CloudCommand

Cyber Security

CloudCommand GmbH chr.schumacher@gmx.tm

CloudCommand

Grundlagen der Programmierung

CloudCommand GmbH chr.schumacher@gmx.tm

Listen und Arrays

Überblick

Bedeutung von Datenstrukturen in der Programmierung

- Grundbausteine für die Organisation und Speicherung von Daten
- Ermöglichen effiziente Datenmanipulation und -zugriff

Beispiele:

- Suchen in einer Liste, Sortieren von Daten, Speichern von Schlüssel-Wert-Paaren in einem Wörterbuch

Datenstrukturtypen in der Programmierung

Listen:

- Dynamisch, können Daten beliebigen Typs enthalten

Tupel:

- Unveränderlich, können Daten beliebigen Typs enthalten

Sets:

- Ungeordnete Sammlungen ohne Duplikate

Wörterbücher (Dictionaries):

- Schlüssel-Wert-Paare, schneller Zugriff über Schlüssel

Arrays (mittels Modulen wie array oder numpy):

- Speichern Daten eines Typs, effizient für numerische Daten

Datenstrukturtypen in der Programmierung

Gemeinsamkeiten:

- Beide können eine Sammlung von Elementen speichern
- Beide unterstützen Indizierung und Slicing für den Zugriff auf Elemente

Datenstrukturtypen in der Programmierung

Unterschiede:

- **Listen:**
  - Können Daten unterschiedlichen Typs enthalten
  - Flexibler, da sie dynamisch sind und ihre Größe ändern können

- **Arrays:**
  - Speichern nur Elemente eines einzigen Typs, effizienter für große Datensätze
  - Besser geeignet für mathematische Operationen und wissenschaftliche Berechnungen
  - Benötigen Import eines Moduls wie array oder numpy

Listen in Python

Grundlagen zur Erstellung von Listen

- **Syntax und Listeninitialisierung:**
  - Verwenden von eckigen Klammern [], um eine Liste zu erstellen.

- **Beispiel:**

```
meine_liste = [1, 2, 3, 4]
```

Listen in Python

Grundlagen zur Erstellung von Listen

- **Heterogene Daten in Listen:**
  - Listen können Elemente unterschiedlicher Typen enthalten
  - **Beispiel:**

```
gemischte_liste = [1, "Hallo", 3.14, True]
```

Listen in Python

Zugriff auf Listenelemente

- **Indizierung und Slicing:**
  - Zugriff auf ein Element durch Angabe seines Indexes:

```
meine_liste[0]
```

  - ...oder auf Teile der Liste durch Slicing:

```
meine_liste[1:3]
```

Listen in Python

Zugriff auf Listenelemente

- **Negative Indizes:**
  - Zugriff auf Elemente von hinten, z.B. -1 für das letzte Element:

```
meine_liste[-1]
```

Listen in Python

Hinzufügen und Entfernen von Elementen

- **append()** fügt ein Element am Ende hinzu, z. B.:

```
meine_liste.append(5)
```

- **extend()** erweitert die Liste um die Liste eines iterierbaren Objekts, z.B.:

```
meine_liste.extend([6, 7])
```

Listen in Python

Hinzufügen und Entfernen von Elementen

- **pop()** entfernt und gibt ein Element zurück, standardmäßig das letzte, z.B.:

```
meine_liste.pop()
```

- **remove()** entfernt das erste Vorkommen eines Elements, z.B.:

```
meine_liste.remove(2)
```

Listen in Python

```python
# Erstelle eine Liste
liste = [1, 2, 3, 4, 5]

# append(): Füge ein Element am Ende der Liste hinzu
liste.append(6)
print("Liste nach dem Anhängen von 6 mit append():", liste)

# extend(): Füge Elemente einer anderen Liste am Ende hinzu
erweiterung = [7, 8, 9]
liste.extend(erweiterung)
print("Liste nach dem Anhängen von [7, 8, 9] mit extend():", liste)

# remove(): Entferne das erste Vorkommen eines Elements aus der Liste
liste.remove(3)
print("Liste nach Entfernen der ersten 3:", liste)
```

Listen in Python

Listen verketten und wiederholen

- Verwendung von **+** zum Verketten und ***** zum Wiederholen, z.B.:

```
neue_liste = meine_liste + [8, 9]
wiederholte_liste = meine_liste * 2
```

Listen sortieren und umkehren

- **meine_liste.sort()** sortiert die Liste in aufsteigender Reihenfolge.
- **meine_liste.reverse()** kehrt die Reihenfolge der Elemente um.

Listen in Python

Listenkomprehensionen

- Ermöglicht die effiziente Erstellung und Manipulation von Listen in einer einzigen Zeile, z.B.:

```
quadrate = [x**2 for x in range(10)]
```

Listen in Python

Verschachtelte Listen und Matrixoperationen

- **Erstellen und Zugreifen auf mehrdimensionale Listen:**
  - Listen innerhalb von Listen, z.B.:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  - Zugriff via:

```
matrix[0][1]
```

Code-Beispiel: Listen 1

```python
# Erstelle eine mehrdimensionale Liste
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print("Element in Zeile 2, Spalte 1:", matrix[1][0]) # Zugriff

matrix[2][2] = 10 # Ändern eines Elements der mehrdimensionalen Liste
print("Matrix nach Änderung des Elements in Zeile 3, Spalte 3:", matrix)

print("Alle Elemente der Matrix:") # Iteration durch die Liste
for zeile in matrix:
    for element in zeile:
        print(element, end=" ")
    print()
```

Code-Beispiel: Listen 2

```
# Definition unserer kleinen Beispieldatenbank:
personenliste = [
[1, "Max", "Mustermann", "max.mustermann@example.com"],
[2, "Anna", "Müller", "anna.mueller@example.com"],
[3, "Tom", "Schmidt", "tom.schmidt@example.com"],
[4, "Sarah", "Fischer", "sarah.fischer@example.com"],
[5, "David", "Weber", "david.weber@example.com"],
[6, "Lena", "Schneider", "lena.schneider@example.com"],
[7, "Markus", "Schulz", "markus.schulz@example.com"],
[8, "Julia", "Bauer", "julia.bauer@example.com"],
[9, "Paul", "Wagner", "paul.wagner@example.com"],
[10, "Lisa", "Hoffmann", "lisa.hoffmann@example.com"]
]
```

Code-Beispiel: Listen 2

```
# Den Benutzer fragen auf welche Datensatznummer er zugreifen möchte: Werte zwischen 1 und 10 sind erlaubt
dsnr = int(input("\n\nAuf welche Datensatznummer wollen Sie zugreifen ? (1-10) "))
dsnr = dsnr - 1

# Zugriff auf einen Eintrag aus der Liste
eine_person = personenliste[dsnr]
index = eine_person[0]
vorname = eine_person[1]
nachname = eine_person[2]
email = eine_person[3]
```

Code-Beispiel: Listen 2

```
# Ausgabe der Informationen über die gewählte Person
print(f"\n-- Informationen zu Datensatz #{dsnr} --")
print("Index:    ", index)
print("Vorname:  ", vorname)
print("Nachname: ", nachname)
print("E-Mail:   ", email)
print("---------- Datensatz Ende  ----------")
```

Arrays in Python

Performance-Vergleich mit Listen:

- Arrays sind speicher- und performanceeffizienter für die Speicherung großer Mengen von Daten eines einzigen Typs. Sie sind besonders nützlich für numerische Datenoperationen.

Arrays in Python

Verwendung des array Moduls:

- Python bietet das Modul **array**, das die Erstellung von Arrays unterstützt, welche effizienter im Umgang mit großen Datenmengen eines einzigen Typs sind.

Array-Erstellung und unterstützte Datentypen:

- Import des Moduls mittels import array, danach kann ein Array mit einem Typcode (der den Datentyp spezifiziert) und einer initialen Werteliste erstellt werden, z.B.

```
mein_array = array.array('i', [1, 2, 3, 4])
```

Arrays in Python

Zugriff auf Array-Elemente und Slicing:

- Ähnlich wie bei Listen, z.B. **mein_array[0]** für den Zugriff auf das erste Element, **mein_array[1:3]**, für Slicing.

Hinzufügen und Entfernen von Elementen:

- Verwendung von Methoden wie **append()**, zum Hinzufügen, **pop()** zum Entfernen eines Elements am Ende und **remove()** zum Entfernen eines spezifizierten Elements.

Arrays in Python

Array-Methoden für Manipulationen:

- Neben **append()**, **pop()** und **remove()** unterstützen Arrays auch Methoden wie **insert()** zum Einfügen eines Elements an einer bestimmten Position.

Arrays in Python

Lesen und Schreiben von Arrays in Dateien:

- Arrays können effizient in Dateien geschrieben und aus Dateien gelesen werden, was sie nützlich für Datenverarbeitungsaufgaben macht. Beispielsweise kann ein Array von Zahlen direkt in eine binäre Datei geschrieben und aus dieser gelesen werden, was bei großen Datenmengen Zeit und Speicherplatz spart.

Arrays in Python

```python
import array
import tempfile

# Schreiben in eine temporäre Datei
with tempfile.NamedTemporaryFile() as temp:
    arr = array.array('i', [1, 2, 3, 4, 5])
    arr.tofile(temp.file)  # Schreibt das Array in eine Datei
    temp.file.seek(0)  # Zurück zum Anfang der Datei

    # Ein neues Array aus der Datei lesen
    read_arr = array.array('i')
    read_arr.fromfile(temp.file, len(arr))
    print(read_arr)  # Gibt das gelesene Array aus
```

NumPy Arrays

Warum NumPy für wissenschaftliches Rechnen?

- **Performance- und Speicherplatzvorteile:**
  - NumPy bietet optimierte Speicherung und Datenverarbeitung, was es deutlich schneller als native Python-Listen für numerische Operationen macht.

- **Umfangreiche wissenschaftliche Funktionen:**
  - Beinhaltet Funktionen für lineare Algebra, Fourier-Transformationen und weitere komplexe mathematische Operationen.

NumPy Arrays

Erstellen von NumPy Arrays:

- **Performance- und Speicherplatzvorteile:**
  - Verwendung von **numpy.array()** zum Erstellen von Arrays.

- **Beispiel:**

```
import numpy as np

arr_eindimensional = np.array([1, 2, 3])
arr_mehrdimensional = np.array([[1, 2, 3], [4, 5, 6]])
```

NumPy Arrays

Erstellen von NumPy Arrays:

- **Spezialfunktionen zur Array-Erstellung:**
  - NumPy bietet Funktionen wie
    np.zeros(),
    np.ones(),
    np.arange(), und
    np.linspace()
    für die schnelle Erstellung von spezialisierten Arrays.

NumPy Arrays

Elementweise Operationen, Broadcasting:

- Unterstützt Operationen über Arrays hinweg, ohne explizite Schleifen. Broadcasting erlaubt die Anwendung von Operationen auf Arrays unterschiedlicher Größen.

NumPy Arrays

Elementweise Operationen, Broadcasting:

- **Aggregationsfunktionen:**
  - Funktionen wie **np.sum()**, **np.mean()**, **np.std()** zur Berechnung von Summen, Mittelwerten, Standardabweichungen usw. über gesamte Arrays oder entlang spezifischer Achsen.

NumPy Arrays

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5]) # Erstelle ein NumPy-Array

summe = np.sum(arr) # Summe der Elemente im Array
print("Summe:", summe)

durchschnitt = np.mean(arr) # Durchschnitt der Elemente im Array
print("Durchschnitt:", durchschnitt)

std_abweichung = np.std(arr) # Standardabweichung der Elemente im Array
print("Standardabweichung:", std_abweichung)
```

NumPy Arrays

Indizierung, Slicing und Iteration über Arrays:

- **Komplexe Indizierungsmethoden:**
  - Unterstützt neben der Standard-Indizierung auch booleanische Indizierung, Fancy-Indizierung usw., um auf Elemente zuzugreifen oder Teilmengen zu extrahieren.

- **Slicing:**
  - Ähnlich zu Python-Listen, aber mit erweiterten Möglichkeiten, wie das gleichzeitige Slicing entlang mehrerer Dimensionen.

NumPy Arrays

```python
import numpy as np

# Erstelle ein 2D-Array (Matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Gleichzeitiges Slicing entlang mehrerer Dimensionen
# Ausschneiden eines Teils der Matrix
sliced = matrix[0:2, 1:3]  # Zeilen von 0 bis 1, Spalten von 1 bis 2
print("Ausgeschnittener Teil der Matrix:")
print(sliced)
```

NumPy Arrays

Anwendungsbeispiele von NumPy Arrays in der Datenanalyse:

- **Datenmanipulation:**
  - Anwendung von Aggregationsfunktionen und Transformationen auf Datensätze für die Analyse.

NumPy Arrays

Anwendungsbeispiele von NumPy Arrays in der Datenanalyse:

- **Maschinelles Lernen:**
  - Verwendung von NumPy für die Vorverarbeitung von Daten, Feature-Extraktion und die Manipulation von Datenstrukturen, die in maschinellen Lernalgorithmen verwendet werden.

NumPy Arrays

Anwendungsbeispiele von NumPy Arrays in der Datenanalyse:

- **Bildverarbeitung:**
  - NumPy Arrays können für die Speicherung und Manipulation von Bildern verwendet werden, einschließlich Bildfilterung und -transformation.

Vergleich und Anwendungsfälle

Wann sollte man Listen und wann Arrays verwenden?

- **Listen:**
  - Ideal für heterogene Datensammlungen, wo Elemente unterschiedlichen Typs gespeichert werden sollen oder wenn die Datenstruktur häufig geändert wird (Elemente hinzugefügt/entfernt). Listen sind flexibel und bieten eine breite Palette von eingebauten Methoden zur Datenmanipulation.
- **Beispiel:**
  - Eine Liste von Mitarbeitern, wobei jeder Mitarbeiter durch eine Kombination aus Namen (String), Alter (Integer) und Gehalt (Float) repräsentiert wird.

Vergleich und Anwendungsfälle

Wann sollte man Listen und wann Arrays verwenden?

- **Arrays (array Modul):**
  - Besser geeignet für homogene Datensammlungen großer Mengen, insbesondere wenn Speichereffizienz und Basisoperationen auf numerischen Daten wichtig sind. Arrays unterstützen Operationen, die Listen nicht bieten, wie z.B. effizientes Schreiben und Lesen in Dateien.
- **Beispiel:**
  - Ein Array von Temperaturmesswerten über einen bestimmten Zeitraum, alle als Floats gespeichert.

Vergleich und Anwendungsfälle

Wann sollte man Listen und wann Arrays verwenden?

- **NumPy Arrays:**
  - Die beste Wahl für komplexe numerische Berechnungen, große Datenmengen und multidimensionale Arrays. NumPy bietet umfangreiche mathematische und statistische Funktionen, effizientes Slicing und leistungsstarke Operationen für wissenschaftliches Rechnen.
- **Beispiel:**
  - Ein 2D-NumPy Array zur Repräsentation einer Bildmatrix für Bildverarbeitungsaufgaben oder ein mehrdimensionales Array für die Datenanalyse in maschinellem Lernen.

Vergleich und Anwendungsfälle

Praktische Anwendungsfälle für Listen und Arrays:

- **Listen:**
  - Verwaltung von Sammlungen, die Elemente unterschiedlicher Typen enthalten (z.B. Daten eines Formulars). wenn die Größe der Sammlung häufig geändert oder wenn Elemente oft in der Mitte der Sammlung eingefügt oder entfernt werden.

Vergleich und Anwendungsfälle

Praktische Anwendungsfälle für Listen und Arrays:

- **Arrays (array Modul):**
  - Verarbeitung und Speicherung von großen Mengen homogener Daten, insbesondere wenn diese Daten numerisch sind und Operationen wie Summierung oder Durchschnittsberechnung benötigt werden.

CloudCommand

CloudCommand GmbH chr.schumacher@gmx.tm