Listen

by Woche 2

Wie bereits erwähnt sind Listen die am häufigsten verwendete Art von Sammlungen in Python. Sie werden mit eckigen Klammern geschrieben, können heterogen sein und auch verschachtelt werden.

Außerdem sind Listen geordnert, können Duplikate enthalten und sie sind mutable. Was diese Begriffe bedeuten, wird im Folgenden genauer erläutert. Da Listen die erste Art Sammlung ist, die wir uns genauer ansehen, werden wir ausführlicher auf die Eigenschaften und Funktionen von Listen eingehen um dann bei den anderen Sammlungen nur noch auf die Unterschiede zu Listen eingehen zu müssen.

Mutable vs. Immutable

Bei der Zuweisung von Variablen in Python muss man sich bewusst sein, dass es “mutable” (veränderbar) und “immutable” (unveränderbare) Objekttypen gibt.

- Objekte, die **immutable** sind, können nach ihrer Erzeugung nicht verändert werden. Führt man eine Operation aus, die das Objekt zu verändern scheint, so wird im Speicher tatsächlich ein völlig neues Objekt erzeugt.
- Objekte, die **mutable** sind, werden stattdessen bei einer entsprechenden Operation direkt im Speicher verändert. Wenn sich eine andere Variable auf dasselbe Objekt bezieht, so ändert sich auch der Wert dieser Variable.

Der wichtige Unterschied soll durch diese Beispiele verdeutlicht werden. In beiden Fällen erzeugen wir zunächst eine Variable `var1` und weisen ihr einen Wert zu. Dann weisen wir `var1` einer weiteren Variable `var2` zu und verändern (vermeintlich nur) `var2`.

Immutable

```
var1 = 'Text'
var2 = var1
var2 = var2.lower()

print(var1)
```

```
Text
```

```
print(var2)
```

```
text
```

Alle grundlegenden Datentypen sind *immutable*, sodass bei der Zuweisung des Strings `'Text'` zu `var1` ein *immutable* Objekt erzeugt wird. Durch die Zuweisung von `var1` zu `var2` entsteht zunächst lediglich ein Verweis auf dasselbe Objekt. Sobald allerdings `var2` mit der Funktion `lower()` verändert wird (Großbuchstaben werden zu Kleinbuchstaben), wird ein neues Objekt `var2` im Speicher erzeugt, sodass `var2`, nicht aber `var1` verändert erscheinen.

Als Eselsbrücke kann man sich vorstellen, dass die Informationen auf einem ausgedruckten Blatt Papier stehen. Sie sind somit unveränderbar und können nur durch ein neues Blatt Papier ersetzt werden.

Beispiele:

- Zahlen
- Strings
- Tuples

Mutable

```
var1 = [1, 2, 3]
var2 = var1
var2.append(4)

print(var1)
```

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

```
print(var2)
```

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

In `var1` wird hier eine Liste mit den Zahlen 1, 2 und 3 erzeugt. Mit der Zuweisung von `var1` zu `var2` entsteht auch hier zunächst ein Verweis auf dieselbe Liste. Schließlich wird `var2` mit der Funktion `append()` um die Zahl 4 erweitert. Da Listen *mutable* Objekte sind, wird das Objekt im Speicher verändert. Da `var2` lediglich auf `var1` verweist, wird die zugrundeliegende Liste in `var1` geändert, sodass letztendliche `var1` und `var2` verändert erscheinen.

Als Eselsbrücke kann man sich vorstellen, dass die Informationen auf einem Whiteboard stehen. Sie sind somit veränderbar und können durch Überschreiben oder Löschen verändert werden.

Beispiele:

- Listen
- Dictionaries
- Sets

Je nach Erfahrung z.B. mit anderen Programmiersprachen fühlt sich die eine oder andere Variante ggf. intuitiver an. Wie so oft haben beide ihre Vor- und Nachteile.

Der Vorteil von immutable Objekten ist, dass sie die Integrität und Vorhersehbarkeit des Codes erhöhen. Da ein immutable Objekt nach seiner Erstellung nicht mehr verändert werden kann, verringert dies das Risiko unbeabsichtigter Nebeneffekte. Das macht den Code zuverlässiger und oft einfacher zu debuggen, da die Werte von immutablen Objekten vorhersagbar bleiben.

Der Vorteil von mutable Objekten ist hingegen ihre Effizienz, da nicht bei jeder Änderung eine neue Kopie des Objekts erzeugt werden muss. In Performance-kritischen Anwendungen bzw. mit großen Datenmengen kann dies einen erheblichen Unterschied machen.

Insgesamt ist die Wahl zwischen “mutable” und “immutable” Objekten eine Frage des spezifischen Anwendungsfalls und des gewünschten Verhaltens.

Mit Listen arbeiten

Erstellen

Eine Liste wird erstellt, indem eine durch Kommas getrennte Sequenz von Objekten in eckigen Klammern angegeben wird:

```
meine_liste = ['Etwas', 42, 'Text', 42, 12.6, True]
meine_liste
```

```
['Etwas', 42, 'Text', 42, 12.6, True]
```

```
zahlen_liste = [1, 6, 3, 2]
zahlen_liste
```

```
[1, 6, 3, 2]
```

Wie bereits erwähnt, können auch mehrere Listen (oder andere Sammlungen) in eine Liste getan werden, sodass eine geschachtelte Liste (*nested list*) entsteht:

```
geschachtelte_liste = [meine_liste, zahlen_liste]
geschachtelte_liste
```

```
[['Etwas', 42, 'Text', 42, 12.6, True], [1, 6, 3, 2]]
```

Es ist auch möglich eine leere Liste via `[]` zu erzeugen, sowie bestimmte andere Objekte mit der Funktion `list()` in Listen umzuwandeln.

```
x = []
x
```

```
[]
```

```
x = list('Gut')
x
```

```
['G', 'u', 't']
```

Indizierung

Listen und Sequenzen in Python sind indiziert, d.h. jede Position in der Sequenz hat eine zugehörige Nummer - den Index. Mit diesem kann man auf den Wert an dieser Position zugreifen. Python-Sequenzen beginnen bei Null **(!)**, sodass das erste Element einer Sequenz den Index 0 hat, das zweite Element den Index 1 und so weiter. Ein Element aus einer Liste kann abgerufen werden, indem man den Index in eckige Klammern hinter den Listennamen setzt. Es kann sogar mit negative Indices rückwärts auf die Liste zugegriffen werden, dann ist das letzte Element aber bei −1:

```
x = ['Timon', 'sagt', 'Hakuna', 'Matata', '!']
```

```
x[0]
```

```
'Timon'
```

```
x[2]
```

```
'Hakuna'
```

```
x[-1]
```

```
'!'
```

Mittels Indizierung kann man die einzelnen Elemente nicht nur ansprechen, sondern auch verändern, wobei `del()` genutzt wird um ein Element zu löschen:

```
x[0] = 'Pumbaa'
x
```

```
['Pumbaa', 'sagt', 'Hakuna', 'Matata', '!']
```

Auch die Elemente in geschachtelten Listen können entsprechend mit zusätzlichen Indizes angesprochen werden:

```
x = [['D', 'B', 'Z'], [1, 2]]

x[0]
```

```
['D', 'B', 'Z']
```

```
x = [['D', 'B', 'Z'], [1, 2]]

x[0][1]
```

```
'B'
```

Slicing

Slicing, also das Ausschneiden einer Teilmenge, funktioniert ähnlich wie die Indizierung, allerdings kann in den eckigen Klammern die Syntax `[start:stop:step]` angewendet werden. Dabei bezeichnet

- `start` den Index des ersten Elements ab und inklusive dem ausgeschnitten werden soll (Default = 0)
- `stop` den Index des letzten Elements bis und exklusive **(!)** dem ausgeschnitten werden soll (Default = letztes Element)

- `step` wie of im Ausschnitt Elemente abgetasted werden sollen (Default = 1). Auch hier kann mit negativen Werten gearbeitet um rückwärts abzutasten.

```
x = [1, 2, 3, 4, 5]

x[0:3]
```

```
[1, 2, 3]
```

```
x = [1, 2, 3, 4, 5]

x[0:3:2]
```

```
[1, 3]
```

Lässt man die Werte frei, werden die *defaults* angewendet:

```
x = [1, 2, 3, 4, 5]

x[1:]
```

```
[2, 3, 4, 5]
```

```
x = [1, 2, 3, 4, 5]

x[::2]
```

```
[1, 3, 5]
```

Hinzufügen

Mit `list.append()` kann ein Element ans Ende einer Liste hinzugefügt werden, während mit `list.insert()` ein Element an einer bestimmten Position (=Index; siehe weiter unten) eingefügt werden kann.

```
x = [1, 2, 3]

x.append(10)
x # enthält nun 10 am Ende
```

```
[1, 2, 3, 10]
```

```
x = [1, 2, 3]

x.insert(1, 33)
x # enthält nun 33 an Index 1
```

```
[1, 33, 2, 3]
```

Wenn nicht nur ein einzelnes Element, sondern z.B. eine andere Liste zu einer bestehenden Liste hinzugefügt werden soll, so funktioniert dies entweder schlicht mit einem `+` oder mit `list.extend()`. Der Unterschied ist, dass `+` eine neue Liste erzeugt, während `list.extend()` die bestehende Liste verändert. Wir demonstrieren dies mit Liste `b` von oben:

```
x = [1, 2]
y = ['drei', True]

z = x + y
z
```

```
[1, 2, 'drei', True]
```

```
x = [1, 2]
y = ['drei', True]

x.extend(y)
x
```

```
[1, 2, 'drei', True]
```

Entfernen

Der Befehle `list.clear` entfernt alle Elemente aus einer Liste, während `list.remove()` (**nur!**) das erste passende Element aus einer Liste entfernt. Um ein Element an einer bestimmten Position zu entfernen, kann `del()` in Kombination mit dem Index (nicht dem zu entfernenden Wert!) verwendet werden.

```
x = [1, 2, 3, 2, 1]
```

```
x.clear()
x
```

```
[]
```

```
x = [1, 2, 3, 2, 1]

x.remove(2)
x
```

```
[1, 3, 2, 1]
```

```
x = [1, 2, 3, 2, 1]

del(x[2])
x
```

```
[1, 2, 2, 1]
```

Schließlich sei noch die Funktion `list.pop()` erwähnt, die stets das letzte Element einer Liste herauslöst, d.h. in der ursprünglichen Liste entfernt und gleichzeitig zurückgibt:

```
x = [1, 2, 3]
y = x.pop()

print(y) # y ist nun 3 und
```

```
3
```

```
print(x) # x enthält 3 nicht mehr
```

```
[1, 2]
```

Verändern

Weitere gebräuchliche Listenfunktionen sind `list.sort()`, welche die Elemente einer Liste sortiert, und `list.reverse()`, welche die Reihenfolge der Elemente umkehrt. Beide Funktionen verändern die Liste, auf die sie angewendet werden. `list.sort()` kann auch mit dem Schlüsselwort `reverse = True` angewendet werden, um die Liste in umgekehrter Reihenfolge zu sortieren. Wir nutzen hier die `zahlen_liste = [1, 6, 3, 2]` von oben.

```
x = [3, 1, 2]

x.reverse()
x
```

```
[2, 1, 3]
```

```
x = [3, 1, 2]

x.sort()
x
```

```
[1, 2, 3]
```

```
x = [3, 1, 2]

x.sort(reverse = True)
x
```

```
[3, 2, 1]
```

Prüfen

Es ist oft hilfreich bestimmte Attribute von Listen zu extrahieren bzw. zu prüfen. Die Länge einer Liste, also die Anzahl der in ihr enthaltenen Elemente, kann mittels `len()` ausgegeben werden und wird auch häufig für weitere Operationen benötigt. Ebenso können die Schlüsselwörter `in` und `not` verwendet werden um zu prüfen ob ein Element bzw. ob ein Element nicht in einer Liste enthalten ist. Schließlich kann auch mittels `list.count()` gezählt werden wie oft ein Element in einer Liste enthalten ist.

```
x = [42, 1, 42]

len(x)
```

```
3
```

```
x = [42, 1, 42]

42 in x
```

```
True
```

```
x = [42, 1, 42]

42 not in x
```

```
False
```

```
x = [42, 1, 42]

x.count(42)
```

```
2
```

Gegeben, dass die Liste ausschließlich Zahlen enthält, können auch einfache Rechenoperationen wie Bestimmung des Maximums, Minimums oder der Summe durchgeführt werden.

```
x = [3, 1, 2]

max(x)
```

```
3
```

```
x = [3, 1, 2]

min(x)
```

```
1
```

```
x = [3, 1, 2]

sum(x)
```

```
6
```

Kopieren

Wie bereits erwähnt, sind Listen in Python mutable Objekte. Natürlich kann es aber auch gewünscht sein eine Kopie einer Liste zu erzeugen, die eben nicht dauerhaft auf eine andere Liste verweist, sondern ab dem Moment eine echte, unabhängige Kopie darstellt, mit der separat weitergearbeitet werden kann. Zumindest bedingt **(!)** ist dies möglich via `list.copy()`:

```
liste1 = [1, 2, 3]
liste2 = liste1
liste2.append(4)

print(liste1)
```

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

```
print(liste2)
```

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

```
liste3 = [1, 2, 3]
liste4 = liste3.copy()
liste4.append(4)

print(liste3)
```

```
[1, 2, 3]
```

```
print(liste4)
```

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

Wie zu sehen wird - im Gegensatz zum Beispiel mit `liste1` und `liste2` - das Objekt `liste3` eben nicht durch das Anwenden der `append()` Funktion auf `liste4` verändert. Wie zu erwarten verhält sich `liste4` also als unabhängige Kopie von `liste3`. Dies stimmt leider wie gesagt nur bedingt, da `list.copy()` eine sogenannte flache (*shallow*) Kopie erzeugt. Dies meint, dass das Erzeugen von unabhängigen Kopien nur auf der obersten Listen-Ebene erfolgt. Sobald also eine Liste geschachtelt ist und auf den unteren Ebenen weitere mutable Objekte enthält, sind diese weiterhin nicht unabhängig. Um auch diese unabhängig zu machen, also eine tiefe (*deep*) Kopie zu erzeugen, kann die

Funktion `copy.deepcopy()` aus dem `copy`-Modul verwendet werden. Es folgen drei Beispiele, die dies verdeutlichen sollen. Die Beispiele mit `listeA` bis `listeD` entsprechen dabei der Vorgehensweise von oben, wobei mit `listeF` die tiefe Kopie von `listeE` erzeugt wird. In allen Fällen wird sowohl ein Element in der obersten Listenebene, als auch in der Listenebene darunter verändert, sodass sich die Ergebnisse aller drei Beispiele unterscheiden:

```
# kein extra Modul nötig

listeA = [[1,2], [3,4]]
listeB = listeA
listeB[1].append(5)
listeB.append(6)

print(listeA)
```

```
[[1, 2], [3, 4, 5], 6]
```

```
print(listeB)
```

```
[[1, 2], [3, 4, 5], 6]
```

```
# kein extra Modul nötig

listeC = [[1,2], [3,4]]
listeD = listeC.copy()
listeD[1].append(5)
listeD.append(6)

print(listeC)
```

```
[[1, 2], [3, 4, 5]]
```

```
print(listeD)
```

```
[[1, 2], [3, 4, 5], 6]
```

```
import copy

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

```
listeF = copy.deepcopy(listeE)
listeF[1].append(5)
listeF.append(6)

print(listeE)
```

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

```
print(listeF)
```

```
[[1, 2], [3, 4, 5], 6]
```

Weitere Ressourcen

- Mutable und Immutable in Python
- Mutable vs Immutable Objects in Python
- Python Tutorial deutsch 14/24 - Einführung in Listen
- Python Tutorial deutsch 15/24 - Zugriff auf Listen
- ALL 11 LIST METHODS IN PYTHON EXPLAINED
- More on Lists

Übungen

Erzeuge drei unterschiedlich lange Listen, deren Länge jeweils identisch zu ihrer Summe ist. Keine der Listen darf nur aus Einsen bestehen.

- (A) Geschafft

Erzeuge eine Liste, deren Länge, Summe und Maximalwert identisch ist.

- (A) Geschafft

Erstelle ein separates Jupyter Notebook, in welchem du zu Beginn drei Listen mit den Namen ‘a’, ‘b’ und ‘c’ und einem Inhalt deiner Wahl erstellst. Daraufhin soll jede der in diesem Kapitel genannten Funktionalitäten mindestens einmal auf mindestens eine der Listen angewendet werden. Es gilt also alle Methoden/Funktionen nacheinander anzuwenden, ohne jemals mehr als drei Listen zu verwenden. Diese Übung soll einerseits die Anwendung der Methoden/Funktionen vertiefen und andererseits die Veränderung von mutable Listen durch die Methoden/Funktionen verdeutlichen. Strukturiere das Jupyter Notebook mithilfe von Überschriften und kurzen Erläuterungen so, dass es für jeden Schritt kurz erläutert was geschieht - ähnlich diesem Kapitel hier.

Ziel ist es, dass du dieses Dokument später als Referenz nutzen kannst, falls du dich mal nicht mehr an diese Methoden erinnern solltest.

- (A) Geschafft