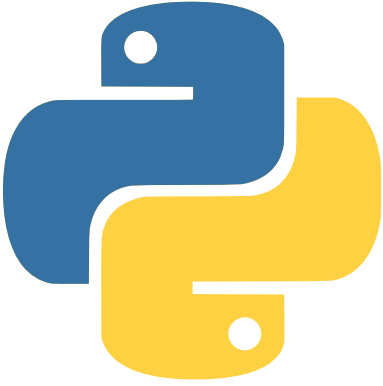
Universität Stuttgart
Projekt digit@L – BOOST. SKILLS. SUPPORT.



Programmierkurs Python

Mutable und immutable Funktionsargumente

Dominik Göddeke

IANS SimTech

Stiftung Innovation in der Hochschullehre

Mutable und immutable Funktionsargumente

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`
- Ändern unter neuem oder altem Variablennamen bewirkt **Neuanlegen eines Objekts** im Speicher, reine Zuweisung noch nicht

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`
- Ändern unter neuem oder altem Variablennamen bewirkt **Neuanlegen eines Objekts** im Speicher, reine Zuweisung noch nicht
- **Mutable**: beispielsweise `dict`, `list`, `set`

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`
- Ändern unter neuem oder altem Variablennamen bewirkt **Neuanlegen eines Objekts** im Speicher, reine Zuweisung noch nicht
- **Mutable**: beispielsweise `dict`, `list`, `set`
- Dürfen sich im Speicher ändern

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`
- Ändern unter neuem oder altem Variablennamen bewirkt **Neuanlegen eines Objekts** im Speicher, reine Zuweisung noch nicht
- **Mutable**: beispielsweise `dict`, `list`, `set`
- Dürfen sich im Speicher ändern
- Keine automatische Kopie

Erinnerung: mutable und immutable Datentypen

- Unterscheidung zwischen **veränderbaren (mutable)** und **unveränderbaren (immutable)** Datentypen
- **Immutable**: beispielsweise `int`, `tuple`, `float`, `str`
- Ändern unter neuem oder altem Variablennamen bewirkt **Neuanlegen eines Objekts** im Speicher, reine Zuweisung noch nicht
- **Mutable**: beispielsweise `dict`, `list`, `set`
- Dürfen sich im Speicher ändern
- Keine automatische Kopie
- **In diesem Abschnitt: Wiederholung, und einige Subtilitäten** bei der Veränderbarkeit von Variablen als Funktionsargumente

Codebeispiele

Fortsetzung

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst
- Sondern ändert nur die Adresse, so dass `a` auf die Zahl `42` zeigt

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst
- Sondern ändert nur die Adresse, so dass `a` auf die Zahl `42` zeigt
- **Das Objekt selbst ändert sich nicht!**

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst
- Sondern ändert nur die Adresse, so dass `a` auf die Zahl `42` zeigt
- **Das Objekt selbst ändert sich nicht!**
- Im Beispiel: Zahl `1` im Speicher nicht mehr erreichbar

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst
- Sondern ändert nur die Adresse, so dass `a` auf die Zahl `42` zeigt
- **Das Objekt selbst ändert sich nicht!**
- Im Beispiel: Zahl `1` im Speicher nicht mehr erreichbar
- Hintergrundprozess, sogenannter **Python Garbage Collector**, räumt „irgendwann“ auf, spätestens zum Programmende

Erinnerung: mutable und immutable Datentypen

- Beispiel zeigt: **immutable** Objekte nur einmal im Speicher ablegt
- Anweisung `a += 41` ändert nicht das Objekt (die Zahl `1`) selbst
- Sondern ändert nur die Adresse, so dass `a` auf die Zahl `42` zeigt
- **Das Objekt selbst ändert sich nicht!**
- Im Beispiel: Zahl `1` im Speicher nicht mehr erreichbar
- Hintergrundprozess, sogenannter **Python Garbage Collector**, räumt „irgendwann“ auf, spätestens zum Programmende
- Gegenteil: Änderung des Inhalts bei **mutable** Datentypen, nicht der Variable

Codebeispiele

Ein ausführlicheres Beispiel

Ein ausführlicheres Beispiel

- Mutable Datentypen erlauben potentiellen **Laufzeit- und Speicherplatz-Gewinn**

Ein ausführlicheres Beispiel

- Mutable Datentypen erlauben potentiellen **Laufzeit- und Speicherplatz-Gewinn**
- Im Beispiel: 1.000.000 zufällige Ziffern in einem String

Ein ausführlicheres Beispiel

- Mutable Datentypen erlauben potentiellen **Laufzeit- und Speicherplatz-Gewinn**
- Im Beispiel: 1.000.000 zufällige Ziffern in einem String
- Einmal mit String-Konkatenation (immutable, also viele Kopien)

Ein ausführlicheres Beispiel

- Mutable Datentypen erlauben potentiellen **Laufzeit- und Speicherplatz-Gewinn**
- Im Beispiel: 1.000.000 zufällige Ziffern in einem String
- Einmal mit String-Konkatenation (immutable, also viele Kopien)
- Einmal mit Umweg über Liste (mutable) und Konvertierung in String am Ende

Ein ausführlicheres Beispiel

- Mutable Datentypen erlauben potentiellen **Laufzeit- und Speicherplatz-Gewinn**
- Im Beispiel: 1.000.000 zufällige Ziffern in einem String
- Einmal mit String-Konkatenation (immutable, also viele Kopien)
- Einmal mit Umweg über Liste (mutable) und Konvertierung in String am Ende
- Laufzeitmessungen

So verhält sich Python

Immutable/Mutable Datentypen als Funktionsargument

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht
  - Zuweisung `x += 4` legt neues **lokales** Objekt an

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht
  - Zuweisung `x += 4` legt neues **lokales** Objekt an
  - Damit Überschreiben der Variable `x`

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht
  - Zuweisung `x += 4` legt neues **lokales** Objekt an
  - Damit Überschreiben der Variable `x`
  - Ausgangswert von `x` in der Funktion nicht mehr erreichbar

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht
  - Zuweisung `x += 4` legt neues **lokales** Objekt an
  - Damit Überschreiben der Variable `x`
  - Ausgangswert von `x` in der Funktion nicht mehr erreichbar
- Weil außerhalb der Funktion `a` (globale Variable) noch auf den Ausgangswert zeigt, dort noch erreichbar

Immutable/Mutable Datentypen als Funktionsargument

```
def my_function(x):
    x += 4 # ebenso für x = x + 4 wegen immutable
    return x

a = 2
b = my_function(a)
```

- **Probieren Sie gerne aus, was passiert!**
- Variable ändert sich offensichtlich nicht
  - Zuweisung `x += 4` legt neues **lokales** Objekt an
  - Damit Überschreiben der Variable `x`
  - Ausgangswert von `x` in der Funktion nicht mehr erreichbar
- Weil außerhalb der Funktion `a` (globale Variable) noch auf den Ausgangswert zeigt, dort noch erreichbar
- Fazit: tatsächlich **intuitiv im Fall immutable**

Immutable/Mutable Datentypen als Funktionsargument

- **Mutable Funktionsargumente**: potentiell deutlich unintuitiver

Immutable/Mutable Datentypen als Funktionsargument

- **Mutable Funktionsargumente**: potentiell deutlich unintuitiver
- Wichtig: Verständnis lokaler und globaler Variablen

Immutable/Mutable Datentypen als Funktionsargument

- **Mutable Funktionsargumente**: potentiell deutlich unintuitiver
- Wichtig: Verständnis lokaler und globaler Variablen
- Wichtig: Verständnis, wann bei mutable Datentypen tatsächlich ein neues Objekt angelegt wird

Immutable/Mutable Datentypen als Funktionsargument

- **Mutable Funktionsargumente**: potentiell deutlich unintuitiver
- Wichtig: Verständnis lokaler und globaler Variablen
- Wichtig: Verständnis, wann bei mutable Datentypen tatsächlich ein neues Objekt angelegt wird
- Dann am Ende doch nicht unintuitiv

Codebeispiele

Experimente mit Listenänderungen

Experimente mit Listenänderungen

- Erinnerung: Listenänderungen mit `...+=...`: kein neues Objekt

Experimente mit Listenänderungen

- Erinnerung: Listenänderungen mit `...+=...`: kein neues Objekt
- Erinnerung: Listenänderungen mit `... = ... + ...`: neues Objekt

Experimente mit Listenänderungen

- Erinnerung: Listenänderungen mit `...+=...`: kein neues Objekt
- Erinnerung: Listenänderungen mit `... = ... + ...`: neues Objekt
  - Vorheriges Beispiel war Variante: `... = ...`

Codebeispiele

Fazit

Fazit

- Großer Unterschied zwischen `+=` oder `= ... + ...` (oder `= ...`, ist dasselbe)

Fazit

- Großer Unterschied zwischen `+=` oder `= ... + ...` (oder `= ...`, ist dasselbe)
- Bei Listen wie im Beispiel, aber auch bei Dictionaries und Sets

Fazit

- Großer Unterschied zwischen `+=` oder `= ... + ...` (oder `= ...`, ist dasselbe)
- Bei Listen wie im Beispiel, aber auch bei Dictionaries und Sets
- In einer früheren Lerneinheit: bereits wegen Unintuitivität davon abgeraten

Fazit

- Großer Unterschied zwischen `+=` oder `= ... + ...` (oder `= ...`, ist dasselbe)
- Bei Listen wie im Beispiel, aber auch bei Dictionaries und Sets
- In einer früheren Lerneinheit: bereits wegen Unintuitivität davon abgeraten
- Explizit: **Vermeiden Sie + für mutables, solange Sie die Details nicht verstanden haben!**

Fazit

- Großer Unterschied zwischen `+=` oder `= ... + ...` (oder `= ...`, ist dasselbe)
- Bei Listen wie im Beispiel, aber auch bei Dictionaries und Sets
- In einer früheren Lerneinheit: bereits wegen Unintuitivität davon abgeraten
- Explizit: **Vermeiden Sie + für mutables, solange Sie die Details nicht verstanden haben!**
- **Saubere Lösung**: Verwendung einer **Methode** die über die Referenz in der Argumentliste einer Funktion aufgerufen wird

Sauberes Codebeispiel

Letzte Komplikationen

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

ZOERR

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