de dados
print("Idade daqui a 10 anos:", anos_futuros)
print("Idade há 5 anos atrás:", anos_passados)
print("IMC:", imc)
print("Parte real do número complexo:", parte_real)
print("Parte imaginária do número complexo:",
parte_imaginaria)
print("Nome completo:", nome_completo)
```

Nesse exemplo, utilizamos variáveis dos diferentes tipos de dados básicos em algoritmos estruturados em Python. Realizamos operações matemáticas com números inteiros e de ponto flutuante, utilizamos valores booleanos em uma estrutura condicional, trabalhamos com números complexos e manipulamos cadeias de caracteres.

Agora que você conhece os tipos de dados básicos em Python, pode avançar para um tipo de estrutura de dados mais complexa: as listas. As listas são extremamente úteis para armazenar coleções de elementos relacionados. Elas nos permitem manipular conjuntos de valores de maneira eficiente e flexível.

Listas e suas operações

Em Python, a lista (*list*) é um tipo de dado versátil e poderoso que nos permite armazenar e manipular coleções de elementos de maneira flexível (LUTZ; ASCHER, 2007). As listas são mutáveis, o que significa que podemos adicionar, modificar e remover elementos após a criação da lista. A seguir, vamos explorar as principais operações e funções relacionadas às listas em Python (MENEZES, 2019).

Criação de uma lista

Para criar uma lista, utilizamos colchetes [] e separamos os elementos por vírgulas. Os elementos podem ser de qualquer tipo de dado e até mesmo outras listas. Podemos criar uma lista vazia ou inicializá-la com elementos. Veja os exemplos:

```
lista_vazia = []
lista_numeros = [1, 2, 3, 4, 5]
lista_strings = ["maçã", "banana", "laranja"]
```

Nesse código, três listas são criadas. A primeira lista, `lista _ vazia`, é inicializada sem elemento algum, ou seja, é uma lista vazia. A segunda lista, `lista _ numeros`, contém os números de 1 a 5. A terceira lista, `lista _ strings`, contém *strings* que representam frutas.

Acesso aos elementos da lista

Podemos acessar elementos individuais da lista utilizando índices. Os índices em Python começam em 0, ou seja, o primeiro elemento está no índice 0, o segundo no índice 1 e assim por diante. Veja um exemplo:

```
lista = ["maçã", "banana", "laranja",”manga”]
primeiro _ elemento = lista[0] # "maçã"
print(primeiro _ elemento)
```

Nesse código, a lista contendo frutas é criada. Em seguida, o primeiro elemento da lista é acessado utilizando o índice `[0]`. Lembre-se de que os índices das listas em Python começam em 0, então o primeiro elemento tem índice 0.

Além de realizar acessos individuais aos elementos, é possível acessar um subconjunto de elementos de uma lista utilizando a notação de fatiamento (AMARAL, 2018). O formato geral é `lista[inicio:fim]`, onde `inicio` é o índice do primeiro elemento a ser incluído e `fim` é o índice do elemento após o último elemento a ser incluído. Veja um exemplo:

```
lista = ["maçã", "banana", "laranja", "manga"]
print(lista[1:3])
```

Nesse código, a lista contendo frutas é criada. Em seguida, acessamos o subconjunto da lista que começa no índice 1 e termina no índice 3, imprimindo o subconjunto `["banana", "laranja", "manga"]`.

Inserção de elementos

Podemos adicionar elementos a uma lista utilizando o método `append()` ou o método `insert()`. O método `append()` adiciona um elemento ao final da lista, enquanto o método `insert()` permite especificar a posição de inserção. Veja um exemplo:

```
lista = ["maçã", "banana", "laranja"]
lista.append("morango")
lista.insert(1, "abacaxi")
```

Nesse código, uma lista inicial é criada. Em seguida, o método `append()` é utilizado para adicionar o elemento `morango` ao final da lista. O método `insert()` é utilizado para inserir o elemento `abacaxi` na posição de índice 1 da lista, deslocando os elementos existentes.

Remoção de elementos

Podemos remover elementos de uma lista utilizando os métodos `pop()`, `remove()` ou utilizando a instrução `del`. O método `pop()` remove e retorna o último elemento da lista ou um elemento em uma posição específica. O método `remove()` remove a primeira ocorrência de um elemento específico. A instrução `del` remove um elemento em uma posição específica sem retornar seu valor. Veja o exemplo:

```
lista = ["maçã", "banana", "laranja"]
ultimo_elemento = lista.pop() # remove e retorna "laranja"
lista.remove("banana")
del lista[0]
```

Nesse código, uma lista inicial é criada. Em seguida, o método `pop()` é utilizado sem um índice específico, o que remove e retorna o último elemento da lista, ou seja, o elemento `laranja` é removido da lista e atribuído à variável `ultimo_elemento`. Depois, o método `remove()` é utilizado para remover a primeira ocorrência do elemento `banana` da lista. Por fim, a instrução `del` é usada para remover o elemento no índice 0 da lista, ou seja, o elemento `maçã` é removido.

Funções úteis para listas

Python também fornece algumas funções integradas que podem ser úteis ao trabalhar com listas; veja a seguir (PYTHON, 2023).

- `len(lista)`: retorna o número de elementos na lista.
- `max(lista)`: retorna o elemento máximo da lista.
- `min(lista)`: retorna o elemento mínimo da lista.
- `sum(lista)`: retorna a soma dos elementos da lista.

As listas em Python são muito flexíveis e permitem armazenar diferentes tipos de dados, inclusive outras listas. Com essas operações e funções, é possível manipular listas de forma eficiente e realizar diversas tarefas, desde cálculos estatísticos até a organização de dados. Veja o exemplo:

```
lista = [5, 2, 8, 1, 9]
tamanho = len(lista) # 5
maior_elemento = max(lista) # 9
soma = sum(lista) # 25
```

Nesse código, uma lista de números é criada. Em seguida, a função `len()` é utilizada para obter o tamanho da lista, ou seja, a quantidade de elementos presentes. A função `max()` é usada para encontrar o maior elemento da lista, e a função `sum()` é utilizada para calcular a soma de todos os elementos da lista.

Saiba mais

A documentação oficial do Python oferece uma descrição detalhada das listas e de todas as operações que podem ser realizadas com elas. No *site* oficial do Python, você vai encontrar informações sobre métodos, funções embutidas, iteração, fatiamento e muito mais.

Agora que você conhece as listas em Python e suas operações, pode explorar outra estrutura de dados semelhante: as tuplas. As tuplas também são coleções de elementos, mas diferem das listas em alguns aspectos fundamentais.

Comparação entre listas e tuplas

Em Python, tanto as listas quanto as tuplas são estruturas de dados que agrupam conjuntos de elementos. Ambas podem ser indexadas, e seus elementos podem ser acessados usando a mesma sintaxe. No entanto, existem diferenças cruciais entre elas. A principal diferença entre listas e tuplas é a mutabilidade. As listas são mutáveis, o que significa que podemos adicionar, remover e modificar elementos após sua criação. Em contrapartida, as tuplas são imutáveis, ou seja, uma vez criadas, não podem ser modificadas. Isso significa que os elementos de uma tupla são fixos e não podem ser alterados individualmente. A seguir, veja algumas diferenças importantes entre listas e tuplas (AMARAL, 2018; MENEZES, 2019).

Mutabilidade

As listas são mutáveis, o que permite adicionar, remover e modificar elementos. Por exemplo, podemos alterar o valor de um elemento específico em uma lista, adicionar um novo elemento ao final ou remover um elemento existente. Já as tuplas são imutáveis e não permitem tais operações. Veja o exemplo:

```
# Lista (mutável)
lista = [1, 2, 3]
lista[0] = 4
lista.append(5)
lista.remove(2)
print(lista) # Output: [4, 3, 5]

# Tupla (imutável)
tupla = (1, 2, 3)
tupla[0] = 4 # Erro: As tuplas são imutáveis e não podem
ser modificadas
```

Nesse exemplo, uma lista e uma tupla são inicializadas com os mesmos elementos. Na lista, podemos modificar o valor do primeiro elemento, adicionar um novo elemento no final e remover o elemento com valor 2. Por outro lado, tentar modificar um elemento da tupla resulta em um erro, pois as tuplas são imutáveis.

Desempenho

Devido à sua imutabilidade, as tuplas tendem a ser ligeiramente mais eficientes em termos de desempenho e uso de memória do que as listas. Isso ocorre porque as tuplas ocupam menos espaço em memória e não exigem operações adicionais para suportar a mutabilidade. Em situações em que não precisamos modificar a coleção de elementos, o uso de tuplas pode trazer benefícios de desempenho. Por exemplo:

```
import timeit

# Medição de tempo para criar uma lista com 10 milhões de
elementos
tempo_lista = timeit.timeit("lista = [i for i in
range(10000000)]", number=1)

# Medição de tempo para criar uma tupla com 10 milhões de
elementos
tempo_tupla = timeit.timeit("tupla = tuple(i for i in
range(10000000))", number=1)

print("Tempo para criar uma lista:", tempo_lista)
print("Tempo para criar uma tupla:", tempo_tupla)
```

Nesse exemplo, usamos a biblioteca Timeit para medir o tempo necessário para criar uma lista e uma tupla com 10 milhões de elementos. Geralmente, as tuplas são criadas mais rapidamente do que as listas, pois não exigem a sobrecarga de memória associada à mutabilidade.

Uso adequado

As listas são mais adequadas quando precisamos de uma estrutura de dados que possa ser modificada ao longo do tempo. Elas são ideais para situações em que a ordem e a mutabilidade são importantes, como armazenar dados que serão atualizados, reordenados ou modificados durante a execução do programa. As tuplas são mais adequadas quando precisamos de uma coleção de elementos imutáveis, como coordenadas geográficas fixas, informações constantes que não devem ser alteradas ou até mesmo para uso em chaves de dicionários, pois as tuplas são *hashable* (podem ser usadas como chaves de dicionário). Por exemplo:

```
# Lista de nomes de alunos (mutável)
alunos = ["João", "Maria", "Pedro"]

# Tupla de informações sobre uma pessoa (imutável)
pessoa = ("João", 25, "Masculino", "São Paulo")
```

Nesse exemplo, uma lista é usada para armazenar nomes de alunos, pois a lista pode ser modificada à medida que novos alunos são adicionados ou removidos. Por outro lado, uma tupla é usada para armazenar informações sobre uma pessoa, como nome, idade, gênero e cidade. Essas informações são consideradas constantes e não precisam ser modificadas, tornando a tupla uma escolha apropriada.

Em resumo, as listas e as tuplas são estruturas de dados semelhantes em Python, mas têm diferenças fundamentais em termos de mutabilidade e desempenho. As listas oferecem flexibilidade e capacidade de modificação, enquanto as tuplas oferecem imutabilidade e potencialmente um melhor desempenho.

Fique atento

Lembre-se de que as listas são mutáveis, o que significa que uma alteração feita em uma lista pode afetar outras variáveis que fazem referência a ela. Ao atribuir uma lista a outra variável, ambas as variáveis apontam para a mesma lista na memória. Portanto, modificações em uma das variáveis também serão refletidas na outra. Se você precisar criar uma cópia independente da lista, utilize o método `copy()` ou a função `list()` para evitar efeitos colaterais indesejados (MENEZES, 2019).

Neste capítulo, você conheceu os tipos de dados básicos em Python, como números inteiros, números de ponto flutuante, booleanos, números complexos e cadeias de caracteres. Em seguida, estudou as listas em Python, uma estrutura de dados versátil que permite armazenar coleções de elementos e realizar operações como criação, acesso, inserção e remoção. Por fim, viu diferenças entre as listas e as tuplas, como sua mutabilidade, seu desempenho e seu uso adequado.

Referências

AMARAL, F. *Introdução a ciência de dados*. Rio de Janeiro: Alta Books, 2018. *E-book*.

LUTZ, M.; ASCHER, D. *Aprendendo Python*. 2. ed. Porto Alegre: Bookman, 2007.

MENEZES, N. N C. *Introdução à programação com Python*: algoritmos e lógica de programação para iniciantes. 3. ed. Rio de Janeiro: Novatec, 2019.

PYTHON. Tipos embutidos. *Python Software Foundation*, 2023. Disponível em: https://docs.python.org/pt-br/3/library/stdtypes.html. Acesso em: 18 jul. 2023.

SEBESTA, R. W. *Conceitos de linguagens de programação*. 11. ed. Porto Alegre: Bookman, 2018.

Leitura recomendada

DOWNEY, A. B. *Pense em Python*: pense como um cientista da computação. Rio de Janeiro: Novatec, 2016.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

sagah