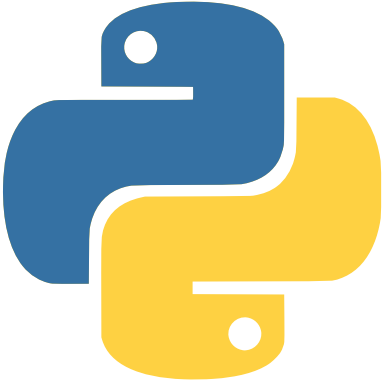
Universität Stuttgart
Projekt digit@L – BOOST. SKILLS. SUPPORT.



Programmierkurs Python

Mutable vs. Immutable

Dominik Göddeke

IANS SimTech

Stiftung Innovation in der Hochschullehre

CC BY

Mutable vs. immutable

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt
- Natürlich nicht alle . . .

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt
- Natürlich nicht alle . . .
- Umgang in der Regel angenehm und, nach Einarbeitung, auch intuitiv

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt
- Natürlich nicht alle . . .
- Umgang in der Regel angenehm und, nach Einarbeitung, auch intuitiv
- Problem: Existenz gewisser **„Tücken“**

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt
- Natürlich nicht alle . . .
- Umgang in der Regel angenehm und, nach Einarbeitung, auch intuitiv
- Problem: Existenz gewisser **„Tücken“**
- Kann bei ersten Kontakt mit Python etwas herausfordernd sein

Einleitung

- **Zwischenstand**: reichlich Datentypen und Datenstrukturen kennengelernt
- Natürlich nicht alle . . .
- Umgang in der Regel angenehm und, nach Einarbeitung, auch intuitiv
- Problem: Existenz gewisser **„Tücken“**
- Kann bei ersten Kontakt mit Python etwas herausfordernd sein
- Deshalb dieser längere und langsamere Abschnitt

Codebeispiele

Diskussion

Diskussion

- Modifikation der ersten Liste über die zweite

Diskussion

- Modifikation der ersten Liste über die zweite
- `id()` liefert: alle Variablen zeigen auf das gleiche **Objekt** (die gleiche Stelle) im Speicher, in diesem Fall die gleiche Liste

Diskussion

- Modifikation der ersten Liste über die zweite
- `id()` liefert: alle Variablen zeigen auf das gleiche **Objekt** (die gleiche Stelle) im Speicher, in diesem Fall die gleiche Liste
- Sprechweise: `lst2` ist **Referenz** auf erste Variable

Diskussion

- Modifikation der ersten Liste über die zweite
- `id()` liefert: alle Variablen zeigen auf das gleiche **Objekt** (die gleiche Stelle) im Speicher, in diesem Fall die gleiche Liste
- Sprechweise: `lst2` ist **Referenz** auf erste Variable
- Praktisch ein anderer Name für dasselbe Objekt

Unterscheidung zwischen mutable und immutable

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen
  - `list`, `dict`, `set`, ...

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen
  - `list`, `dict`, `set`, ...
- Gegenstück: **immutable** (unveränderbare) Datentypen

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen
  - `list`, `dict`, `set`, ...
- Gegenstück: **immutable** (unveränderbare) Datentypen
  - `int`, `float`, `complex`, `bool`, `tuple`, `str`, ...

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen
  - `list`, `dict`, `set`, ...
- Gegenstück: **immutable** (unveränderbare) Datentypen
  - `int`, `float`, `complex`, `bool`, `tuple`, `str`, ...
- Veränderung immutabler Datentypen impliziert **Neuanlegung** mit neuer Speicheradresse

Unterscheidung zwischen mutable und immutable

- In Python: Speicherung aller Daten als sogenannte **Objekte**
- Einige Objekte können Inhalt ändern, ohne Identität (Speicheradresse) zu ändern
- Sprechweise: **mutable** (veränderbare) Datentypen
  - `list`, `dict`, `set`, ...
- Gegenstück: **immutable** (unveränderbare) Datentypen
  - `int`, `float`, `complex`, `bool`, `tuple`, `str`, ...
- Veränderung immutabler Datentypen impliziert **Neuanlegung** mit neuer Speicheradresse
- Wichtig: passiert nicht bei Zuweisung eines neuen Variablennamens, sondern erst bei Änderung des Inhalts per Zugriff über den neuen Namen

Codebeispiel

Interne Speicherung mutabler Datentypen

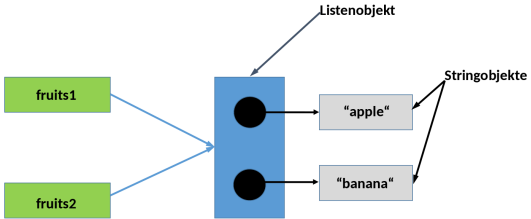
Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
```

Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
```

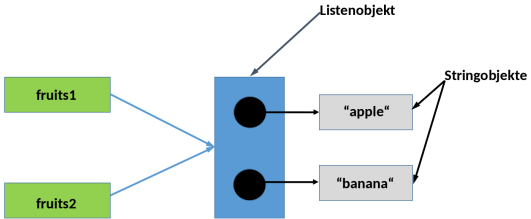
- Untersuchung mit `id()`, im Jupyter für eigene Experimente



Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
```

- Untersuchung mit `id()`, im Jupyter für eigene Experimente

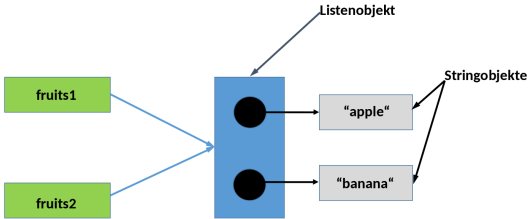


- Beide Variablen referenzieren das gleiche Objekt

Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
```

- Untersuchung mit `id()`, im Jupyter für eigene Experimente



- Beide Variablen referenzieren das gleiche Objekt
- Zugriff auf `fruits1[0]` oder `fruits2[0]` liefert dasselbe Objekt im Speicher

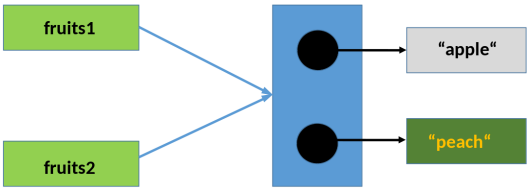
Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
fruits2[1] = "peach"
```

Interne Speicherung mutabler Datentypen

```
fruits1 = ["apple", "banana"]
fruits2 = fruits1
fruits2[1] = "peach"
```

- Beide Variablen zeigen immer noch auf die gleiche Liste



Kopieren statt Referenzieren

Kopieren statt Referenzieren

- **Kopie** der Liste nötig, um Einträge unabhängig voneinander zu bearbeiten

Kopieren statt Referenzieren

- **Kopie** der Liste nötig, um Einträge unabhängig voneinander zu bearbeiten
- **Erinnerung**: `[:]` Syntax für indizierte Listenzugriffe

Kopieren statt Referenzieren

- **Kopie** der Liste nötig, um Einträge unabhängig voneinander zu bearbeiten
- **Erinnerung**: `[:]` Syntax für indizierte Listenzugriffe
- Intuitiv sinnvoll für Kopie, da jedes Element der Ausgangsliste einmal addressiert wird

Kopieren statt Referenzieren

- **Kopie** der Liste nötig, um Einträge unabhängig voneinander zu bearbeiten
- **Erinnerung**: `[:]` Syntax für indizierte Listenzugriffe
- Intuitiv sinnvoll für Kopie, da jedes Element der Ausgangsliste einmal addressiert wird
- Also insgesamt Kopie aller Elemente, jeweils einzeln

Kopieren statt Referenzieren

- **Kopie** der Liste nötig, um Einträge unabhängig voneinander zu bearbeiten
- **Erinnerung**: `[:]` Syntax für indizierte Listenzugriffe
- Intuitiv sinnvoll für Kopie, da jedes Element der Ausgangsliste einmal addressiert wird
- Also insgesamt Kopie aller Elemente, jeweils einzeln
- **Wichtig** hier: einzelne Elemente der Beispielliste immutable (`str`)

Codebeispiel

Ein komplexeres Beispiel

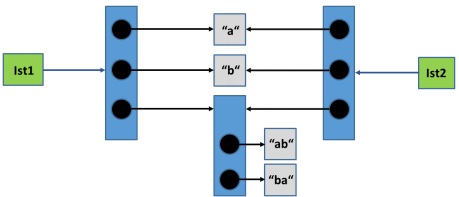
Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
```

Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
```

- Subliste ist als Liste natürlich mutable



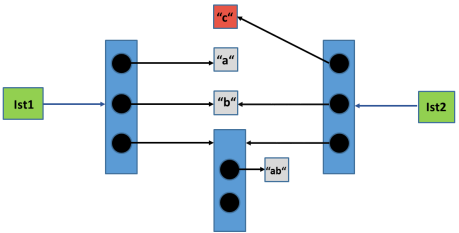
Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
lst2[0] = 'c'
```

Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
lst2[0] = 'c'
```

- Ergebnis entspricht unseren Erwartungen, `str` immutable



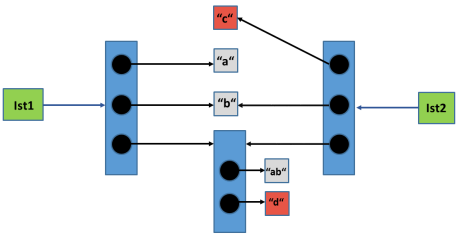
Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
lst2[0] = 'c'
lst2[2][1] = 'd'
```

Ein komplexeres Beispiel

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1[:]
lst2[0] = 'c'
lst2[2][1] = 'd'
```

- Achtung: gemeinsame Liste natürlich immer noch mutable



Deep Copies

Deep Copies

- **Vollständige Kopie** eines Objekts und aller Unterobjekte

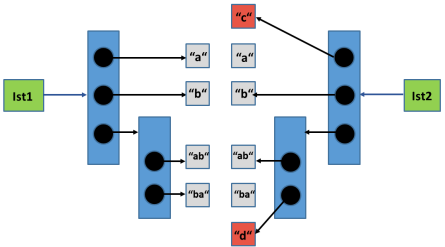
Deep Copies

- **Vollständige Kopie** eines Objekts und aller Unterobjekte
- Bibliothek `copy` mit Funktion `deepcopy()`

Codebeispiel

Diskussion

Diskussion



Diskussion

- **Wichtig**: „richtig“ oder „falsch“ ist falsche Kategorie

Diskussion

- **Wichtig**: „richtig“ oder „falsch“ ist falsche Kategorie
- Umsetzung bzw. Festlegung des Verhaltens ist nur **Konvention**

Diskussion

- **Wichtig**: „richtig“ oder „falsch“ ist falsche Kategorie
- Umsetzung bzw. Festlegung des Verhaltens ist nur **Konvention**
- Erst dann richtig oder falsch, wenn wir sie falsch verwenden

Diskussion

- **Wichtig**: „richtig“ oder „falsch“ ist falsche Kategorie
- Umsetzung bzw. Festlegung des Verhaltens ist nur **Konvention**
- Erst dann richtig oder falsch, wenn wir sie falsch verwenden
- Vollständige Kopie nicht immer wünschenswert (kann sehr teuer sein)

Diskussion

- **Wichtig**: „richtig“ oder „falsch“ ist falsche Kategorie
- Umsetzung bzw. Festlegung des Verhaltens ist nur **Konvention**
- Erst dann richtig oder falsch, wenn wir sie falsch verwenden
- Vollständige Kopie nicht immer wünschenswert (kann sehr teuer sein)
- **Empfehlung**: Verinnerlichung im Laufe des Kurses

Dictionaries und Sets

Dictionaries und Sets

- `dict` und `set` sind mutable

Dictionaries und Sets

- `dict` und `set` sind mutable
- Gleiche Konsequenzen wie bei Listen

Dictionaries und Sets

- `dict` und `set` sind mutable
- Gleiche Konsequenzen wie bei Listen
- Keine Beispiele, ausnahmsweise

Ein Wort der Vorsicht

Ein Wort der Vorsicht

- Mutable Datentypen wie Listen, Dictionaries und Mengen: **Plus-Operator** für Konkatenation

Ein Wort der Vorsicht

- Mutable Datentypen wie Listen, Dictionaries und Mengen: **Plus-Operator** für Konkatenation
- Legt immer ein neues Objekt an

Ein Wort der Vorsicht

- Mutable Datentypen wie Listen, Dictionaries und Mengen: **Plus-Operator** für Konkatenation
- Legt immer ein neues Objekt an
- Hochgradig **unintuitive Konsequenzen**

Ein Wort der Vorsicht

- Mutable Datentypen wie Listen, Dictionaries und Mengen: **Plus-Operator** für Konkatenation
- Legt immer ein neues Objekt an
- Hochgradig **unintuitive Konsequenzen**
- Sollte vermieden werden

Ein Wort der Vorsicht

- Mutable Datentypen wie Listen, Dictionaries und Mengen: **Plus-Operator** für Konkatenation
- Legt immer ein neues Objekt an
- Hochgradig **unintuitive Konsequenzen**
- Sollte vermieden werden
- Auftrennung von Zuweisung und (intendierten) Modifikation

Codebeispiel

Impressum, Danksagung und Quellen

Stiftung Innovation in der Hochschullehre

Gefördert durch die Stiftung Innovation in der Hochschullehre im Rahmen des Projekts digit@L, https://stiftung-hochschullehre.de

SimTech

Gefördert mit Mitteln der Deutschen Forschungsgemeinschaft (EXC 2075 - 390740016) im Rahmen der Exzellenzstrategie

---

Autor: Dominik Göddeke, IANS, Universität Stuttgart

CC BY

Weitere Quellen:

- Logos Universität Stuttgart, IANS, SimTech: Universität Stuttgart, alle Rechte vorbehalten
- Logo Python: https://freesvg.org/387, CC-0
- Logo Stiftung: Stiftung Innovation in der Hochschullehre, alle Rechte vorbehalten
- Logo ZOERR: Universität Tübingen, alle Rechte vorbehalten

ZOERR

Veröffentlicht auf dem Zentralen OER Repositorium Baden-Württemberg, https://www.zoerr.de