LU04c - List Comprehensions

List Comprehensions sind eine kompakte Schreibweise, um aus vorhandenen Daten neue Listen zu erzeugen. Man kann damit Elemente **transformieren** (z. B. quadrieren, in Grossbuchstaben umwandeln) und optional **filtern** – in einer einzigen, gut lesbaren Zeile. Die Schreibweise erinnert an mathematische Mengen-Schreibweise.

Hauptverwendungszweck von List Comprehensions

List Comprehensions werden in Python hauptsächlich für zwei Dinge verwendet:

- **Transformation**: Jedes Element eines Iterables wird in eine neue Form gebracht (z. B. `x` → `x**2` oder `„audi“` → `„AUDI“`). * Filterung (optional)**: Es werden nur diejenigen Elemente übernommen, für die eine Bedingung erfüllt ist. Typisch ist eine Kombination aus beidem: > „Nimm alle Elemente aus einer bestehenden Datenquelle, filtere sie nach einer Bedingung und wandle sie in eine neue Form um.“ ===== Die Syntax ===== Die allgemeine Form einer List Comprehension ist: <code python> newlist = [expression for item in iterable if condition] </code>** * expression**: Der Ausdruck, dessen Wert in die neue Liste aufgenommen wird (oft eine Transformation von `item`). *** item**: Der Name für das aktuelle Element in der Schleife. *** iterable**: Die Datenquelle, über die iteriert wird (z. B. Liste, Tupel, range, String, …). *** condition **(optional): Eine Bedingung, die `True` oder `False` ergibt. Nur wenn sie `True` ist, wird `expression` in die neue Liste aufgenommen. Ohne Filter-Bedingung sieht die einfachste Form so aus: <code python> newlist = [expression for item in iterable] </code> Das ist äquivalent zu: <code python> newlist = [] for item in iterable: # optional: if condition: newlist.append(expression) </code> ===== Teile einer List Comprehension ===== ==== Expression ==== Die** Expression **ist der Teil, der tatsächlich in der neuen Liste landet. Sie kann: * einfach das aktuelle Element sein (`item`) * oder eine beliebige Berechnung / Funktionsanwendung auf `item`. -** Beispiel mathematische Operation**: <code python> # Expression = x * 2 doubled = [x * 2 for x in range(5)] # Ergebnis: [0, 2, 4, 6, 8] </code> -** Beispiel mit Funktionsaufruf**: <code python> cars = ['audi', 'bmw', 'subaru', 'toyota'] newlist = [car.upper() for car in cars] print(newlist) # ['AUDI', 'BMW', 'SUBARU', 'TOYOTA'] </code> ==== Item ====** item **ist der Name für das aktuelle Element der Iteration. Er wird im `for`-Teil definiert: `for item in iterable`. -** Beispiel**: <code python> cars = ['audi', 'bmw', 'subaru', 'toyota'] newlist = [item.upper() for item in cars] print(newlist) </code> Hier ist `item` nacheinander `'audi'`, `'bmw'`, `'subaru'`, `'toyota'`. ==== Iterable ==== Das** Iterable **kann ein beliebiges iterierbares Objekt sein, z. B. eine Liste, ein Tupel, ein Set, ein String oder das Ergebnis von `range()`. -** Beispiel mit einer Liste**: <code python> squared = [x**2 for x in [1, 2, 3, 4]]

Ergebnis: [1, 4, 9, 16]

numbers = [1, 2, 3, 4] squared = [x**2 for x in numbers] # Ergebnis: [1, 4, 9, 16] </code> -** Beispiel mit einem Tupel**: <code python> doubled = [x * 2 for x in (1, 2, 3, 4)] # Ergebnis: [2, 4, 6, 8] </code> -** Beispiel mit range()**: <code python> newlist = [x for x in range(10)] # Ergebnis: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] </code> ==== Condition ==== Die** Condition **ist ein optionaler Filter, der entscheidet, ob ein Element in die neue Liste aufgenommen wird.**

Nur wenn die Bedingung `True` ergibt, wird `expression` übernommen. - Beispiel mit Filter**: <code python> cars = ['audi', 'vw', 'seat', 'skoda', 'tesla'] newlist = [car for car in cars if car != 'tesla'] # newlist = ['audi', 'vw', 'seat', 'skoda'] </code> Die Bedingung `car != 'tesla'` liefert `True` für alle Elemente ausser `'tesla'`, daher enthält die neue Liste alle Autos ausser Tesla. Die Bedingung ist optional und kann weggelassen werden: -** Ohne Filter**: <code python> newlist = [x for x in cars] # Kopie der ursprünglichen Liste </code> ===== Weitere Beispiele ===== -** Mit einem Iterable eines anderen Typs (String)**: <code python> letters = [letter for letter in 'Hallo Welt'] # Ergebnis: ['H', 'a', 'l', 'l', 'o', ' ', 'W', 'e', 'l', 't'] </code> -** Nested List Comprehensions (verschachtelt)**: <code python> matrix = 1, 2, 3], [4, 5, 6], [7, 8, 9 flattened = [num for row in matrix for num in row] # Ergebnis: [1, 2, 3, 4, 5, 6, 7, 8, 9] </code>**

List Comprehensions sind ein mächtiges Werkzeug in Python, um vorhandene Daten in einer einzigen, klaren Zeile zu **transformieren** und (falls nötig) zu **filtern**. Richtig eingesetzt machen sie Code kürzer und oft besser lesbar.

—-

M323-LU04