合肥大學
HEFEI UNIVERSITY

Programming with Python

17. Listen

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn · https://thomasweise.github.io

School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-09-07

Programming with Python

Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

1. Einleitung
2. Beispiele
3. Zusammenfassung

Einleitung

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.
- Wir haben auch gelernt, wie wir eine Variable verwenden können, um eine Instanz eines solchen Datentyps zu speichern.

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.
- Wir haben auch gelernt, wie wir eine Variable verwenden können, um eine Instanz eines solchen Datentyps zu speichern.
- In vielen Fällen wollen wir aber nicht nur ein einziges Objekt speichern.

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.
- Wir haben auch gelernt, wie wir eine Variable verwenden können, um eine Instanz eines solchen Datentyps zu speichern.
- In vielen Fällen wollen wir aber nicht nur ein einziges Objekt speichern.
- Oftmals wollen wir Kollektionen von Objekten speichern und bearbeiten[1–3].

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.
- Wir haben auch gelernt, wie wir eine Variable verwenden können, um eine Instanz eines solchen Datentyps zu speichern.
- In vielen Fällen wollen wir aber nicht nur ein einziges Objekt speichern.
- Oftmals wollen wir Kollektionen von Objekten speichern und bearbeiten[1–3].
- Python bietet uns vier Arten von Kollektionen: Listen (EN: *lists*), Tupel (EN: *tuples*), Mengen (EN: *sets*), und Dictionaries.

Einleitung

- Wir haben bereits einfache Datentypen wie Ganzzahlen und Boolesche Werte gelernt.
- Wir haben auch gelernt, wie wir eine Variable verwenden können, um eine Instanz eines solchen Datentyps zu speichern.
- In vielen Fällen wollen wir aber nicht nur ein einziges Objekt speichern.
- Oftmals wollen wir Kollektionen von Objekten speichern und bearbeiten[1–3].
- Python bietet uns vier Arten von Kollektionen: Listen (EN: *lists*), Tupel (EN: *tuples*), Mengen (EN: *sets*), und Dictionaries.
- Wir fangen mit Listen an.

Listen

- Listen sind *veränderbare* (EN: *mutable*) Sequenzen von Objekten.

Listen

- Listen sind *veränderbare* (EN: *mutable*) Sequenzen von Objekten.
- Wir können eine Listenvariable `my_list` bestehend aus den drei Strings `"ax"`, `"by"`, und `"cz"` erstellen, in dem wir schreiben `my_list = ["ax", "by", "cz"]`.

Listen

- Listen sind *veränderbare* (EN: *mutable*) Sequenzen von Objekten.
- Wir können eine Listenvariable `my_list` bestehend aus den drei Strings `"ax"`, `"by"`, und `"cz"` erstellen, in dem wir schreiben `my_list = ["ax", "by", "cz"]`.
- Auf die Elemente der Liste können wir genauso zugreifen wie auf die einzelnen Zeichen einer Zeichenkette, in dem wir sie mit eckigen Klammern indizieren[12]: `my_list[0]` gibt uns das erste Element der Liste zurück, nämlich `"ax"`.

Listen

- Listen sind *veränderbare* (EN: *mutable*) Sequenzen von Objekten.
- Wir können eine Listenvariable `my_list` bestehend aus den drei Strings `"ax"`, `"by"`, und `"cz"` erstellen, in dem wir schreiben `my_list = ["ax", "by", "cz"]`.
- Auf die Elemente der Liste können wir genauso zugreifen wie auf die einzelnen Zeichen einer Zeichenkette, in dem wir sie mit eckigen Klammern indizieren[12]: `my_list[0]` gibt uns das erste Element der Liste zurück, nämlich `"ax"`.
- Listen sind also veränderliche Sequenzen von Objekten, aber anstelle von nur Zeichen (wie Strings) können sie beliebige Objekte beinhalten.

Listen

- Listen sind *veränderbare* (EN: *mutable*) Sequenzen von Objekten.
- Wir können eine Listenvariable `my_list` bestehend aus den drei Strings `"ax"`, `"by"`, und `"cz"` erstellen, in dem wir schreiben `my_list = ["ax", "by", "cz"]`.
- Auf die Elemente der Liste können wir genauso zugreifen wie auf die einzelnen Zeichen einer Zeichenkette, in dem wir sie mit eckigen Klammern indizieren[12]: `my_list[0]` gibt uns das erste Element der Liste zurück, nämlich `"ax"`.
- Listen sind also veränderliche Sequenzen von Objekten, aber anstelle von nur Zeichen (wie Strings) können sie beliebige Objekte beinhalten.
- Los geht's.

Beispiele

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listenvariablen werden mit dem Type Hint `list[elementTyp]` annotiert, wobei `elementType` der Datentyp für die Elemente ist[6].

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listenvariablen werden mit dem Type Hint `list[elementTyp]` annotiert, wobei `elementType` der Datentyp für die Elemente ist[6].
- Listen können als Literale mit eckigen Klammern definiert werden.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listenvariablen werden mit dem Type Hint `list[elementTyp]` annotiert, wobei `elementType` der Datentyp für die Elemente ist[6].
- Listen können als Literale mit eckigen Klammern definiert werden.
- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listenvariablen werden mit dem Type Hint `list[elementTyp]` annotiert, wobei `elementType` der Datentyp für die Elemente ist[6].
- Listen können als Literale mit eckigen Klammern definiert werden.
- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listenvariablen werden mit dem Type Hint `list[elementTyp]` annotiert, wobei `elementType` der Datentyp für die Elemente ist[6].
- Listen können als Literale mit eckigen Klammern definiert werden.
- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listen können als Literale mit eckigen Klammern definiert werden.
- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.
- `l1.extend(l2)` hängt alle Elemente des Contains `l2` an die Liste `l1` an.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- Listen können als Literale mit eckigen Klammern definiert werden.
- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.
- `l1.extend(l2)` hängt alle Elemente des Contains `l2` an die Liste `l1` an.
- Indizieren erfolgt genau wie bei Strings.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.
- `l1.extend(l2)` hängt alle Elemente des Contains `l2` an die Liste `l1` an.
- Indizieren erfolgt genau wie bei Strings.
- `del lst[i]` löscht das Element an Index `i` aus der Liste `lst`.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.
- `l1.extend(l2)` hängt alle Elemente des Contains `l2` an die Liste `l1` an.
- Indizieren erfolgt genau wie bei Strings.
- `del lst[i]` löscht das Element an Index `i` aus der Liste `lst`.

```python
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create list.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

↓ python3 lists_1.py ↓

```
We got 3 fruits: ['apple', 'pear', 'orange']
There now are 4 fruits: ['apple', 'pear', 'orange', 'cherry']
The vegetables are: ['onion', 'potato', 'leek'].
Fruits and vegetables: ['apple', 'pear', 'orange', 'cherry', 'onion', '
    ↪ potato', 'leek']
len(food) = 7
food[0] = 'apple'
food[1] = 'pear'
food[2] = 'orange'
food[-1] = 'leek'
food[-2] = 'potato'
food[-3] = 'onion'
Food is now: ['apple', 'orange', 'cherry', 'onion', 'potato', 'leek'].
```

Type Hints, Listen erstellen, Elemente anhängen, verbinden, und indizieren

- `len(lst)` liefert die Länge = Anzahl der Elemente in der Liste `lst`.
- `lst.append(x)` hängt Element `x` an die Liste `lst` an.
- `[]` ist eine leere Liste.
- `l1.extend(l2)` hängt alle Elemente des Contains `l2` an die Liste `l1` an.
- Indizieren erfolgt genau wie bei Strings.
- `del lst[i]` löscht das Element an Index `i` aus der Liste `lst`.

```
We got 3 fruits: ['apple', 'pear', 'orange']
There now are 4 fruits: ['apple', 'pear', 'orange', 'cherry']
The vegetables are: ['onion', 'potato', 'leek'].
Fruits and vegetables: ['apple', 'pear', 'orange', 'cherry', 'onion', '
    ↪ potato', 'leek']
len(food) = 7
food[0] = 'apple'
food[1] = 'pear'
food[2] = 'orange'
food[-1] = 'leek'
food[-2] = 'potato'
food[-3] = 'onion'
Food is now: ['apple', 'orange', 'cherry', 'onion', 'potato', 'leek'].
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a in lst` ist `True`, wenn Element `a` in Liste `lst` auftaucht.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a in lst` ist `True`, wenn Element `a` in Liste `lst` auftaucht.
- `a not in lst` ist `True`, wenn Element `a` *nicht* in Liste `lst` auftaucht.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a in lst` ist `True`, wenn Element `a` in Liste `lst` auftaucht.
- `a not in lst` ist `True`, wenn Element `a` *nicht* in Liste `lst` auftaucht.
- `lst.insert(i, e)` fügt Element `e` an Index `i` in Liste `lst` ein.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a in lst` ist `True`, wenn Element `a` in Liste `lst` auftaucht.
- `a not in lst` ist `True`, wenn Element `a` *nicht* in Liste `lst` auftaucht.
- `lst.insert(i, e)` fügt Element `e` an Index `i` in Liste `lst` ein.
- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a in lst` ist `True`, wenn Element `a` in Liste `lst` auftaucht.
- `a not in lst` ist `True`, wenn Element `a` *nicht* in Liste `lst` auftaucht.
- `lst.insert(i, e)` fügt Element `e` an Index `i` in Liste `lst` ein.
- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `a not in lst` ist `True`, wenn Element `a` *nicht* in Liste `lst` auftaucht.
- `lst.insert(i, e)` fügt Element `e` an Index `i` in Liste `lst` ein.
- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.
- `lst.reverse()` kehrt die Reihenfolge der Elemente in Liste `lst` um.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `lst.insert(i, e)` fügt Element `e` an Index `i` in Liste `lst` ein.
- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.
- `lst.reverse()` kehrt die Reihenfolge der Elemente in Liste `lst` um.
- `list(cont)` erstelle eine neue Liste mit dem Inhalt des Kontainers `cont`.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.
- `lst.reverse()` kehrt die Reihenfolge der Elemente in Liste `lst` um.
- `list(cont)` erstelle eine neue Liste mit dem Inhalt des Kontainers `cont`.
- `==`, `!=`, `is` und `is not` funktionieren auch mit Listen.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.
- `lst.reverse()` kehrt die Reihenfolge der Elemente in Liste `lst` um.
- `list(cont)` erstelle eine neue Liste mit dem Inhalt des Kontainers `cont`.
- `==`, `!=`, `is` und `is not` funktionieren auch mit Listen.

```python
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

↓ python3 lists_2.py ↓

```
The numbers are: [1, 7, 56, 2, 4].
is 7 in the list: True
is 2 NOT in the list: False
7 ist at index 1.
2 ist at index 3.
After inserting 12, the numbers are: [1, 7, 12, 56, 2, 4].
After removing 56, numbers are: [1, 7, 12, 2, 4].
The sorted numbers are: [1, 2, 4, 7, 12].
The reversed numbers are: [12, 7, 4, 2, 1].
cpy == numbers: True.
cpy is numbers: False.
cpy == numbers: False.
cpy is numbers: False.
cpy is not numbers: True.
```

Suchen, einfügen, löschen, sortieren, kopieren und vergleichen

- `lst.remove(e)` löscht Element `e` aus der Liste `lst`.
- `lst.sort()` sortiert die Liste `lst`.
- `lst.reverse()` kehrt die Reihenfolge der Elemente in Liste `lst` um.
- `list(cont)` erstelle eine neue Liste mit dem Inhalt des Kontainers `cont`.
- `==`, `!=`, `is` und `is not` funktionieren auch mit Listen.

```
1  The numbers are: [1, 7, 56, 2, 4].
2  is 7 in the list: True
3  is 2 NOT in the list: False
4  7 ist at index 1.
5  2 ist at index 3.
6  After inserting 12, the numbers are: [1, 7, 12, 56, 2, 4].
7  After removing 56, numbers are: [1, 7, 12, 2, 4].
8  The sorted numbers are: [1, 2, 4, 7, 12].
9  The reversed numbers are: [12, 7, 4, 2, 1].
10 cpy == numbers: True.
11 cpy is numbers: False.
12 cpy == numbers: False.
13 cpy is numbers: False.
14 cpy is not numbers: True.
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Die Addition `lst1 + lst2` von zwei Listen `lst1` und `lst2` erzeugt eine neue Liste mit den Elementen von `lst1` gefolgt von den Elementen von `lst2`.

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Die Addition `lst1 + lst2` von zwei Listen `lst1` und `lst2` erzeugt eine neue Liste mit den Elementen von `lst1` gefolgt von den Elementen von `lst2`.
- Die Multiplikation `lst * i` der Liste `lst` mit dem `int i` erzeugt eine neue Liste, in der die Elemente von `lst` `i`-Mal hintereinander vorkommen.

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Die Addition `lst1 + lst2` von zwei Listen `lst1` und `lst2` erzeugt eine neue Liste mit den Elementen von `lst1` gefolgt von den Elementen von `lst2`.
- Die Multiplikation `lst * i` der Liste `lst` mit dem `int i` erzeugt eine neue Liste, in der die Elemente von `lst` `i`-Mal hintereinander vorkommen.
- Listen können genauso ge-sliced werden wie Strings[12].

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Die Multiplikation `lst * i` der Liste `lst` mit dem `int` `i` erzeugt eine neue Liste, in der die Elemente von `lst` `i`-Mal hintereinander vorkommen.
- Listen können genauso ge-sliced werden wie Strings[12].
- Listen-Slices sind immer neue Listen. Sie können unabhängig von der Originalliste verändert werden.

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Listen können genauso ge-sliced werden wie Strings[1][2].
- Listen-Slices sind immer neue Listen. Sie können unabhängig von der Originalliste verändert werden.
- Listen können durch Mehrfachzuweisungen „ausgepackt“ werden, wobei die Anzahl der Variablen auf der linken Seite genau der Länge der Liste auf der rechten Seite entsprechen muss. `a, b = lst` packt die Elemente einer Liste `lst` der Länge 2 in die Variablen `a` und `b` aus.

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Listen können genauso ge-sliced werden wie Strings[1][2].
- Listen-Slices sind immer neue Listen. Sie können unabhängig von der Originalliste verändert werden.
- Listen können durch Mehrfachzuweisungen „ausgepackt“ werden, wobei die Anzahl der Variablen auf der linken Seite genau der Länge der Liste auf der rechten Seite entsprechen muss. `a, b = lst` packt die Elemente einer Liste `lst` der Länge 2 in die Variablen `a` und `b` aus.

```python
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

↓ python3 lists_3.py ↓

```
lst3 = lst1 + lst2 == [1, 2, 3, 4, 5, 6, 7]
lst4 = lst2 * 3 == [5, 6, 7, 5, 6, 7, 5, 6, 7]
lst5 = lst4[2:-2] == [7, 5, 6, 7, 5]
lst6 = lst4[1::2] == [6, 5, 7, 6]
lst7 = lst4[-1:3:-2] == [7, 5, 6]
lst4 = [5, 6, 7, 5, 6, 7, 5, 6, 7], lst7 = [7, 12, 6]
a = 5, b = 6, c = 7
```

Konkatenation, Addition, Multiplikation, Slices, und auspacken

- Listen können genauso ge-sliced werden wie Strings[12].
- Listen-Slices sind immer neue Listen. Sie können unabhängig von der Originalliste verändert werden.
- Listen können durch Mehrfachzuweisungen „ausgepackt“ werden, wobei die Anzahl der Variablen auf der linken Seite genau der Länge der Liste auf der rechten Seite entsprechen muss. `a, b = lst` packt die Elemente einer Liste `lst` der Länge 2 in die Variablen `a` und `b` aus.

```
1 lst3 = lst1 + lst2 == [1, 2, 3, 4, 5, 6, 7]
2 lst4 = lst2 * 3 == [5, 6, 7, 5, 6, 7, 5, 6, 7]
3 lst5 = lst4[2:-2] == [7, 5, 6, 7, 5]
4 lst6 = lst4[1::2] == [6, 5, 7, 6]
5 lst7 = lst4[-1:3:-2] == [7, 5, 6]
6 lst4 = [5, 6, 7, 5, 6, 7, 5, 6, 7], lst7 = [7, 12, 6]
7 a = 5, b = 6, c = 7
```

Zusammenfassung

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.
- Listen sind Sequenzen von Objekten.

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.
- Listen sind Sequenzen von Objekten.
- Listen können beliebige und beliebig viele Objekte beinhalten.

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.
- Listen sind Sequenzen von Objekten.
- Listen können beliebige und beliebig viele Objekte beinhalten.
- Listenvariablen sollten mit Type Hints annotiert werden.

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.
- Listen sind Sequenzen von Objekten.
- Listen können beliebige und beliebig viele Objekte beinhalten.
- Listenvariablen sollten mit Type Hints annotiert werden.
- Listen können genau wie Zeichenketten (Strings) indiziert werden.

Zusammenfassung

- Mit Listen haben wir nun den ersten Kontainerdatentyp kennengelernt.
- Listen sind Sequenzen von Objekten.
- Listen können beliebige und beliebig viele Objekte beinhalten.
- Listenvariablen sollten mit Type Hints annotiert werden.
- Listen können genau wie Zeichenketten (Strings) indiziert werden.
- Listen sind ein wichtiges Werkzeug, um dynamisch veränderliche Kollektionen von Objekten zu verarbeiten.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] "Built-in Types". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html (besucht am 2024-08-22) (siehe S. 5–10).

[2] "`collections.abc` – Abstract Base Classes for Containers". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/collections.abc.html (besucht am 2024-08-22) (siehe S. 5–10).

[3] "Data Model". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3. URL: https://docs.python.org/3/reference/datamodel.html (besucht am 2024-08-22) (siehe S. 5–10).

[4] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 54).

[5] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 54).

[6] Łukasz Langa. *Type Hinting Generics In Standard Collections*. Python Enhancement Proposal (PEP) 585. Beaverton, OR, USA: Python Software Foundation (PSF), 3. März 2019. URL: https://peps.python.org/pep-0585 (besucht am 2024-10-09) (siehe S. 17–21).

[7] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 54).

[8] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: https://peps.python.org/pep-0586 (besucht am 2024-12-17) (siehe S. 54).

[9] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra" Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 54).

[10] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 54).

References II

[11] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 54).

[12] "Sequences". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.2.5. URL: https://docs.python.org/3/reference/datamodel.html#sequences (besucht am 2024-08-24) (siehe S. 11–15, 37–43).

[13] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 54).

[14] *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/reference (besucht am 2025-04-27).

[15] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library (besucht am 2025-04-27).

[16] ."Literals". In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (besucht am 2025-08-29) (siehe S. 54).

[17] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 54).

[18] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 54).

[19] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 54).

Glossary (in English) I

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[13,17]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[11,17]. Learn more at https://github.com.

literal A literal is a specific concrete value, something that is written down as-is[8,16]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

Mypy is a static type checking tool for Python[9] that makes use of type hints. Learn more at https://github.com/python/mypy and in[19].

Python The Python programming language[4,7,10,19], i.e., what you will learn about in our book[19]. Learn more at https://python.org.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[5,18]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[17]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.