Python Tutorial

Projekt Collaborative Writing

Hochschule Kaiserslautern

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Collections

In Python 3 existieren nativ die vier Datenstrukturen List, Tuple, Set und Dictionary, welche im Folgenden vorgestellt werden.

1.1.1 List

Die Datenstruktur List bietet einen geordneten und veränderbaren Behälter für Python-Objekte, der Duplikate von Elementen erlaubt. Da eine List immer sortiert ist, können einzelne Elemente aus der Datenstruktur über den entsprechenden Index ausgewählt und verändert werden. Python unterstützt intern keine Arrays, alternativ hierzu kann eine List verwendet werden.

Eine List kann wie folgt initialisiert werden:

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

# oder
liste = list((1, 2, 3))
```

Dabei kann sie jegliche Art von Objekten beinhalten; der Datentyp spielt hierbei keine Rolle.

Beispiel:

```
liste = [1,"hallo",2.3,(5,6),[22,23,24], 'a']
```

Im Gegensatz zu Java und C++ muss der Programmierer darauf achten und sicherstellen, dass die Datenstruktur mit Werten des entsprechenden Datentyps befüllt wird, um Fehler aufgrund unterschiedlicher Datentypen zu vermeiden.

Der Inhalt einer List kann über die `print()`-Methode ausgegeben werden. Im folgenden Beispiel werden verschiedene Elemente der List auf der Konsole ausgegeben. Wird die List als Parameter gewählt, wird der Inhalt ausgegeben.

```
liste = [1, 2, 3]
print(liste)
```

Wie zuvor erwähnt, ähnelt die Verwaltung einer List der eines Arrays aus Java oder C++. Durch die Verwendung eines Index können einzelne Elemente ausgewählt oder verändert werden.

```
liste = [1, 2, 3]
print(liste[1])
liste[1] = 4
print(liste)
```

Python erlaubt die Nutzung von negativen Indizes. Mit diesen kann der Inhalt der List in umgekehrter Reihenfolge ausgegeben werden. Ein Index von `-1` wird dem letzten Element der List zugeordnet, `-2` dem vorletzten.

```
liste = [1, 2, 3]
print(liste[-1])
print(liste[-2])
```

In Python existiert für die Datenstruktur List keine Methode, die mit `contains()` in Java oder der `find()` aus C++ vergleichbar ist. Stattdessen stehen die Membership Operatoren `in` oder `not in` zur Verfügung, welche auf eine beliebige Sequenz oder die hier beschriebenen Collections angewendet, Auskunft darüber gibt, ob das spezifizierte Element darin enthalten ist.

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

print(2 in liste)

if 2 in liste:
    print("Gefunden!")
else:
    print("Nicht gefunden!")
```

Der Python Interpreter stellt nativ einige Funktionen zur Verfügung. Eine davon ist die `len()`-Methode, welche die Anzahl an Elementen in einem Objekt liefert.

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

print(len(liste))
```

Das `del`-Statement erlaubt das Löschen einzelner Elemente oder der gesamten List.

```
liste = [1, 2, 3]
print(liste)

del liste[1]
print(liste)

del liste

# Erzeugt einen Error, da die Liste nicht mehr existiert!
print(liste)
```

Methoden einer List

append(): Fügt am Ende der List ein Objekt hinzu.

```
liste = [1, 2, 3]
print(liste)
liste.append(4)
print(liste)
```

clear(): Entfernt sämtliche Objekte aus der List.

```
liste = [1, 2, 3]
liste.clear()
print(liste)
```

copy(): Liefert eine Kopie der List.

```
liste = [1, 2, 3]
liste.clear()
print(liste)
```

count(): Liefert die Anzahl des spezifizierten Objekts in der List.

```
liste = [1, 2, 3, 2, 2]
print(liste.count(2))
```

extend(): Fügt der `liste1` den Inhalt der `liste2` am Ende hinzu.

```
liste1 = [1, 2, 3]
liste2 = [4, 5, 6]
liste1.extend(liste2)
print(liste1)
```

index(): Liefert den Index der Position, an der sich das erste spezifizierte Objekt in der List befindet.

```
liste = [1, 2, 3]
print(liste[1])
liste[1] = 4
print(liste)
```

insert(): Fügt ein Objekt an der gewählten Position der List hinzu.

```
liste = ["1", "2", "3"]
liste.insert(1, "4")
print(liste)
```

pop(): Entfernt das Objekt, das sich an der durch den Index spezifizierten Position befindet.

```
liste = [1, 2, 3]
liste.pop(1)
print(liste)
```

remove(): Entfernt das erste Objekt der List, das der Spezifikation entspricht.

```
liste = [1, 2, 3, 2]
liste.remove(2)
print(liste)
```

reverse(): Invertiert die Folge der Objekte in der List.

```
liste = [1, 2, 3]
liste.reverse()
print(liste)
```

sort(): Sortiert die List.

```
# lexikographisches Sortieren
liste = ["b", "c", "a"]
print(liste)
liste.sort()
print(liste)

# sortieren nach Zahlenwert
liste = [6, 1, 2, 3, 4, 5]
print(liste)
liste.sort()
print(liste)

# umkehren der Sortierreihenfolge
liste.sort(reverse=True)
print(liste)

# sortieren nach der Laenge einzelner Objekte
liste = ["aa", "aaa", "a"]


def sortFunc(x):
    return len(x)


liste.sort(key=sortFunc)
print(liste)

# umkehren der Sortierreihenfolge
liste.sort(reverse=True, key=sortFunc)
print(liste)
```

1.1.2 Tuple

Ein Tuple stellt einen geordneten und unveränderbaren Behälter für Python-Objekte dar. Dieser erlaubt, wie eine List, Duplikate und den Zugriff auf einzelne Elemente über einen Index. Tuple sind Datenstrukturen, die ausschließlich gelesen werden können.

Ein Tuple wird mit folgender Syntax erzeugt:

```
tupel = (1, 2, 3)
```

```
# oder
tupel = tuple((1, 2, 3))
```

Es ist möglich, leere Tuple zu erzeugen. Wie zuvor erwähnt, ist deren Inhalt unveränderlich.

Arbeiten mit einem Tuple

Der Inhalt eines Tuple kann, analog zur List, auf der Konsole ausgegeben werden. Das Zuweisen eines neuen Objekts mittels Index führt im Gegensatz zur List zu einem Fehler.

```
tupel = (1, 2, 3)
tupel[0] = 4  # ERROR
```

Die Verwendung der Operatoren `in` und `not in` ist, wie die `len()`-Methode, analog zur List -Datenstruktur.

```
tupel = (1, 2, 3)

print(2 in tupel)
print(len(tupel))
```

Das `del`-Statement erlaubt das Löschen des Tuple. Aufgrund der Unveränderbarkeit der Datenstruktur können keine einzelnen Elemente entfernt werden.

```
tupel = (1, 2, 3)

del tupel
```

Methoden eines Tuple

count(): Liefert die Anzahl des gewählten Werts in einem Tuple.

```
tupel = (1, 2, 4, 3, 2, 2)
print(tupel.count(2))
```

index(): Liefert die Position des ersten Werts, der mit dem spezifizierten Wert übereinstimmt.

```
tupel = (1, 2, 3, 2)
print(tupel.index(2))
```

1.1.3 Set

Ein Set ist durch das Hinzufügen oder Entfernen von Objekten veränderbar und erlaubt keine Duplikate. Das Initialisieren mit mehrfach identischen Werten führt nicht zu einem Fehler, jedoch werden die überzähligen Werte aus dem Set entfernt. Die enthaltenen Elemente sind unveränderlich. Zudem ist die Datenstruktur ungeordnet, weshalb nicht auf einzelne Objekte mittels Index zugegriffen werden kann.

Ein Datenbehälter vom Typ Set kann mit folgender Syntax erzeugt werden:

```
set1 = {1, 2, 3}

# oder
set1 = set((1, 2, 3))
```

Arbeiten mit Sets

Bei der Ausgabe eines Set auf der Konsole ist die Reihenfolge der Elemente zufällig.

Die Syntax für die Ausgabe auf der Konsole ist analog zur List. Die Verwendung eines Index ist nicht erlaubt und führt zu einem Fehler.

```
set1 = {1, 2, 3}
print(set1)
for x in set1:
    print(x)
print(set1[0])  # ERROR
set1[1] = 4  # ERROR
```

Methoden eines Sets

add(): Fügt dem Set ein Objekt hinzu.

```
set1 = {1, 2, 3}
print(set1)
```

```
set1.add(4)
print(set1)
```

clear(): Entfernt alle Elemente aus dem Set.

```
set1 = {1, 2, 3}
print(set1)
set1.clear()
print(set1)
```

copy(): Liefert eine Kopie des Sets.

```
set1 = {1, 2, 3}
x = set1.copy()
print(x)
```

difference(): Liefert ein Set, das diejenigen Elemente enthält, die ausschließlich in `setX` vorkommen. Alle Element, die mit denen von `setY` übereinstimmen, werden aus dem ersten entfernt. Alternativ ist dies auch über den Operator - möglich.

```
set1 = {1, 2, 3}
set2 = {3, 8, 4}
x = set1.difference(set2)
print(x)

# oder
y = set1 - set2
print(y)
```

difference_update(): Entfernt diejenigen Elemente aus dem ersten Set, die mit denen aus dem zweiten übereinstimmen.

```
setX = {1, 2, 3}
setY = {3, 8, 4}
setX.difference_update(setY)
print(setX)
```

discard(): Entfernt das gewählte Element aus dem Set. Duplikate werden ebenfalls entfernt.

```
set1 = {1, 2, 4, 3, 4}
set1.discard(4)
print(set1)
```

intersection(): Liefert ein Set mit der Schnittmenge zweier Sets. Alternativ ist dies auch mit der Angabe des `&`-Operators möglich.

```
setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX & setY)
print(setX.intersection(setY))
```

intersection_update():

```
setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
setX.intersection_update(setY)
print(setX)
```

Entfernt alle Elemente, die sich nicht in der Schnittmenge beider Sets befinden.

isdisjoint(): Gibt Auskunft darüber, ob zwei Sets eine Schnittmenge besitzen. Liefert `True`, wenn kein Element des ersten Sets im zweiten enthalten ist.

```
setX = {1, 2, 3, 4, 5}
setY = {6, 7, 8, 9, 10}
print(setX.isdisjoint(setY))  # Liefert True

setX = {1, 2, 3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.isdisjoint(setY))  # Liefert False
```

issubset(): Gibt an, ob das gewählte Set eine Teilmenge enthält, die exakt dem ersten Set entspricht. Alternativ kann das Zeichen < verwendet werden.

```
setX = {3, 4, 5}
setY = {3, 4, 9, 5, 8, 7}
print(setX.issubset(setY))
print(setX < setY)
```

pop(): Entfernt ein beliebiges Element aus dem Set. Sollte das Set leer sein, wird ein Fehler generiert.

```
set1 = {1, 2, 3}
set1.pop()
print(set1)
```

remove(): Entfernt das gewählte Element aus dem Set. Sollte das gewählte Element nicht in dem Set enthalten sein, wird ein Fehler angezeigt.

```
set1 = {1, 2, 3}
print(set1)
set1.remove(3)
print(set1)
```

symmetric_difference(): Liefert ein Set, das die Vereinigung zweier Sets ohne deren Schnittmenge enthält.

```
set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
print(set1.symmetric_difference(set2))
```

symmetric_difference_update(): Vereinigt zwei Sets und entfernt deren Schnittmenge.

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
set1.symmetric_difference_update(set2)
print(set1)
```

union(): Liefert ein Set, das die Vereinigung zweier Sets darstellt. Duplikate werden entfernt.

```
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8, 9, 10}
print(set1.union(set2))
```

update(): Fügt einem Set die Items eines anderen hinzu. Duplikate werden entfernt.

```
set1 = {1, 2, 3, 4, 5}
set2 = {6, 7, 8, 9, 10}
set1.update(set2)
print(set1)
```

Frozenset

Im Gegensatz zu einem „normalen“ Set kann ein Frozenset nicht mehr verändert werden. Das Hinzufügen eines neuen Elements ist nicht erlaubt und führt zu einem Fehler.

```
set1 = frozenset([1, 2, 3, 4])
set1.add(5)  # ERROR
```

1.1.4 Dictionary

Ein Dictionary ist eine ungeordnete, veränderbare Datenstruktur, die keine Duplikate erlaubt und Schlüssel-Objekt-Paare beinhaltet. Bei einer Ausgabe werden die Werte in zufälliger Reihenfolge ausgegeben, denn ein Dictionary besitzt keine Ordnung.

Ein Datenbehälter vom Typ Dictionary kann mit folgender Syntax erzeugt werden:

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}

dictionary = dict(k1="v1", k2="v2", k3="v3")
```

Demnach befindet sich hinter dem Schlüssel `k1` das Objekt `v1` und analog dazu die weiteren Schlüssel-Objekt-Paare. Über den Schlüssel `k1` lässt sich auf das Objekt `v1` direkt zugreifen. Ebenso kann ein neues Objekt unter dem Schlüssel `k1` zugewiesen werden.

```
print(dictionary["k1"])
```

Eine alternative Möglichkeit, ein Dictionary zu erstellen, ist die Methode `zip()`. Mit deren Hilfe kann aus zwei separaten List-Behältern ein Dictionary generiert werden.

```
sprache = ["englisch", "deutsch", "franzÃ¶sisch"]
laender = ["England", "Deutschland", "Frankreich"]

laendersprache = dict(zip(laender, sprache))

print(laendersprache)
```

Methoden eines Dictionary

clear(): Entfernt alle Einträge aus dem Dictionary.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.clear()
print(dictionary)
```

copy(): Liefert eine Kopie des Dictionary.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
x = dictionary.copy()
print(x)
```

fromkeys(): Liefert ein Dictionary mit den angegebenen Schlüsseln und Objekten.

```
x = ("k1", "k2", "k3")
y = "v"
dictionary = dict.fromkeys(x, y)
print(dictionary)
```

get(): Liefert das Objekt, das dem angegebenen Schlüssel zugeordnet ist.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.get("k1"))
```

items(): Liefert eine List mit einem Tuple für jedes Schlüssel-Objekt-Paar.

```
dictionary = {
    "k1": "v1",
```

```
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.items())
```

keys(): Liefert eine List von allen im Dictionary verwendeten Schlüsseln.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.keys())
```

pop(): Entfernt das Element mit dem entsprechenden Schlüssel aus dem Dictionary und liefert das Objekt zurück.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.pop("k1"))
```

popitem(): Liefert das zuletzt hinzugefügte Schlüssel-Objekt-Paar als Tuple und entfernt es aus dem Dictionary.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.popitem())
```

setdefault(): Liefert das dem Schlüssel zugeordneten Objekt. Existiert dieser Schlüssel nicht, wird ein neues Schlüssel-Objekt-Paar mit dem angegebenen Schlüssel und Objekt angelegt.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
```

```
x = dictionary.setdefault("k2", "v4")
print(x)
print(dictionary)
x = dictionary.setdefault("k4", "v5")
print(x)
print(dictionary)
```

update(): Fügt dem Dictionary ein Schlüssel-Objekt-Paar hinzu.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary)
dictionary.update({"k5": "v5"})
print(dictionary)
```

values(): Liefert eine Liste mit allen im Dictionary enthaltenen Werten.

```
dictionary = {
    "k1": "v1",
    "k2": "v2",
    "k3": "v3"
}
print(dictionary.values())
```

1.1.5 Zusammenfassung

In diesem Abschnitt wurde gezeigt, dass Python 3 uns mehrere Collections zur Aufbewahrung von Daten bereitstellt. Diese können je nach Datenstruktur unterschiedliche Eigenschaften aufweisen. Während eine List die Daten sortiert vorhält und Duplikate zulässt, werden bei einem Set entsprechende doppelte Einträge vermieden. Betrachtet man das Set und seine Methoden genauer, ist dies der dahinter liegenden Mathematik, konkret der Mengenlehre geschuldet. Aus diesem Grund können alle gängigen mathematischen Operation auf Sets angewendet werden. Zum Schluss haben wir in diesem Abschnitt das Dictionary kennengelernt. Dieses ist ähnlich den Maps in Java. Dabei besteht ein Dictionary aus Schlüssel-Objekt-Paaren, die hinter jedem Schlüssel ein entsprechendes Objekt Mappen bzw. bereitstellen.

Kapitel 2

Benutzeroberflächen

Kapitel 3

Python Bibliotheken

Kapitel 4

Weiterführende Themen

Literaturverzeichnis