合肥大學
HEFEI UNIVERSITY

Programming with Python

34. Iteration

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO)
School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-01-29

Programming with Python

Dies ist ein Kurs über das Programmieren mit der Programmiersprache Python an der Universität Hefei (合肥大学).

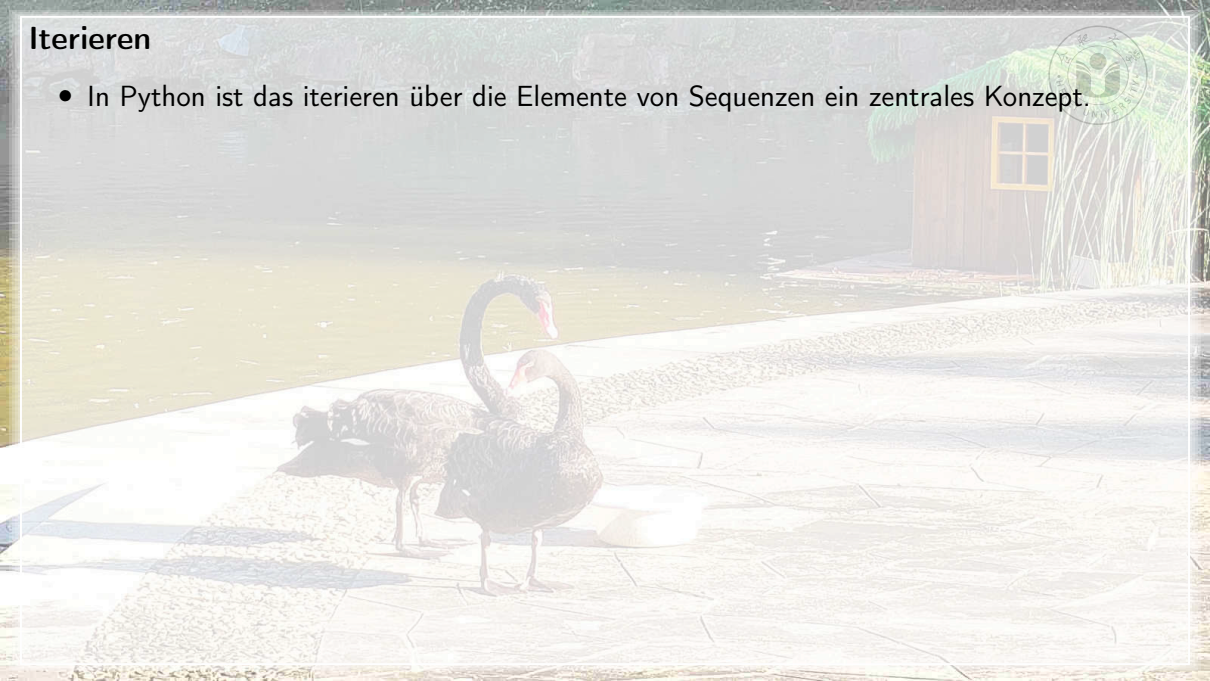
Die Webseite mit dem Lehrmaterial dieses Kurses ist https://thomasweise.github.io/programmingWithPython (siehe auch den QR-Kode unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter https://github.com/thomasWeise/programmingWithPythonCode.

Outline

Einleitung

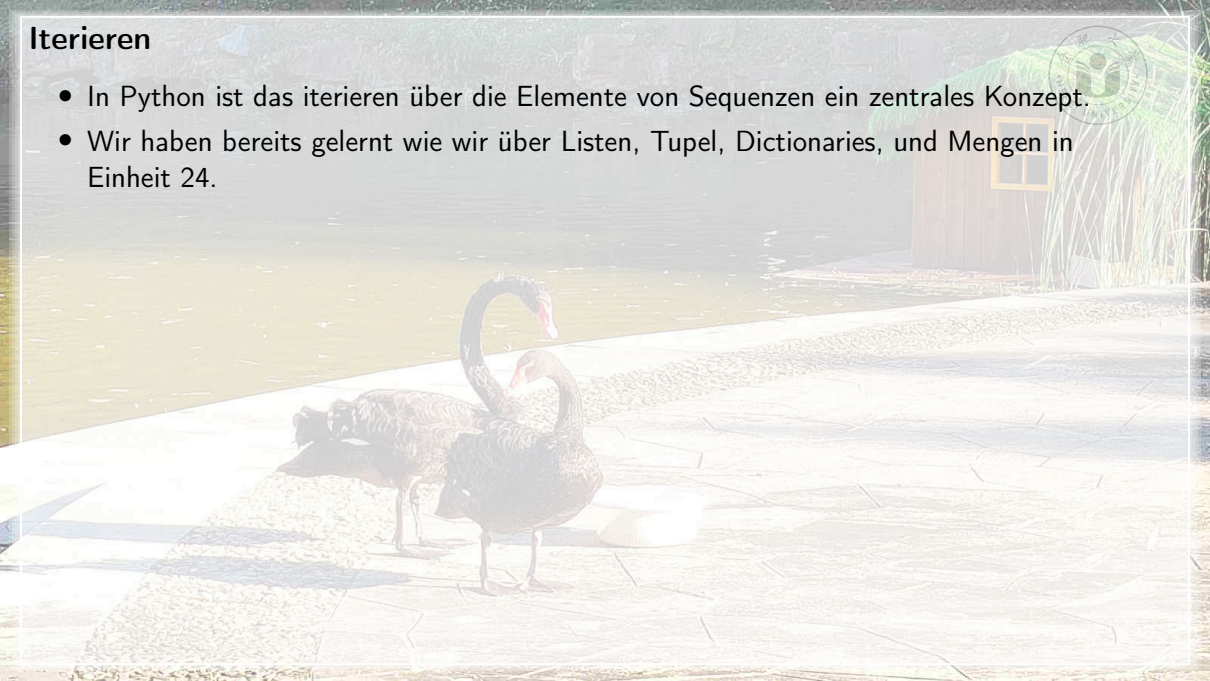
Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.



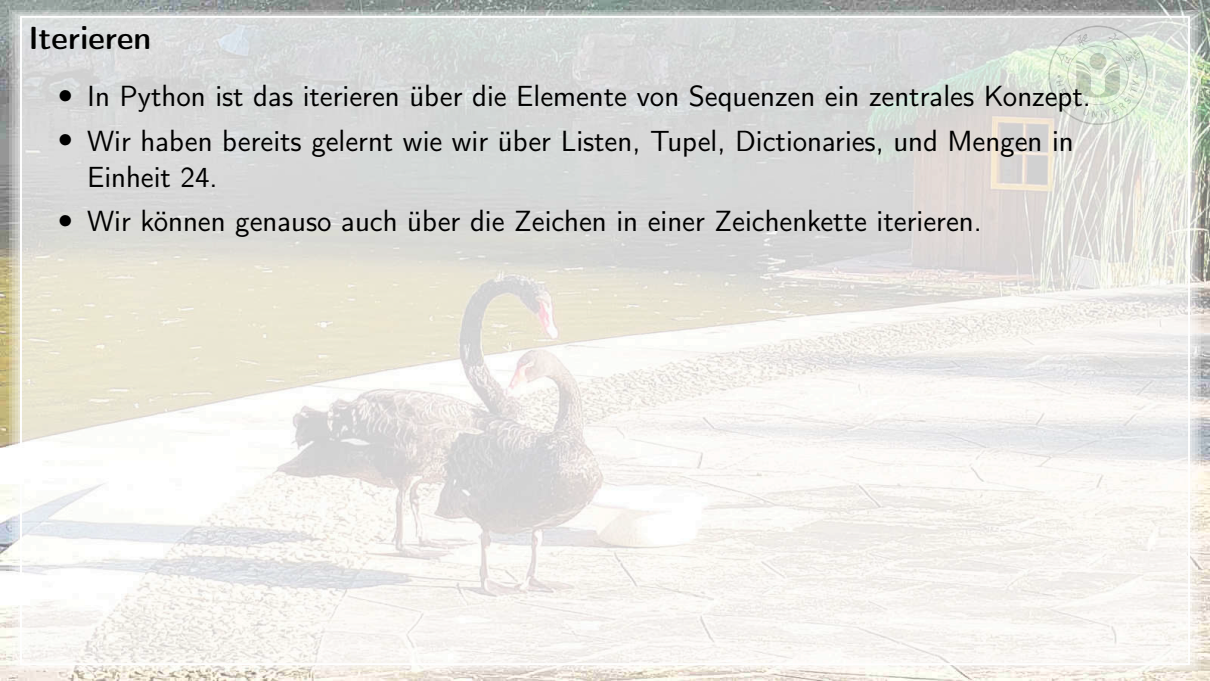
Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.



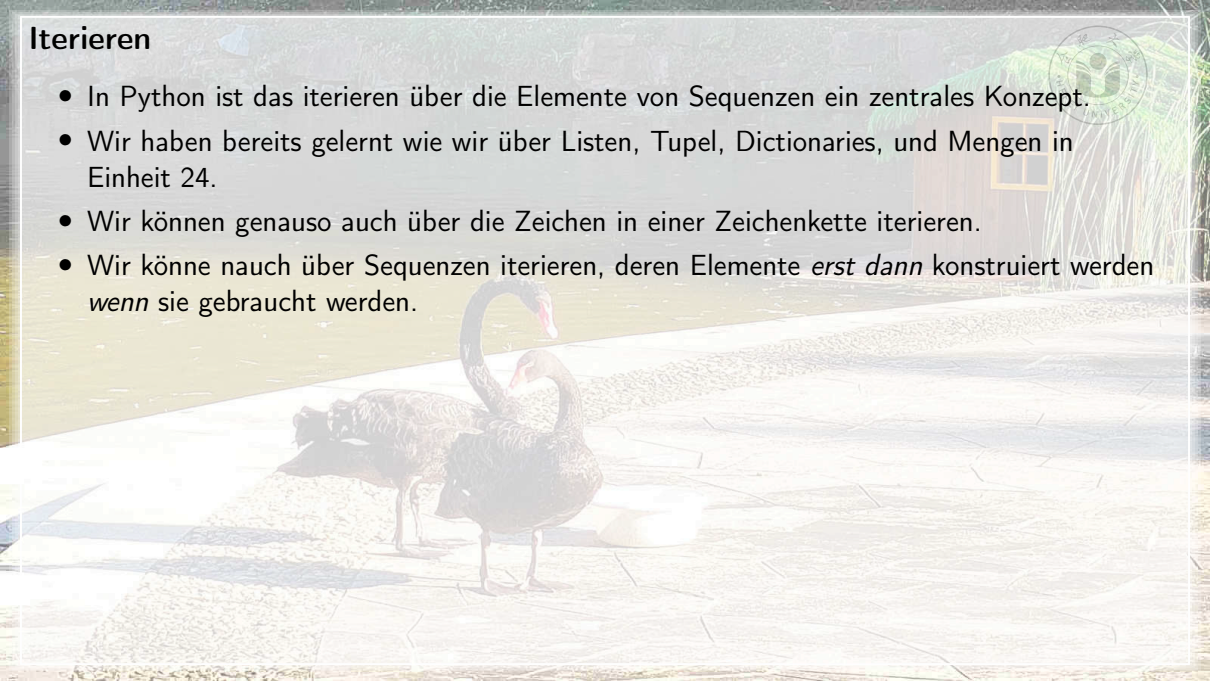
Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.



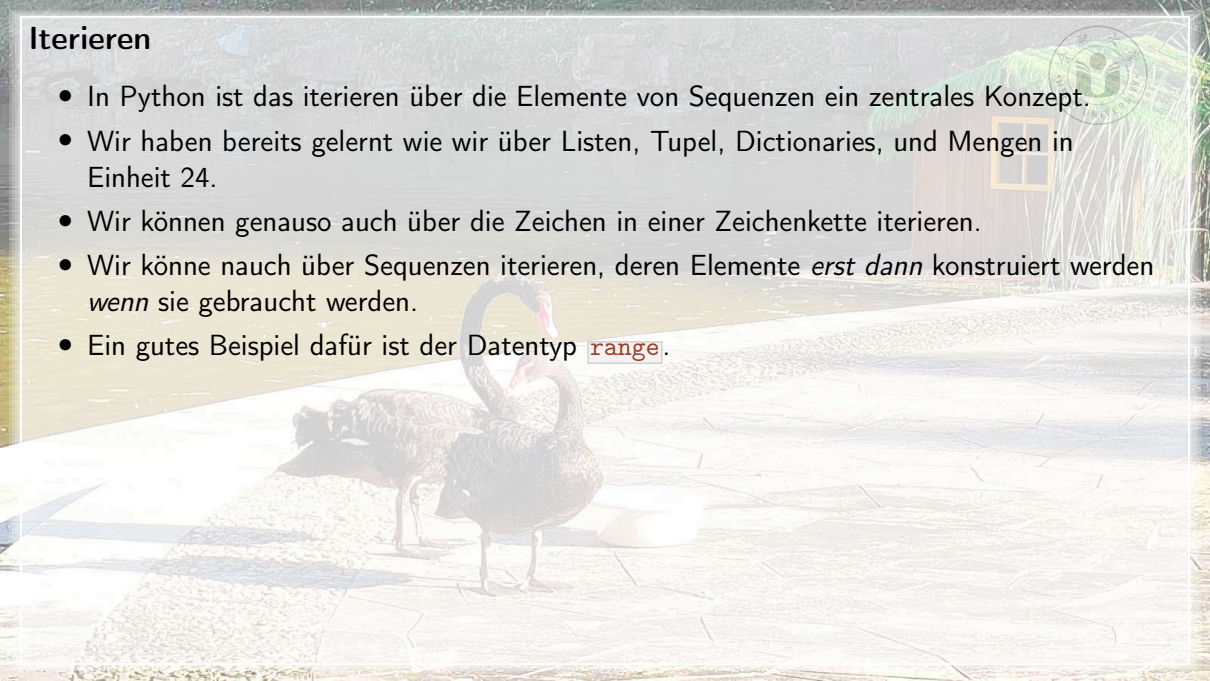
Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.



Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.



Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.
- Wir können über 1 000 000 000 000 `int`-Elemente mit `range(100_000_000_000_000)` iterieren.

Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.
- Wir können über 1 000 000 000 000 `int`-Elemente mit `range(100_000_000_000_000)` iterieren.
- Soviele Ganzzahlen passen vielleicht gar nicht in den Speicher...

Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.
- Wir können über 1 000 000 000 000 `int`-Elemente mit `range(100_000_000_000_000)` iterieren.
- Soviele Ganzzahlen passen vielleicht gar nicht in den Speicher...
- Stattdessen werden diese eine nach der Anderen angelegt und Bereitgestellt so wie sie benötigt werden.

Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.
- Wir können über 1 000 000 000 000 `int`-Elemente mit `range(100_000_000_000_000)` iterieren.
- Soviele Ganzzahlen passen vielleicht gar nicht in den Speicher...
- Stattdessen werden diese eine nach der Anderen angelegt und Bereitgestellt so wie sie benötigt werden.
- Aus Sicht des Programmierers können wir über `ranges` und `lists` genau gleich iterieren.

Iterieren

- In Python ist das iterieren über die Elemente von Sequenzen ein zentrales Konzept.
- Wir haben bereits gelernt wie wir über Listen, Tupel, Dictionaries, und Mengen in Einheit 24.
- Wir können genauso auch über die Zeichen in einer Zeichenkette iterieren.
- Wir könne nauch über Sequenzen iterieren, deren Elemente *erst dann* konstruiert werden *wenn* sie gebraucht werden.
- Ein gutes Beispiel dafür ist der Datentyp `range`.
- Wir können über 1 000 000 000 000 `int`-Elemente mit `range(100_000_000_000_000)` iterieren.
- Soviele Ganzzahlen passen vielleicht gar nicht in den Speicher...
- Stattdessen werden diese eine nach der Anderen angelegt und Bereitgestellt so wie sie benötigt werden.
- Aus Sicht des Programmierers können wir über `ranges` und `lists` genau gleich iterieren.
- Viele Objekte in Python unterstützen Iterationen.

Kollektionen aus Iterationen erstellen

- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.

Kollektionen aus Iterationen erstellen

- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.
- Die Datentypen `list`, `tuple`, `set`, und `dict` können als Funktionen verwendet werden, die eine Sequenz von Elementen als Parameter akzeptiert und dann eine Instanz des entsprechenden Datentyps erstellt.

Kollektionen aus Iterationen erstellen

- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.
- Die Datentypen `list`, `tuple`, `set`, und `dict` können als Funktionen verwendet werden, die eine Sequenz von Elementen als Parameter akzeptiert und dann eine Instanz des entsprechenden Datentyps erstellt.
- Wir wissen, dass `[1, 2, 2, 3]` ein Listenliteral mit den entsprechenden Elementen ist.

Kollektionen aus Iterationen erstellen

- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.
- Die Datentypen `list`, `tuple`, `set`, und `dict` können als Funktionen verwendet werden, die eine Sequenz von Elementen als Parameter akzeptiert und dann eine Instanz des entsprechenden Datentyps erstellt.
- Wir wissen, dass `[1, 2, 2, 3]` ein Listenliteral mit den entsprechenden Elementen ist.
- Übergeben wir diese Liste an die `set`-Funktion/datatype, schreiben wir also `set([1, 2, 2, 3])`, dann bekommen wir die Menge `{1, 2, 3}`.

Kollektionen aus Iterationen erstellen

- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.
- Die Datentypen `list`, `tuple`, `set`, und `dict` können als Funktionen verwendet werden, die eine Sequenz von Elementen als Parameter akzeptiert und dann eine Instanz des entsprechenden Datentyps erstellt.
- Wir wissen, dass `[1, 2, 2, 3]` ein Listenliteral mit den entsprechenden Elementen ist.
- Übergeben wir diese Liste an die `set`-Funktion/datatype, schreiben wir also `set([1, 2, 2, 3])`, dann bekommen wir die Menge `{1, 2, 3}`.
- Viele Kollektions-Datenstrukturen haben Methoden, mit denen wir sie verändern können, in dem wir andere Kollektionen als Argumente eingeben.

Kollektionen aus Iterationen erstellen

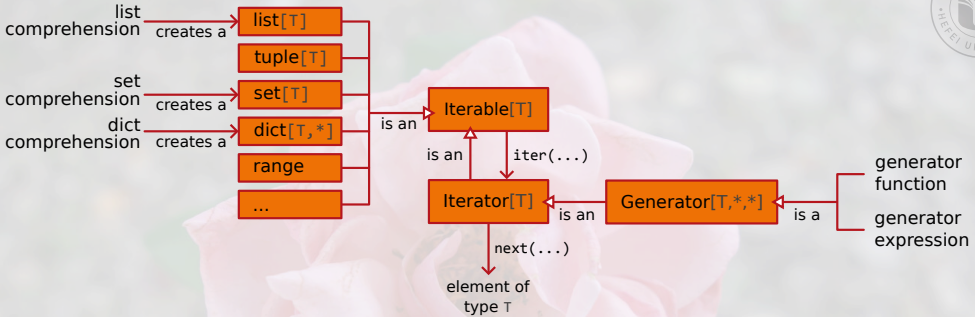
- Wir können auch viele Arten von Kollektion von Sequenzen von Elementen erstellen.
- Die Datentypen `list`, `tuple`, `set`, und `dict` können als Funktionen verwendet werden, die eine Sequenz von Elementen als Parameter akzeptiert und dann eine Instanz des entsprechenden Datentyps erstellt.
- Wir wissen, dass `[1, 2, 2, 3]` ein Listenliteral mit den entsprechenden Elementen ist.
- Übergeben wir diese Liste an die `set`-Funktion/datatype, schreiben wir also `set([1, 2, 2, 3])`, dann bekommen wir die Menge `{1, 2, 3}`.
- Viele Kollektions-Datenstrukturen haben Methoden, mit denen wir sie verändern können, in dem wir andere Kollektionen als Argumente eingeben.
- Z. B. das Aufrufen von `l.extend({1, 2, 3})` hängt die Elemente `1`, `2`, und `3` an eine Liste `l` an.

Iterator, Iterable, Generator und Comprehension

Iterationen und Ähnliches

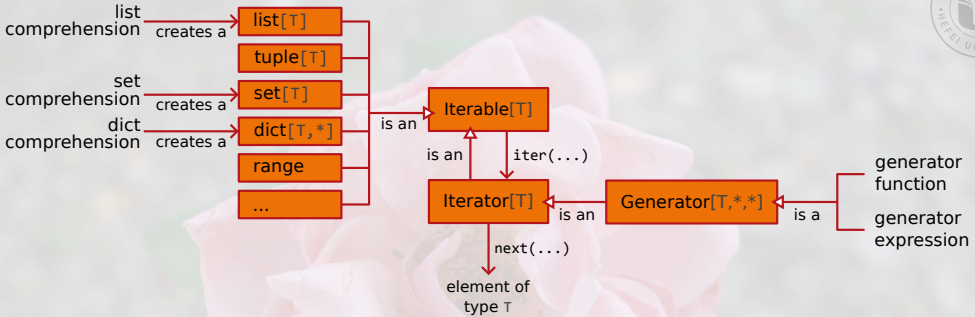
- In sehr vielen Situationen Transformieren, Verarbeiten, oder Erstellen wir Sequenzen von Daten.

Iterationen und Ähnliches



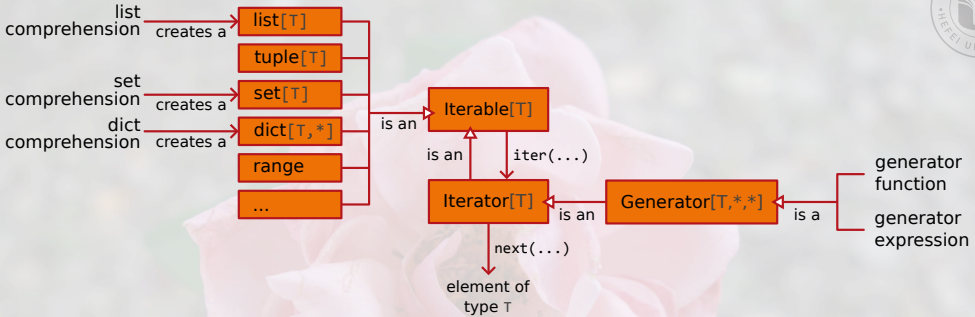
- In sehr vielen Situationen Transformieren, Verarbeiten, oder Erstellen wir Sequenzen von Daten.
- In Python gibt es viele verschiedene Manifestationen vom *Iterieren* über Objekte die *iterierbar* sind.

Iterationen und Ähnliches



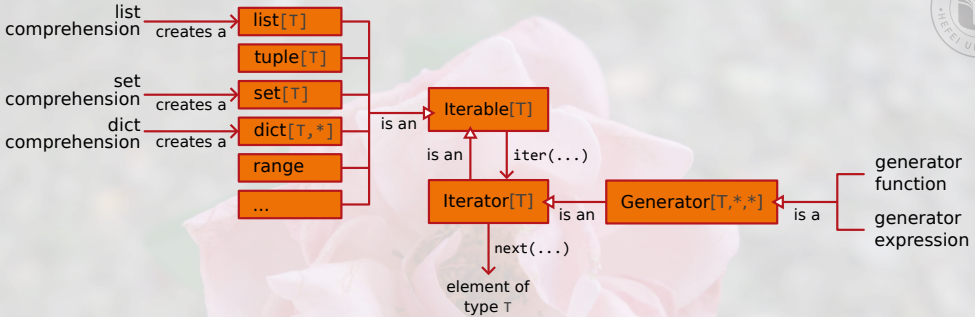
- In sehr vielen Situationen Transformieren, Verarbeiten, oder Erstellen wir Sequenzen von Daten.
- In Python gibt es viele verschiedene Manifestationen vom *Iterieren* über Objekte die *iterierbar* sind.
- Das primitivste Konzept ist der `Iterator`[4,12,28].

Iterationen und Ähnliches



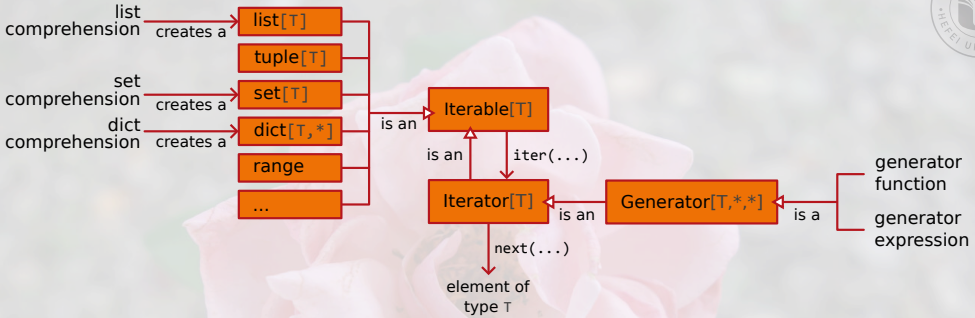
- In sehr vielen Situationen Transformieren, Verarbeiten, oder Erstellen wir Sequenzen von Daten.
- In Python gibt es viele verschiedene Manifestationen vom *Iterieren* über Objekte die *iterierbar* sind.
- Das primitivste Konzept ist der `Iterator`[4,12,28].
- Das ist ein Object das eine Iteration über die Elemente einer Sequenz repräsentiert.

Iterationen und Ähnliches



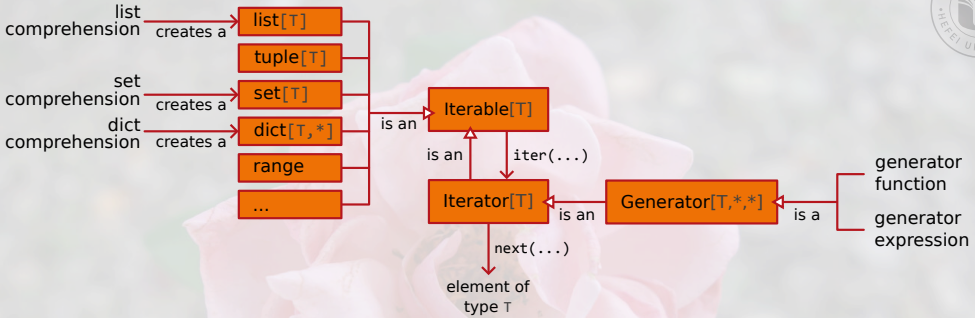
- In Python gibt es viele verschiedene Manifestationen vom *Iterieren* über Objekte die *iterierbar* sind.
- Das primitivste Konzept ist der `Iterator`[4,12,28].
- Das ist ein Object das eine Iteration über die Elemente einer Sequenz repräsentiert.
- Wenn wir ein `Iterator`-Object `u` haben, dann können wir das nächste Element der Sequenz für die es steht via `next(u)` erhalten.

Iterationen und Ähnliches



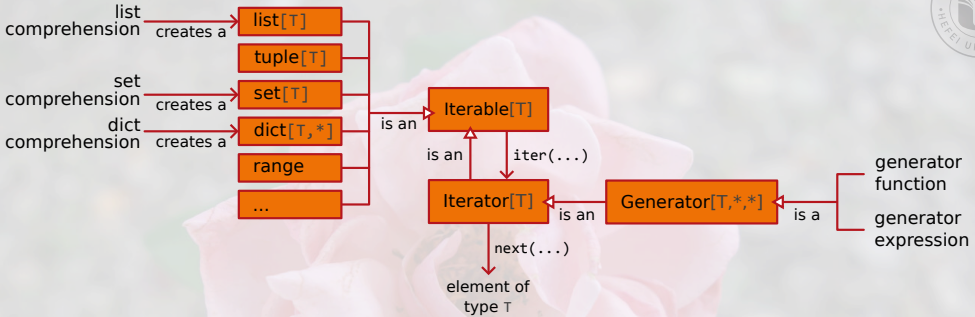
- Das primitivste Konzept ist der `Iterator`[4,12,28].
- Das ist ein Object das eine Iteration über die Elemente einer Sequenz repräsentiert.
- Wenn wir ein `Iterator`-Object `u` haben, dann können wir das nächste Element der Sequenz für die es steht via `next(u)` erhalten.
- Gibt es kein nächstes Element, dann löst dies eine `StopIteration`-Ausnahme aus.

Iterationen und Ähnliches



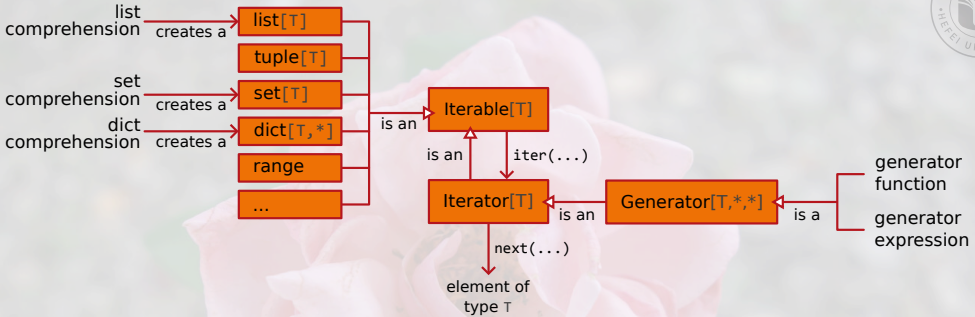
- Das primitivste Konzept ist der `Iterator`[4,12,28].
- Das ist ein Object das eine Iteration über die Elemente einer Sequenz repräsentiert.
- Wenn wir ein `Iterator`-Object `u` haben, dann können wir das nächste Element der Sequenz für die es steht via `next(u)` erhalten.
- Gibt es kein nächstes Element, dann löst dies eine `StopIteration`-Ausnahme aus.
- Solche Iteratoren sind „Einweg-Objekte“, wir können sie nur einmal benutzen.

Iterationen und Ähnliches



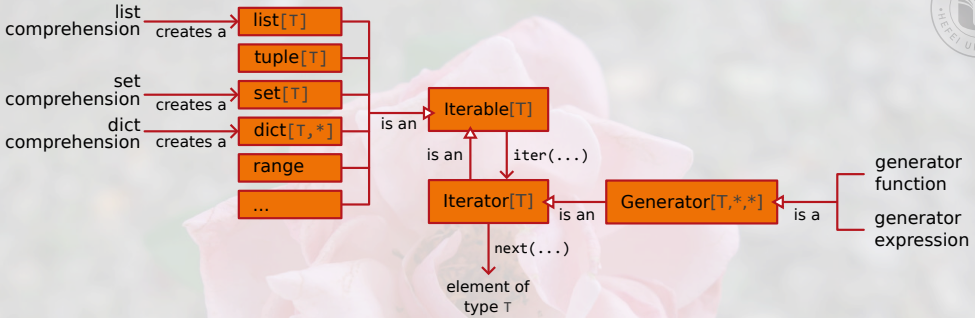
- Wenn wir ein `Iterator`-Object `u` haben, dann können wir das nächste Element der Sequenz für die es steht via `next(u)` erhalten.
- Gibt es kein nächstes Element, dann löst dies eine `StopIteration`-Ausnahme aus.
- Solche Iteratoren sind „Einweg-Objekte“, wir können sie nur einmal benutzen.
- Eine `for`-Schleife konsumiert die Elemente eines `Iterator` bis die `StopIteration` ausgelöst wird.

Iterationen und Ähnliches



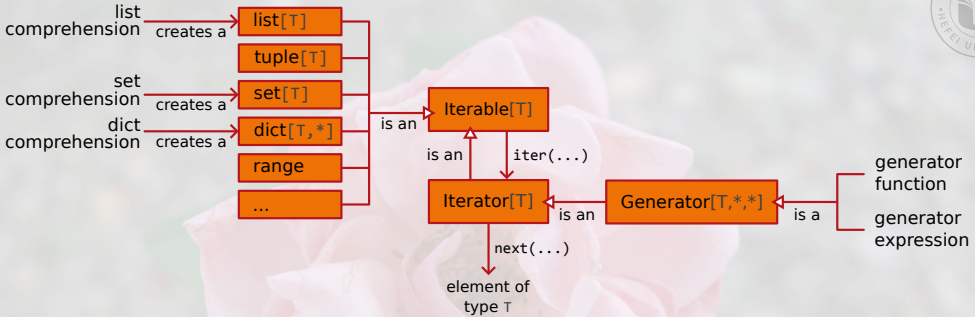
- Gibt es kein nächstes Element, dann löst dies eine `StopIteration`-Ausnahme aus.
- Solche Iteratoren sind „Einweg-Objekte“, wir können sie nur einmal benutzen.
- Eine `for`-Schleife konsumiert die Elemente eines `Iterator` bis die `StopIteration` ausgelöst wird.
- `Generator`-Funktionen und Ausdrücke sind Spezialfälle von `Iterator` und erlauben uns mehr Kontrolle bzw. eine einfacherere Syntax für das definieren von Elementsequenzen.

Iterationen und Ähnliches



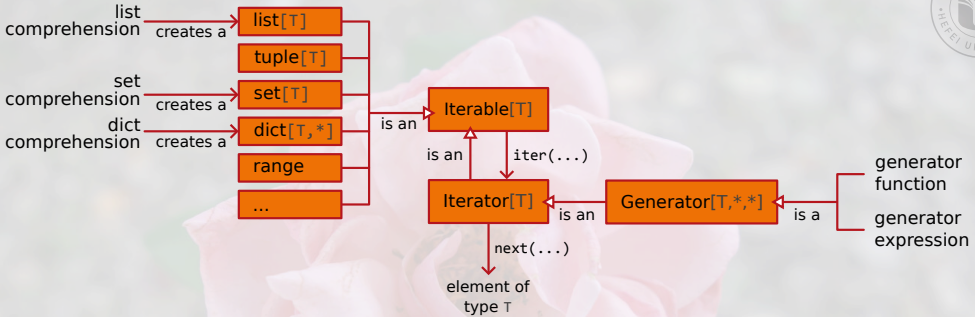
- Solche Iteratoren sind „Einweg-Objekte“, wir können sie nur einmal benutzen.
- Eine `for`-Schleife konsumiert die Elemente eines `Iterator` bis die `StopIteration` ausgelöst wird.
- `Generator`-Funktionen und Ausdrücke sind Spezialfälle von `Iterator` und erlauben uns mehr Kontrolle bzw. eine einfacherere Syntax für das definieren von Elementsequenzen.
- Viele Datenstruktueren wir Kollektionen erlauben es uns, so oft wie wir wollen über ihre Elemente zu iterieren.

Iterationen und Ähnliches



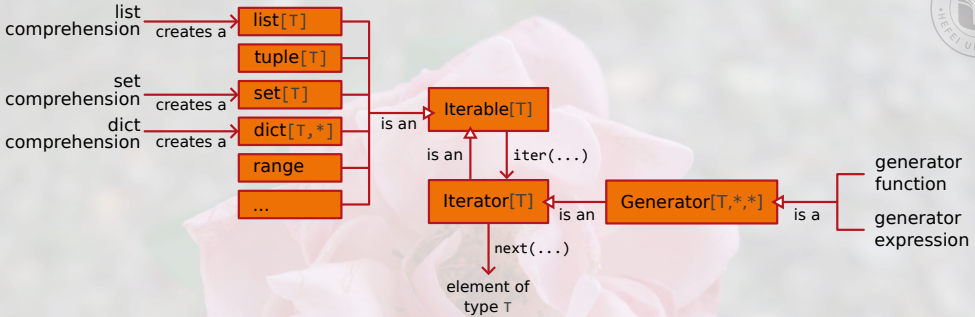
- Eine `for`-Schleife konsumiert die Elemente eines `Iterator` bis die `StopIteration` ausgelöst wird.
- `Generator`-Funktionen und Ausdrücke sind Spezialfälle von `Iterator` und erlauben uns mehr Kontrolle bzw. eine einfacherere Syntax für das definieren von Elementsequenzen.
- Viele Datenstruktueren wir Kollektionen erlauben es uns, so oft wie wir wollen über ihre Elemente zu iterieren.
- Sie alle sind Instanzen des `Iterable`-Iterfaces[4,11].

Iterationen und Ähnliches



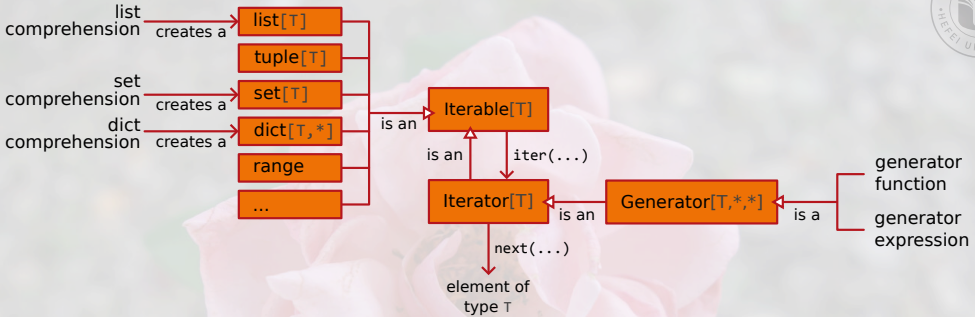
- Eine `for`-Schleife konsumiert die Elemente eines `Iterator` bis die `StopIteration` ausgelöst wird.
- `Generator`-Funktionen und Ausdrücke sind Spezialfälle von `Iterator` und erlauben uns mehr Kontrolle bzw. eine einfacherere Syntax für das definieren von Elementsequenzen.
- Viele Datenstruktueren wir Kollektionen erlauben es uns, so oft wie wir wollen über ihre Elemente zu iterieren.
- Sie alle sind Instanzen des `Iterable`-Iterfaces[4,11].

Iterationen und Ähnliches



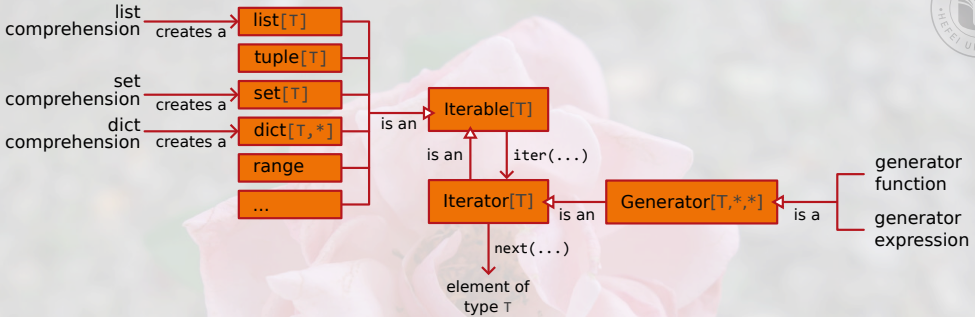
- `Generator`-Funktionen und Ausdrücke sind Spezialfälle von `Iterator` und erlauben uns mehr Kontrolle bzw. eine einfacherere Syntax für das definieren von Elementsequenzen.
- Viele Datenstruktueren wir Kollektionen erlauben es uns, so oft wie wir wollen über ihre Elemente zu iterieren.
- Sie alle sind Instanzen des `Iterable`-Iterfaces[4,11].
- Wir können `iter(coll)` für eine Kollektion `coll`, die dieses Interface implementiert, aufrufen, und wir bekommen einen `Iterator`.

Iterationen und Ähnliches



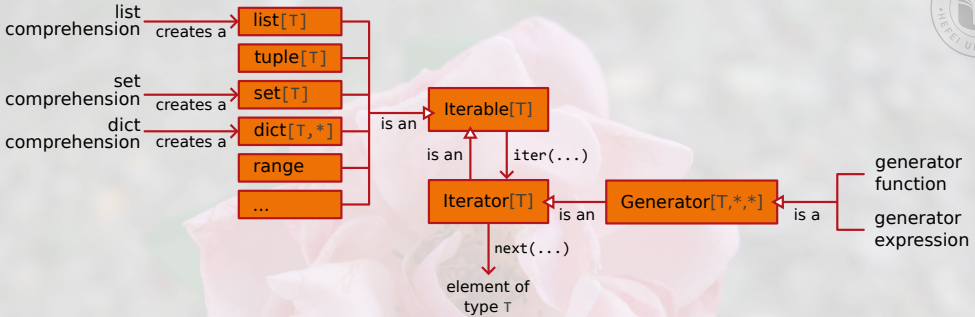
- Viele Datenstruktueren wir Kollektionen erlauben es uns, so oft wie wir wollen über ihre Elemente zu iterieren.
- Sie alle sind Instanzen des `Iterable`-Iterfaces[4,11].
- Wir können `iter(coll)` für eine Kollektion `coll`, die dieses Interface implementiert, aufrufen, und wir bekommen einen `Iterator`.
- Wann immer wir über z. B. eine Liste iterieren, dann wird erst auf diese Art ein `Iterator` erzeugt.

Iterationen und Ähnliches



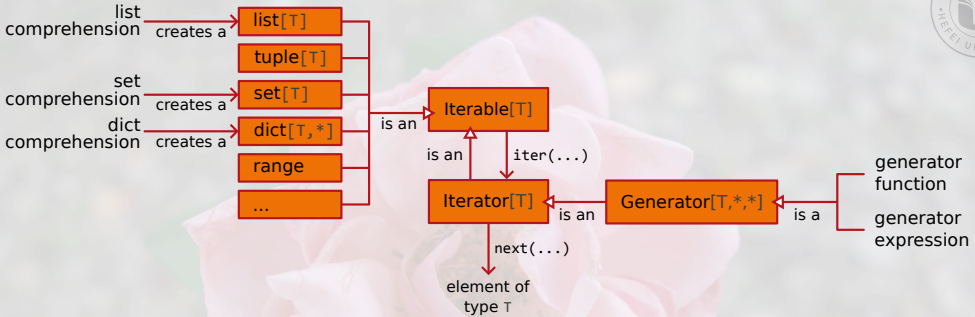
- Wann immer wir über z. B. eine Liste iterieren, dann wird erst auf diese Art ein `Iterator` erzeugt.
- Als Randnotiz sei gesagt, dass wir den Operator `iter` nicht nur auf `Iterables`, sondern auch auf `Iterator` anwenden können.[4,12].
- Wenn wir das tun, dann liefert er einfach den `Iterator` gleich wieder zurück.

Iterationen und Ähnliches



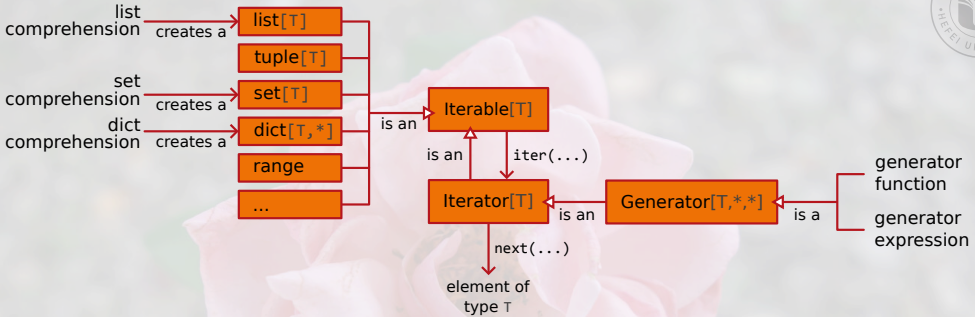
- Als Randnotiz sei gesagt, dass wir den Operator `iter` nicht nur auf `Iterables`, sondern auch auf `Iterator` anwenden können.[4,12].
- Wenn wir das tun, dann liefert er einfach den `Iterator` gleich wieder zurück.
- Damit können alle APIs, die einen `Iterable` als Parameter erfordern und diesen nur einmal brauchen auch einen `Iterator` akzeptieren.

Iterationen und Ähnliches



- Wenn wir das tun, dann liefert er einfach den `Iterator` gleich wieder zurück.
- Damit können alle APIs, die einen `Iterable` als Parameter erfordern und diesen nur einmal brauchen auch einen `Iterator` akzeptieren.
- Wir können auch Kollektionen wir Listen, Mengen, oder Dictionaries durch so genannte *comprehension* erstellen, wobei wir im Grunde eine `for`-Schleife *in* das entsprechende Literal schreiben.

Iterationen und Ähnliches



- Damit können alle APIs, die einen `Iterable` als Parameter erfordern und diesen nur einmal brauchen auch einen `Iterator` akzeptieren.
- Wir können auch Kollektionen wir Listen, Mengen, oder Dictionaries durch so genannte *comprehension* erstellen, wobei wir im Grunde eine `for`-Schleife *in* das entsprechende Literal schreiben.
- Alles das werden wir uns nach und nach anschauen.

Iteratoren

Iterator über Listen

- Ein Objekt das uns erlaubt auf seine Elemente eins nach dem Anderen, also iterativ, ist eine Instanz von `typing.Iterable`.

Iterator über Listen

- Ein Objekt das uns erlaubt auf seine Elemente eins nach dem Anderen, also iterativ, ist eine Instanz von `typing.Iterable`.
- Die eigentliche Iteration findet dann mit Hilfe eines `typing.Iterator` statt[4,12,28].

Iterator über Listen

- Ein Objekt das uns erlaubt auf seine Elemente eins nach dem Anderen, also iterativ, ist eine Instanz von `typing.Iterable`.
- Die eigentliche Iteration findet dann mit Hilfe eines `typing.Iterator` statt[4,12,28].
- Diese Unterscheidung ist notwendig, weil wir normalerweise erlauben wollen, beliebig oft über den Inhalt von Objekten zu iterieren.

Iterator über Listen

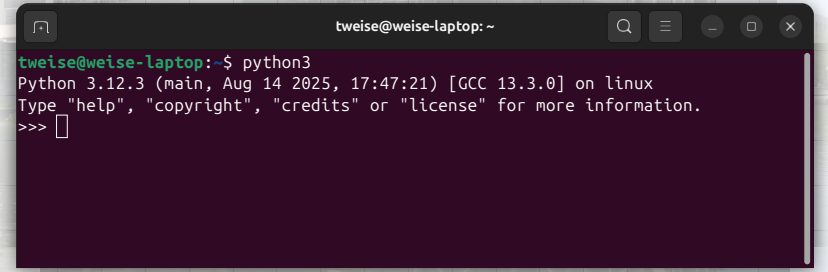
- Die eigentliche Iteration findet dann mit Hilfe eines `typing.Iterator` statt[4,12,28].
- Diese Unterscheidung ist notwendig, weil wir normalerweise erlauben wollen, beliebig oft über den Inhalt von Objekten zu iterieren.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).

Iterator über Listen

- Diese Unterscheidung ist notwendig, weil wir normalerweise erlauben wollen, beliebig oft über den Inhalt von Objekten zu iterieren.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.

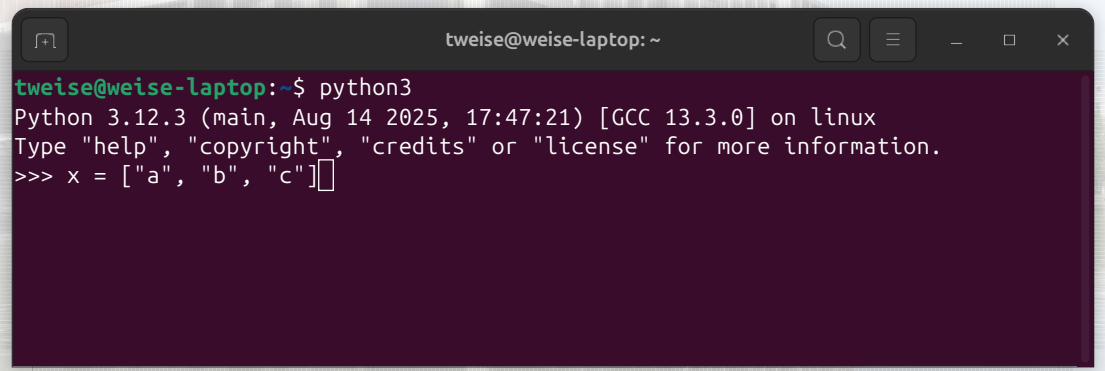
Iterator über Listen

- Diese Unterscheidung ist notwendig, weil wir normalerweise erlauben wollen, beliebig oft über den Inhalt von Objekten zu iterieren.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl+Alt+T, unter Microsoft Windows durch Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir sind nun im Interpreter.



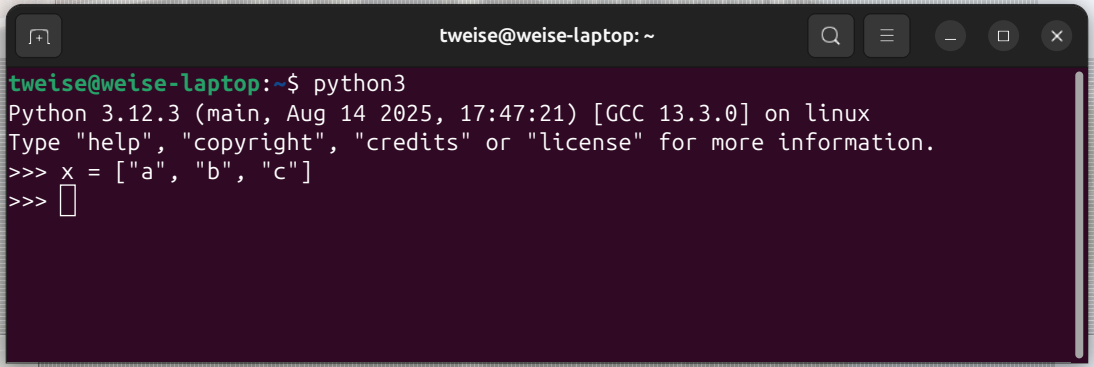
Iterator über Listen

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl+Alt+T, unter Microsoft Windows durch Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Sagen wir, wir haben eine Liste `x = ["a", "b", "c"]`.



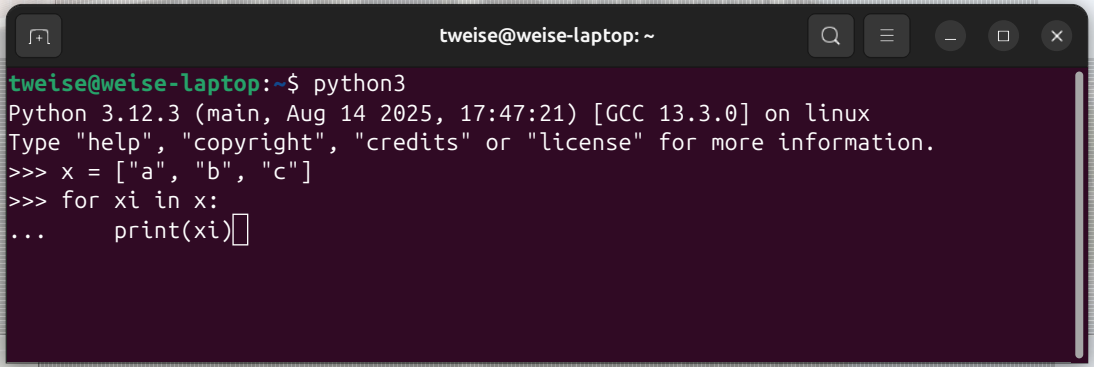
Iterator über Listen

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl+Alt+T, unter Microsoft Windows durch Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Sagen wir, wir haben eine Liste `x = ["a", "b", "c"]`.



Iterator über Listen

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

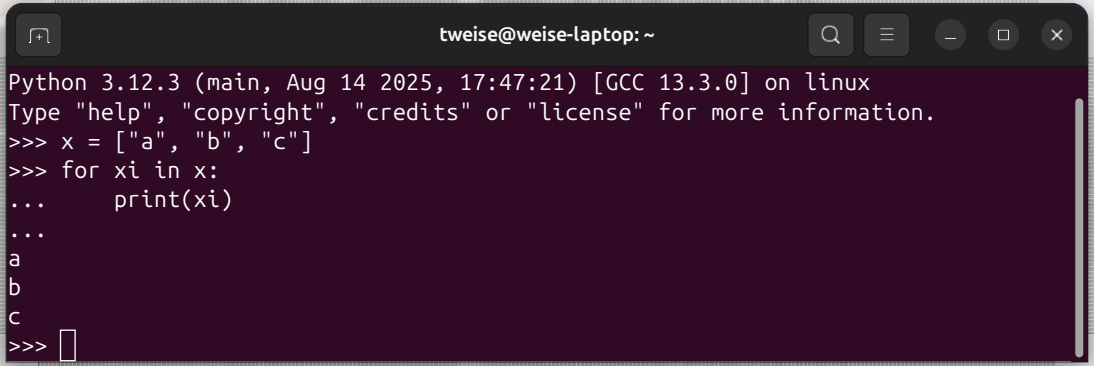


Iterator über Listen

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

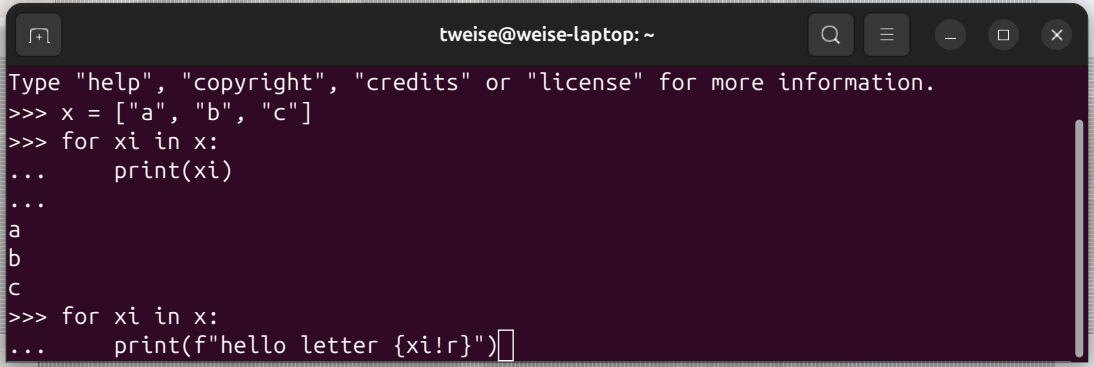
Iterator über Listen

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.



Iterator über Listen

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.



Iterator über Listen

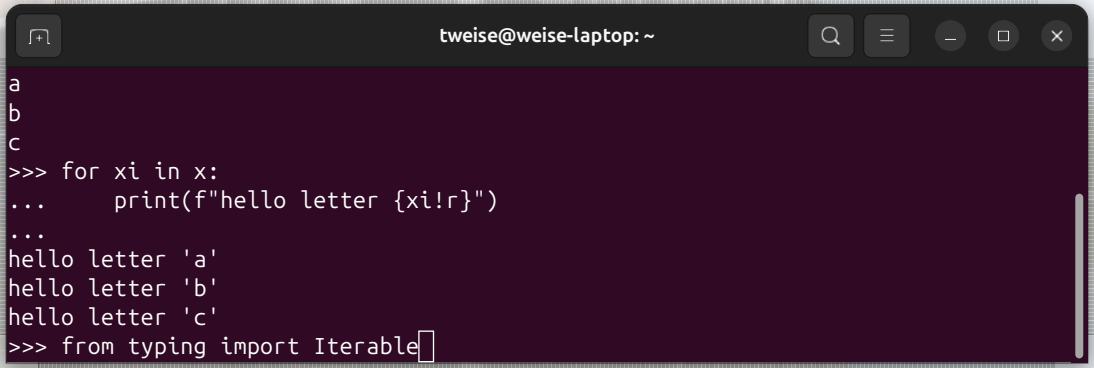
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

Iterator über Listen

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Liste mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

Iterator über Listen

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- `x` ist eine Instanz von `list` und jede Liste ist auch eine Instanz von `Iterable`[11].



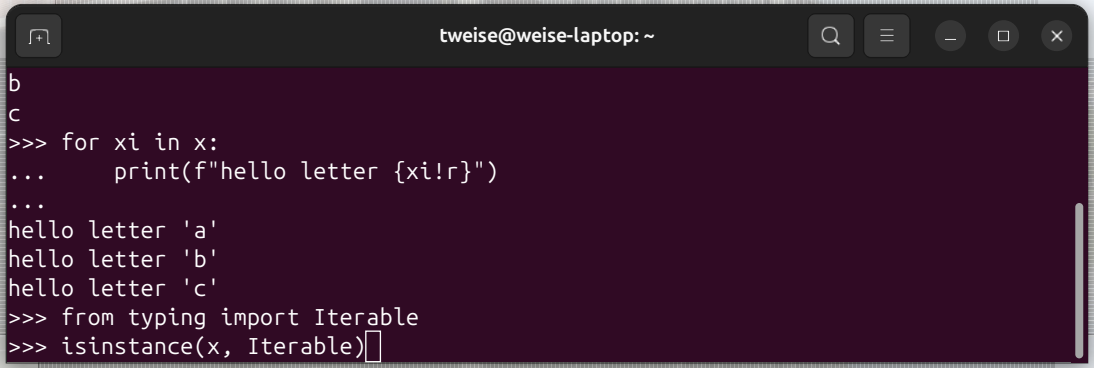
Iterator über Listen

- `x` ist eine Instanz von `list` und jede Liste ist auch eine Instanz von `Iterable`[11].
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.

```
tweise@weise-laptop: ~
b
c
>>> for xi in x:
...     print(f"hello letter {xi!r}")
...
hello letter 'a'
hello letter 'b'
hello letter 'c'
>>> from typing import Iterable
>>>
```

Iterator über Listen

- `x` ist eine Instanz von `list` und jede Liste ist auch eine Instanz von `Iterable`[11].
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.



Iterator über Listen

- `x` ist eine Instanz von `list` und jede Liste ist auch eine Instanz von `Iterable`[11].
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Das ist es tatsächlich.

```
tweise@weise-laptop: ~
>>> for xi in x:
...     print(f"hello letter {xi!r}")
...
hello letter 'a'
hello letter 'b'
hello letter 'c'
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>>
```

Iterator über Listen

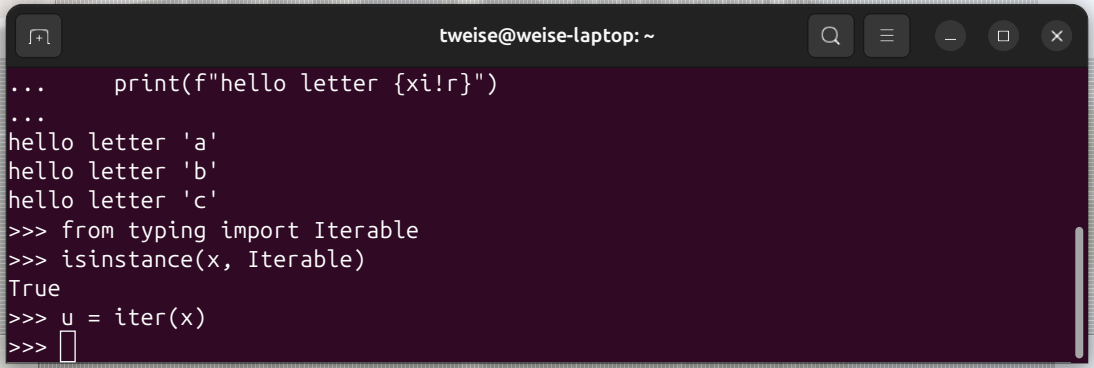
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Jedes Mal, wenn wir über `x` iterieren, dann wird intern eine Instanz von `Iterator` erstellt, in dem `iter(x)` aufgerufen wird.

tweise@weise-laptop: ~

```
>>> for xi in x:
...     print(f"hello letter {xi!r}")
...
hello letter 'a'
hello letter 'b'
hello letter 'c'
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
```

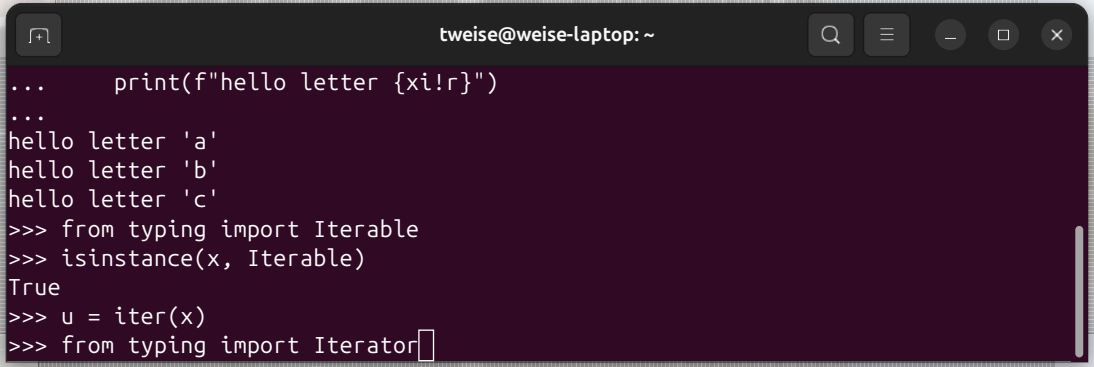
Iterator über Listen

- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Jedes Mal, wenn wir über `x` iterieren, dann wird intern eine Instanz von `Iterator` erstellt, in dem `iter(x)` aufgerufen wird.
- Natürlich können wir auch selbst `u = iter(x)` machen.



Iterator über Listen

- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.



Iterator über Listen

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.

Iterator über Listen

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.
- Prüfen wir ob `u` wirlich eine Instance von `Iterator` ist.

tweise@weise-laptop: ~

```
...
hello letter 'a'
hello letter 'b'
hello letter 'c'
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
```

Iterator über Listen

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.
- Prüfen wir ob `u` wirlich eine Instance von `Iterator` ist.
- Ist es.

tweise@weise-laptop: ~

```
hello letter 'b'
hello letter 'c'
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>>
```

Iterator über Listen

- Genaugenommen ist es ein Spezialfall davon, nämlich ein `list_iterator`.

Iterator über Listen

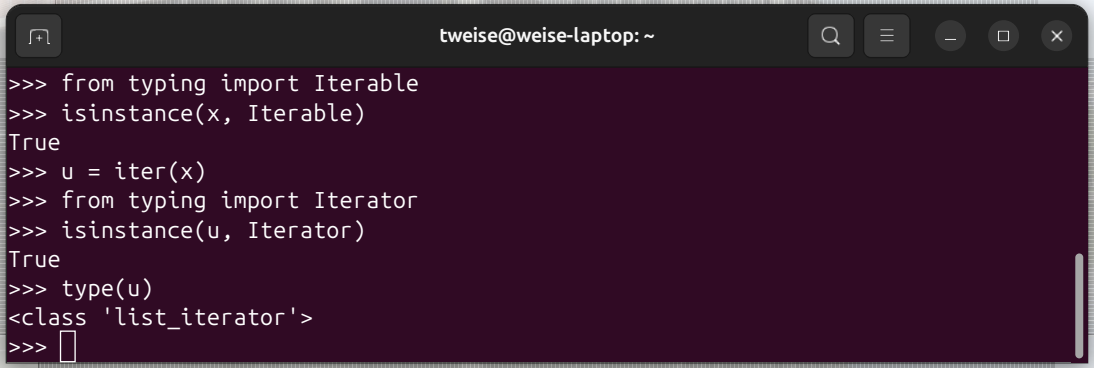
- Genaugenommen ist es ein Spezialfall davon, nämlich ein `list_iterator`.

tweise@weise-laptop: ~

```
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'list_iterator'>
>>>
```

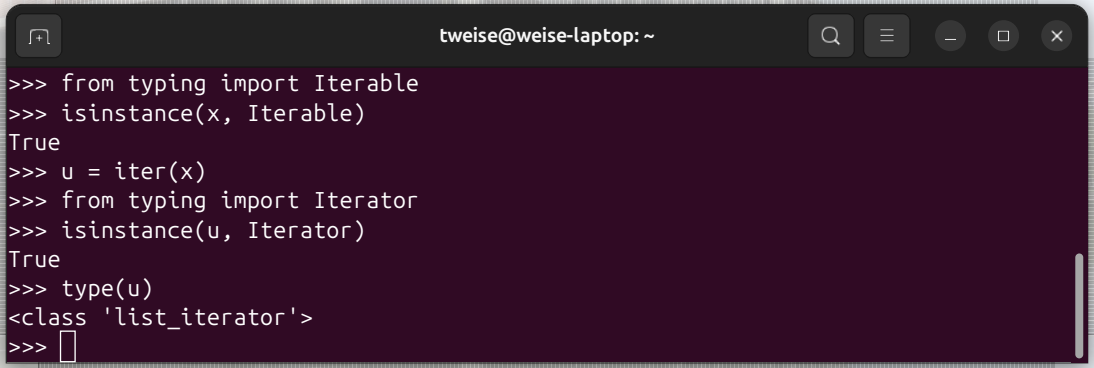
Iterator über Listen

- Alles, was so ein Iterator machen muss, ist sich eine Referenz auf die Liste, zu der er gehört, zu merken, sowie die aktuelle Position in der Liste, also den Index des aktuellen Elements.



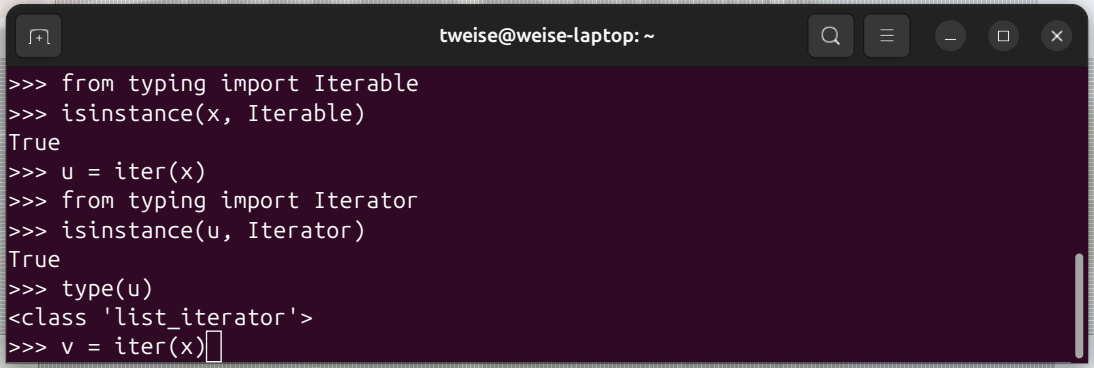
Iterator über Listen

- Wir können dann immer mit `next(u)` das nächste Element abfragen. Dabei wird das Element am aktuellen Index des Iterators zurückgeliefert und der Index dann um eins erhöht.



Iterator über Listen

- Wir können auch einen weiteren völlig unabhängigen Iterator `v` für `x` erstellen, der sich ebenfalls eine Referenz auf `x` sowie einen eigenen Index merkt.



Iterator über Listen

- Wir können auch einen weiteren völlig unabhängigen Iterator `v` für `x` erstellen, der sich ebenfalls eine Referenz auf `x` sowie einen eigenen Index merkt.

Iterator über Listen

- `next(u)` gibt uns das erste Element in der Iteration `u` über `x`.

tweise@weise-laptop: ~

```
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'list_iterator'>
>>> v = iter(x)
>>> next(u)
```

Iterator über Listen

- `next(u)` gibt uns das erste Element in der Iteration `u` über `x`.
- Das ist das erste Element aus der Liste, nämlich `"a"`.

tweise@weise-laptop: ~

```
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'list_iterator'>
>>> v = iter(x)
>>> next(u)
'a'
>>>
```

Iterator über Listen

- Jetzt gibt `next(u)` uns das nächste, also zweite Element in der Iteration `u` über `x`.

Iterator über Listen

- Jetzt gibt `next(u)` uns das nächste, also zweite Element in der Iteration `u` über `x`.
- Das ist das zweite Element aus der Liste, nämlich `"b"`.

Iterator über Listen

- `next(v)` gibt uns das erste Element in der Iteration `v` über `x`.

tweise@weise-laptop: ~

```
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'list_iterator'>
>>> v = iter(x)
>>> next(u)
'a'
>>> next(u)
'b'
>>> next(v)
```

Iterator über Listen

- `next(v)` gibt uns das erste Element in der Iteration `v` über `x`.
- Das ist auch das erste Element aus der Liste, nämlich `"a"`.

Iterator über Listen

- Nun gibt `next(u)` uns das nächste, also dritte und letzte Element in der Iteration `u` über `x`.

tweise@weise-laptop: ~

```
>>> type(u)
<class 'list_iterator'>
>>> v = iter(x)
>>> next(u)
'a'
>>> next(u)
'b'
>>> next(v)
'a'
>>> next(u)
```

Iterator über Listen

- Nun gibt `next(u)` uns das nächste, also dritte und letzte Element in der Iteration `u` über `x`.
- Das dritte Element aus der Liste ist `"c"`.

tweise@weise-laptop: ~

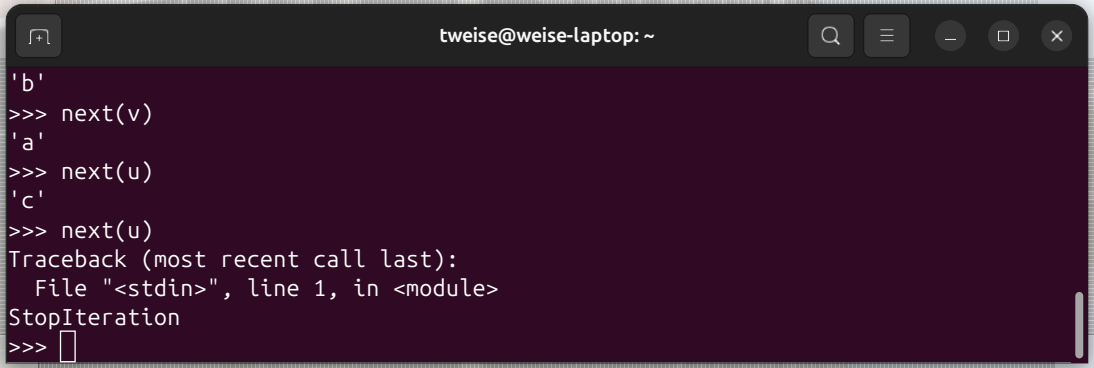
```
>>> v = iter(x)
>>> next(u)
'a'
>>> next(u)
'b'
>>> next(v)
'a'
>>> next(u)
'c'
>>>
```

Iterator über Listen

- Nun sind wir am Ende der Iteration `u`. Wenn wir nochmal `next(u)` machen...

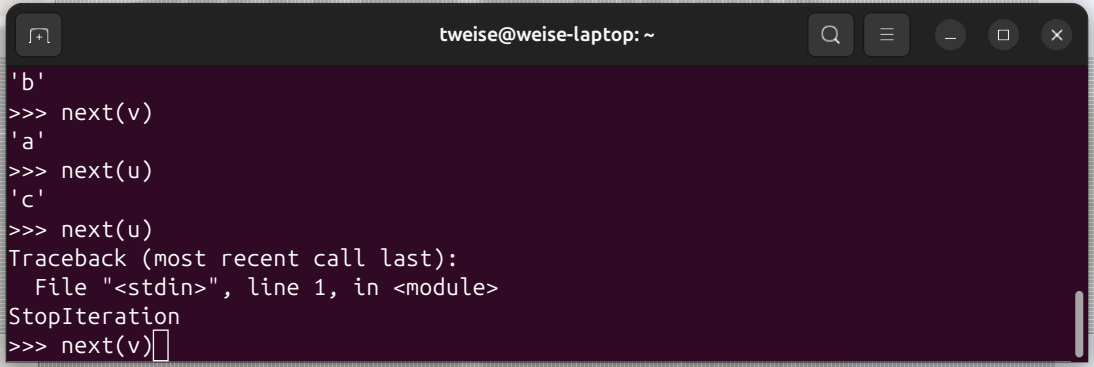
Iterator über Listen

- Nun sind wir am Ende der Iteration `u`. Wenn wir nochmal `next(u)` machen...
- ...dann wird eine `StopIteration` Ausnahme ausgelöst. Das ist kein Fehler, sondern gewollt. Irgendwie muss ja signalisiert werden, dass die Iteration zuende ist. `None` zurückzuliefern würde nicht gehen, weil das ja auch in der Liste vorkommen könnte. Also hat man einfach den Mechanismus für Ausnahmen dafür verwendet...



Iterator über Listen

- Wir können `next(v)` mit dem unabhängigen Iterator `v` über `x` machen und bekommen das zweite Element aus dessen Sequenz.



Iterator über Listen

- Wir können `next(v)` mit dem unabhängigen Iterator `v` über `x` machen und bekommen das zweite Element aus dessen Sequenz.
- Das ist das zweite Element aus der Liste, nämlich `"b"`.

tweise@weise-laptop: ~

```
'a'
>>> next(u)
'c'
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
'b'
>>>
```

Iterator über Listen

- Via `iter(x)` können wir einen weiteren unabhängigen Iterator `w` über `x` erstellen.

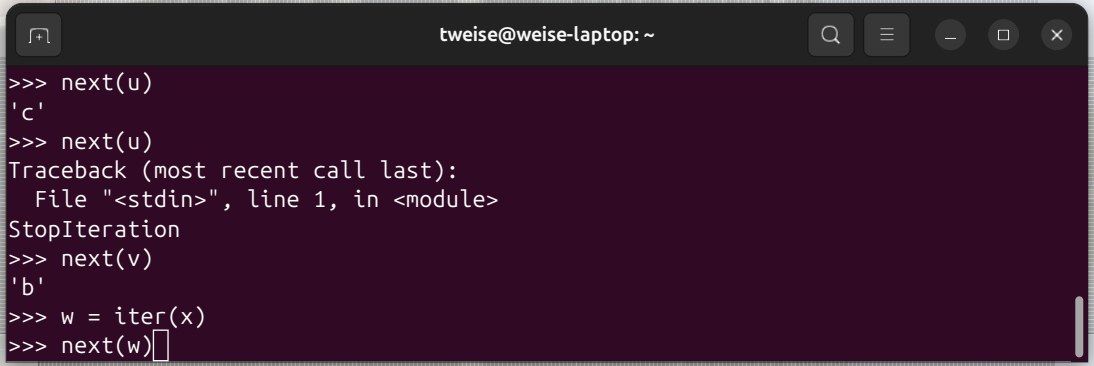
Iterator über Listen

- Via `iter(x)` können wir einen weiteren unabhängigen Iterator `w` über `x` erstellen.

```
tweise@weise-laptop: ~
>>> next(u)
'c'
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
'b'
>>> w = iter(x)
>>>
```

Iterator über Listen

- Machen wir `next(w)` bekommen wir wieder das erste Element aus der Liste.



Iterator über Listen

- Machen wir `next(w)` bekommen wir wieder das erste Element aus der Liste.
- ... nämlich `"a"`.

tweise@weise-laptop: ~

```
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
'b'
>>> w = iter(x)
>>> next(w)
'a'
>>>
```

Iterator über Listen

- `next(v)` liefert uns jetzt das letzte Element aus seiner Sequenz.

Iterator über Listen

- `next(v)` liefert uns jetzt das letzte Element aus seiner Sequenz.
- ... nämlich `"c"`.

Iterator über Listen

- ... nämlich `"c"`.
- Und wenn wir nochmal `next(v)` machen, bekommen wir wieder eine `StopIteration`-Ausnahme.

```
tweise@weise-laptop: ~
>>> next(w)
'a'
>>>
>>> next(v)
'c'
>>> next(v)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.
- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.

Über andere Kollektionen iterieren

- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.

```
"""Iterating over a set: The resulting order is not clear!"""

my_set: set[str] = {      # Create a set of the 26 Latin letters.
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"}
my_list: list[str] = []   # A list to receive the set elements.

for element in my_set:    # Iterate over the set: The order is undefined.
    my_list.append(element)

# Merging all the letters in the order in which they were visited into a
# single string and printing them. Each time we run this program, the
# result is likely to be different.
print("".join(my_list))
```

↓ `python3 set_iteration.py` ↓

```
octvygdknjilamxrbewzpfqhsu
```

Über andere Kollektionen iterieren

- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.

```python
"""Iterating over a set: The resulting order is not clear!"""

my_set: set[str] = {      # Create a set of the 26 Latin letters.
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"}
my_list: list[str] = []   # A list to receive the set elements.

for element in my_set:    # Iterate over the set: The order is undefined.
    my_list.append(element)

# Merging all the letters in the order in which they were visited into a
# single string and printing them. Each time we run this program, the
# result is likely to be different.
print("".join(my_list))
```

↓ `python3 set_iteration.py` ↓

```
octvygdknjilamxrbewzpfqhsu
```

```
gbvswtzmxakqfuircndopeyljh
```

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.
- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.
- Interessanterweise können wir auch über Dictionaries genau so iterieren.

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.
- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.
- Interessanterweise können wir auch über Dictionaries genau so iterieren.
- Allerdings bekommen wir dann nur die Schlüssel geliefert.

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.
- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.
- Interessanterweise können wir auch über Dictionaries genau so iterieren.
- Allerdings bekommen wir dann nur die Schlüssel geliefert.
- Wollen wir über die Schlüssel-Wert-Paare aus einem Dictionary `d` iterieren, dann müssen wir über `d.items()` iterieren.

Über andere Kollektionen iterieren

- Die Methode um über Kollektionen `col` zu iterieren, in dem wir zuerst einen Iterator `it` mit Hilfe der `iter`-Funktion erzeugen via `it = iter(col)` und danach dann `next` auf diesen Iterator anwenden, also `next(it)` aufrufen, funktioniert für Listen genauso wie für Tupel.
- Es funktioniert auch für Mengen, wobei die Reihenfolge von Elementen in einem `set` undefiniert ist.
- Jedesmal, wenn wir ein Program mit einer Iteration über ein `set` ausführen, könnte es eine andere Reihenfolge verwenden.
- Interessanterweise können wir auch über Dictionaries genau so iterieren.
- Allerdings bekommen wir dann nur die Schlüssel geliefert.
- Wollen wir über die Schlüssel-Wert-Paare aus einem Dictionary `d` iterieren, dann müssen wir über `d.items()` iterieren.
- Wollen wir über die Werte aus einem Dictionary `d` iterieren, dann müssen wir über `d.values()` iterieren.

Iterator über Ranges

- Wir können genauso auch über `ranges` iterieren.

Iterator über Ranges

- Wir können genauso auch über `ranges` iterieren.
- `ranges` sind wie Listen Kollektionen, allerdings werden ihre Elemente nicht explizit erstellt und gespeichert.

Iterator über Ranges

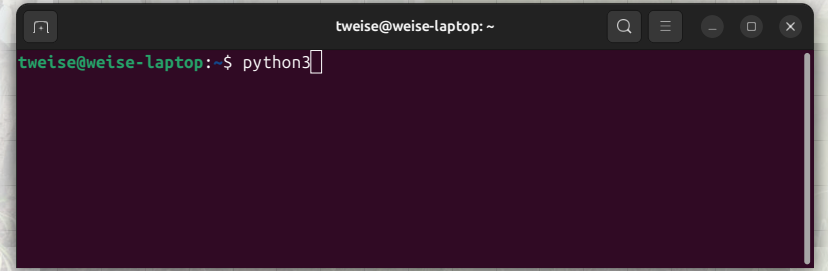
- Wir können genauso auch über `ranges` iterieren.
- `ranges` sind wie Listen Kollektionen, allerdings werden ihre Elemente nicht explizit erstellt und gespeichert.
- Stattdessen werden die Elemente erst erstellt, wenn ein `Iterator` über die `range` sie zurückliefern muss.

Iterator über Ranges

- `ranges` sind wie Listen Kollektionen, allerdings werden ihre Elemente nicht explizit erstellt und gespeichert.
- Stattdessen werden die Elemente erst erstellt, wenn ein `Iterator` über die `range` sie zurückliefern muss.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).

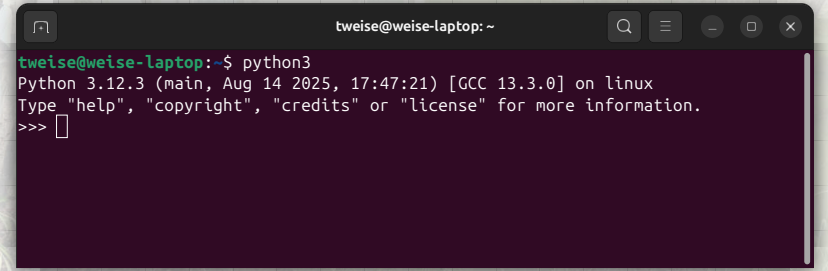
Iterator über Ranges

- Stattdessen werden die Elemente erst erstellt, wenn ein `Iterator` über die `range` sie zurückliefern muss.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.



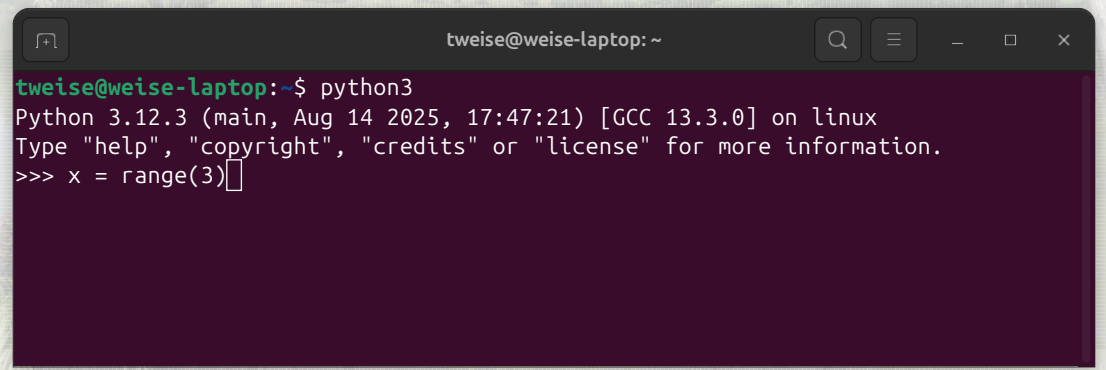
Iterator über Ranges

- Stattdessen werden die Elemente erst erstellt, wenn ein `Iterator` über die `range` sie zurückliefern muss.
- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir sind nun im Interpreter.



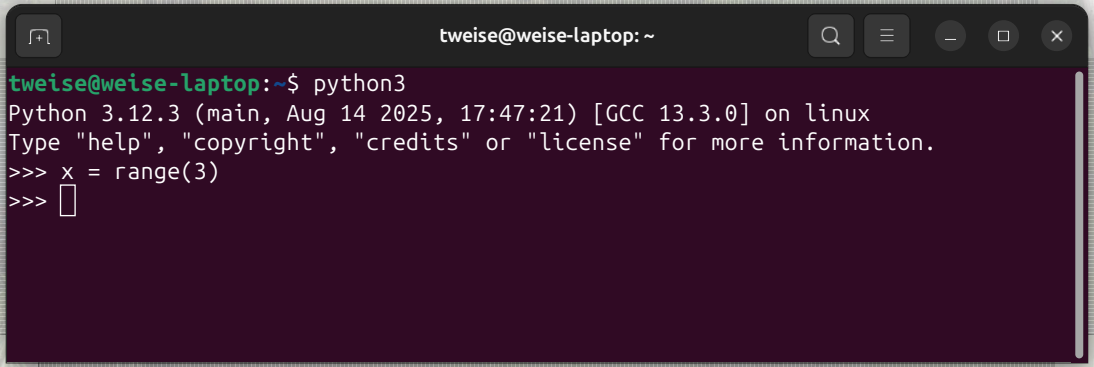
Iterator über Ranges

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Sagen wir, wir haben eine range `x = range(3)` über die drei Zahlen 0, 1, und 2.



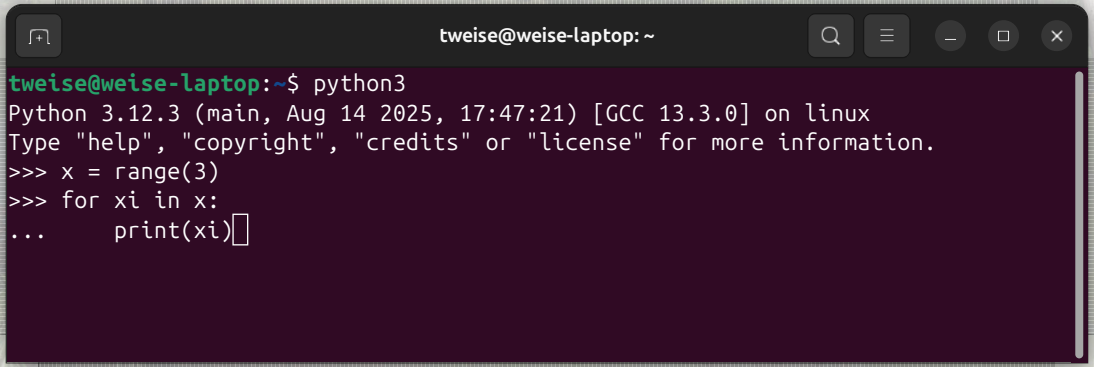
Iterator über Ranges

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl + Alt + T, unter Microsoft Windows durch Druck auf ⊞ + R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Sagen wir, wir haben eine range `x = range(3)` über die drei Zahlen 0, 1, und 2.



Iterator über Ranges

- Wir öffnen ein Terminal um uns das anzuschauen (unter Ubuntu Linux durch Drücken von Ctrl+Alt+T, unter Microsoft Windows durch Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵).
- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

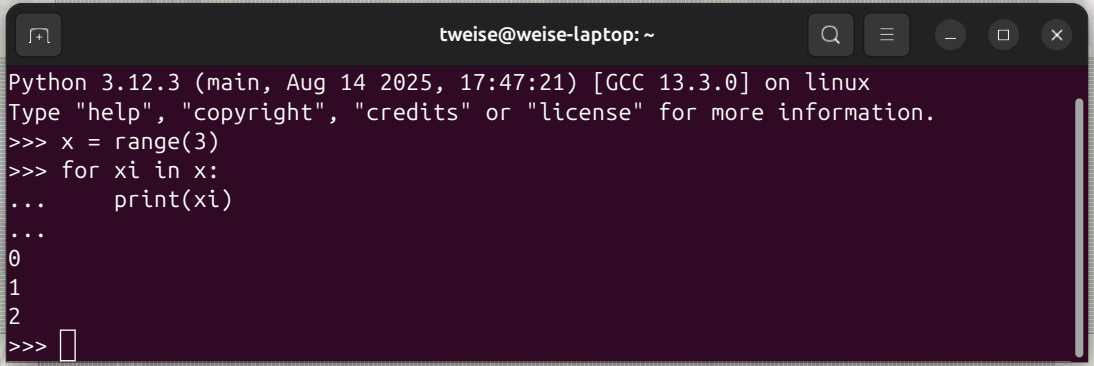


Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

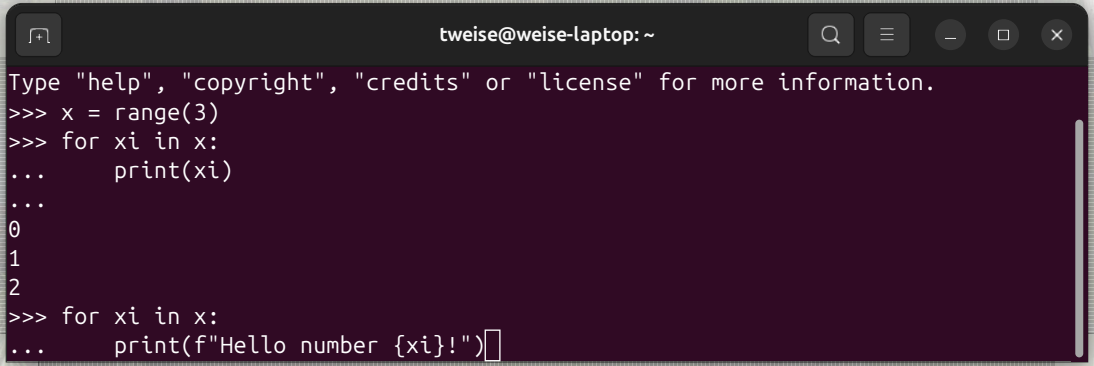
Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.



Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

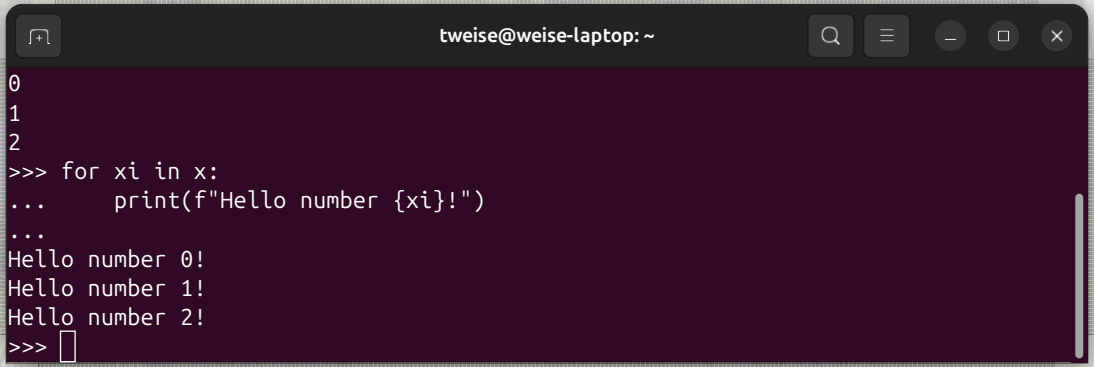


Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.

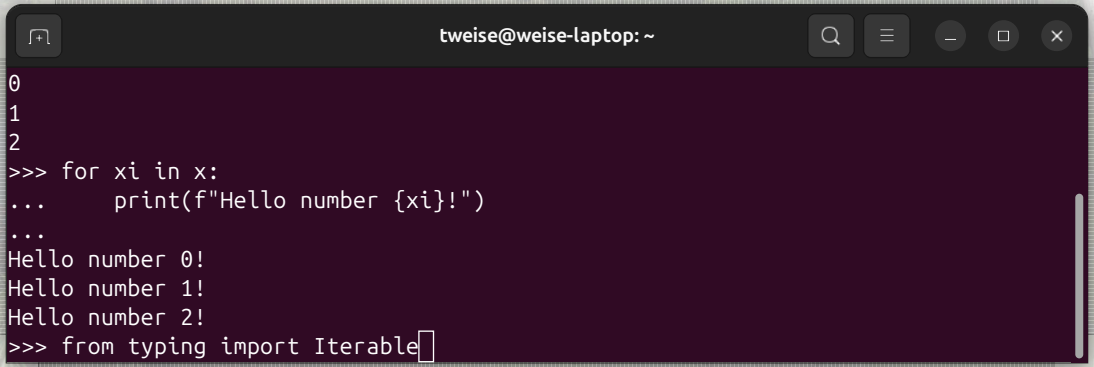
Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- Wir können über diese Range mit `for xi in x`-ähnlichen Schleifen beliebig oft iterieren.



Iterator über Ranges

- Wir starten den Python-Interpreter, in dem wir `python3` schreiben und ↵ drücken.
- `x` ist eine Instanz von `range` und jede Range ist auch eine Instanz von `Iterable`.



Iterator über Ranges

- `x` ist eine Instanz von `range` und jede Range ist auch eine Instanz von `Iterable`.
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.

```
tweise@weise-laptop: ~
1
2
>>> for xi in x:
...     print(f"Hello number {xi}!")
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>>
```

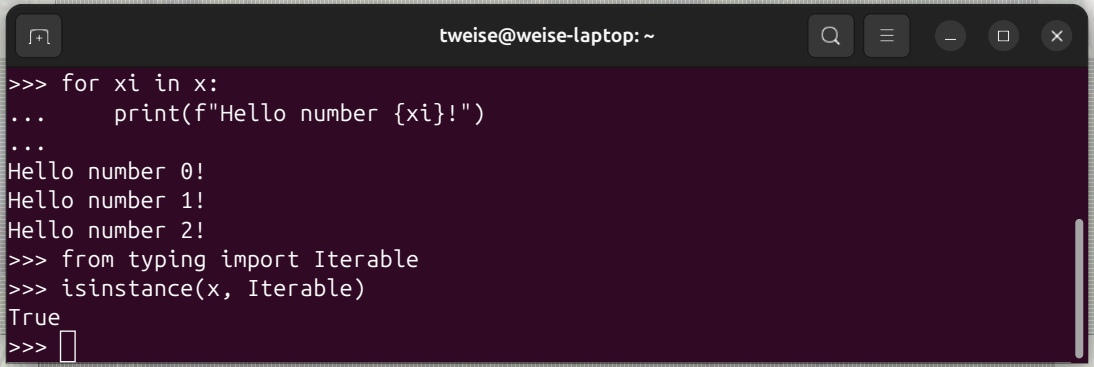
Iterator über Ranges

- `x` ist eine Instanz von `range` und jede Range ist auch eine Instanz von `Iterable`.
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.

```
tweise@weise-laptop: ~
1
2
>>> for xi in x:
...     print(f"Hello number {xi}!")
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>> isinstance(x, Iterable)
```

Iterator über Ranges

- `x` ist eine Instanz von `range` und jede Range ist auch eine Instanz von `Iterable`.
- Wir wollen das nachrüfen und importieren daher diesen Datentyp.
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Das ist es tatsächlich.



Iterator über Ranges

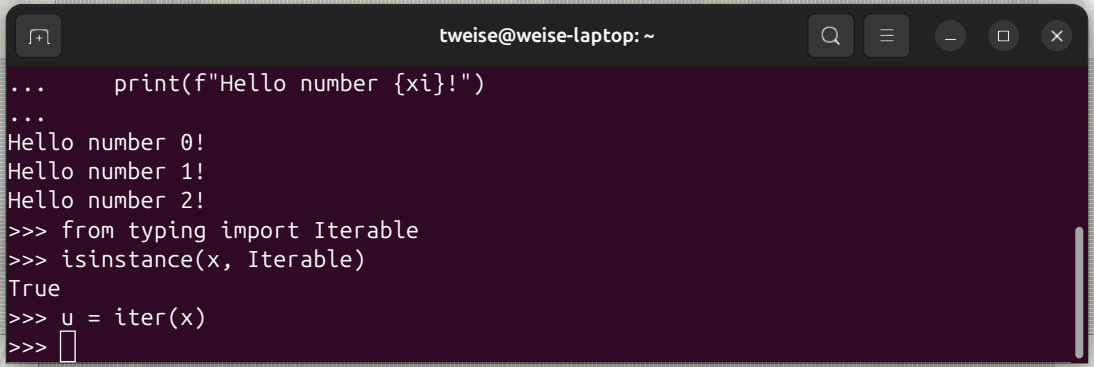
- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Jedes Mal, wenn wir über `x` iterieren, dann wird intern eine Instanz von `Iterator` erstellt, in dem `iter(x)` aufgerufen wird.

tweise@weise-laptop: ~

```
>>> for xi in x:
...     print(f"Hello number {xi}!")
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
```

Iterator über Ranges

- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Jedes Mal, wenn wir über `x` iterieren, dann wird intern eine Instanz von `Iterator` erstellt, in dem `iter(x)` aufgerufen wird.
- Natürlich können wir auch selbst `u = iter(x)` machen.



Iterator über Ranges

- Wir prüfen, ob `x` wirklich eine Instance von `Iterable` ist.
- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.

tweise@weise-laptop: ~

```
...     print(f"Hello number {xi}!")
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
```

Iterator über Ranges

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.

```
tweise@weise-laptop: ~
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>>
```

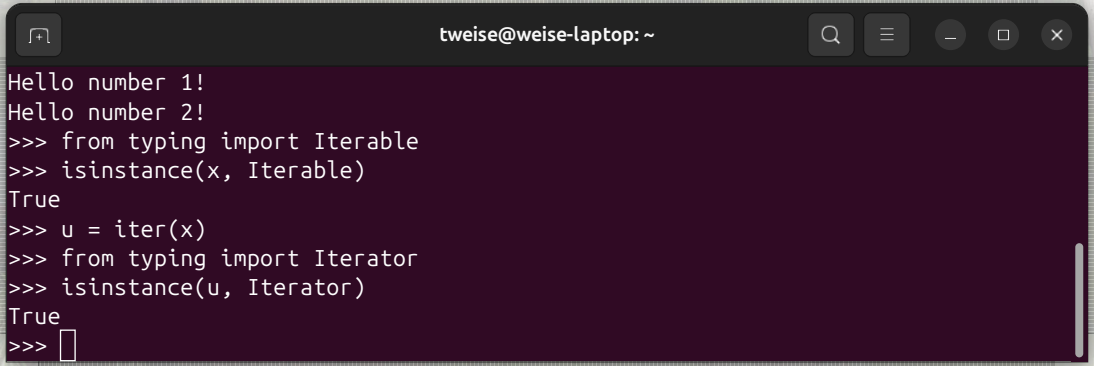
Iterator über Ranges

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.
- Prüfen wir ob `u` wirlich eine Instance von `Iterator` ist.

```
tweise@weise-laptop: ~
...
Hello number 0!
Hello number 1!
Hello number 2!
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
```

Iterator über Ranges

- Importieren wir den Datentyp `Iterator`, damit wir schauen können, ob `u` wirklich eine Instanz davon ist.
- Prüfen wir ob `u` wirlich eine Instance von `Iterator` ist.
- Ist es.



Iterator über Ranges

- Genaugenommen ist es ein Spezialfall davon, nämlich ein `range_iterator`.

Iterator über Ranges

- Genaugenommen ist es ein Spezialfall davon, nämlich ein `range_iterator`.

tweise@weise-laptop: ~

```
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>>
```

Iterator über Ranges

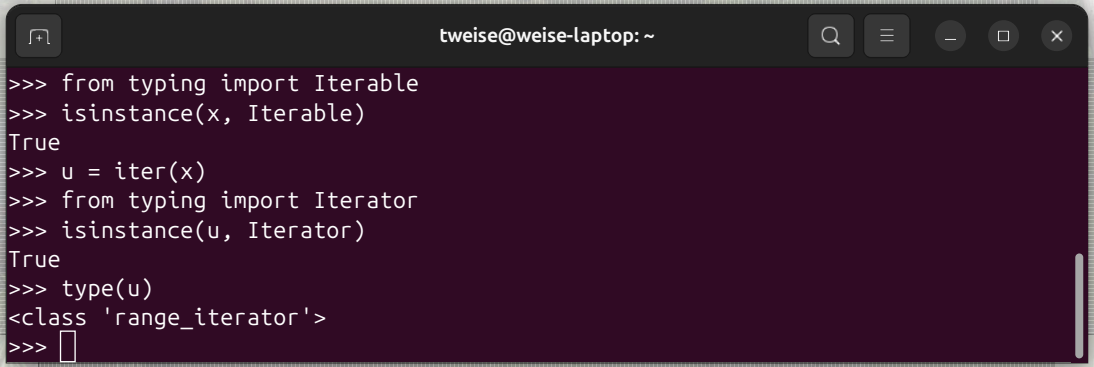
- Alles, was so ein Iterator machen muss, ist sich eine Referenz auf die Range, zu der er gehört, zu merken, sowie die aktuelle Position in der Range.

tweise@weise-laptop: ~

```
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>>
```

Iterator über Ranges

- Alles, was so ein Iterator machen muss, ist sich eine Referenz auf die Range, zu der er gehört, zu merken, sowie die aktuelle Position in der Range.
- Wir können dann immer mit `next(u)` das nächste Element abfragen. Dabei wird das aktuellen aktuelle Element in der Sequenz zurückgeliefert. Die Position wird entsprechend der Schrittweite weitergerückt.



Iterator über Ranges

- Wir können auch einen weiteren völlig unabhängigen Iterator `v` für `x` erstellen, der sich ebenfalls eine Referenz auf `x` sowie eine Position.

tweise@weise-laptop: ~

```
>>> from typing import Iterable
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
```

Iterator über Ranges

- Wir können auch einen weiteren völlig unabhängigen Iterator `v` für `x` erstellen, der sich ebenfalls eine Referenz auf `x` sowie eine Position.

tweise@weise-laptop: ~

```
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>>
```

Iterator über Ranges

- `next(u)` gibt uns das erste Element in der Iteration `u` über `x`.

tweise@weise-laptop: ~

```
>>> isinstance(x, Iterable)
True
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
```

Iterator über Ranges

- `next(u)` gibt uns das erste Element in der Iteration `u` über `x`.
- Das ist das erste Element aus der Range, nämlich `0`.

tweise@weise-laptop: ~

```
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>>
```

Iterator über Ranges

- Jetzt gibt `next(u)` uns das nächste, also zweite Element in der Iteration `u` über `x`.

tweise@weise-laptop: ~

```
>>> u = iter(x)
>>> from typing import Iterator
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
```

Iterator über Ranges

- Jetzt gibt `next(u)` uns das nächste, also zweite Element in der Iteration `u` über `x`.
- Das ist das zweite Element aus der Range, nämlich `1`.

tweise@weise-laptop: ~

```
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>>
```

Iterator über Ranges

- `next(v)` gibt uns das erste Element in der Iteration `v` über `x`.

tweise@weise-laptop: ~

```
>>> isinstance(u, Iterator)
True
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>> next(v)
```

Iterator über Ranges

- `next(v)` gibt uns das erste Element in der Iteration `v` über `x`.
- Das ist auch das erste Element aus der Range, nämlich `0`.

tweise@weise-laptop: ~

```
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>> next(v)
0
>>>
```

Iterator über Ranges

- Nun gibt `next(u)` uns das nächste, also dritte und letzte Element in der Iteration `u` über `x`.

tweise@weise-laptop: ~

```
>>> type(u)
<class 'range_iterator'>
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>> next(v)
0
>>> next(u)
```

Iterator über Ranges

- Nun gibt `next(u)` uns das nächste, also dritte und letzte Element in der Iteration `u` über `x`.
- Das dritte Element aus der Range ist `2`.

tweise@weise-laptop: ~

```
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>> next(v)
0
>>> next(u)
2
>>>
```

Iterator über Ranges

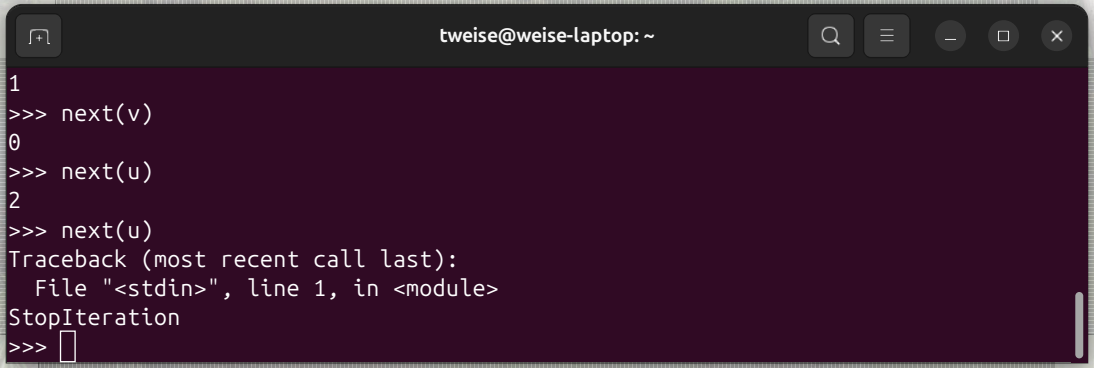
- Nun sind wir am Ende der Iteration `u`. Wenn wir nochmal `next(u)` machen...

tweise@weise-laptop: ~

```
>>> v = iter(x)
>>> next(u)
0
>>> next(u)
1
>>> next(v)
0
>>> next(u)
2
>>> next(u)
```

Iterator über Ranges

- Nun sind wir am Ende der Iteration `u`. Wenn wir nochmal `next(u)` machen...
- ...dann wird eine `StopIteration` Ausnahme ausgelöst. Das ist kein Fehler, sondern gewollt. Irgendwie muss ja signalisiert werden, dass die Iteration zuende ist.



Iterator über Ranges

- Wir können `next(v)` mit dem unabhängigen Iterator `v` über `x` machen und bekommen das zweite Element aus dessen Sequenz.

tweise@weise-laptop: ~

```
1
>>> next(v)
0
>>> next(u)
2
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
```

Iterator über Ranges

- Wir können `next(v)` mit dem unabhängigen Iterator `v` über `x` machen und bekommen das zweite Element aus dessen Sequenz.
- Das ist das zweite Element aus der Range, nämlich `1`.

```
tweise@weise-laptop: ~
0
>>> next(u)
2
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
1
>>>
```

Iterator über Ranges

- Via `iter(x)` können wir einen weiteren unabhängigen Iterator `w` über `x` erstellen.

Iterator über Ranges

- Via `iter(x)` können wir einen weiteren unabhängigen Iterator `w` über `x` erstellen.

tweise@weise-laptop: ~

```
>>> next(u)
2
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
1
>>> w = iter(x)
>>>
```

Iterator über Ranges

- Machen wir `next(w)` bekommen wir wieder das erste Element aus der Range.

Iterator über Ranges

- Machen wir `next(w)` bekommen wir wieder das erste Element aus der Range.
- ... nämlich `0`.

tweise@weise-laptop: ~

```
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
1
>>> w = iter(x)
>>> next(w)
0
>>>
```

Iterator über Ranges

- `next(v)` liefert uns jetzt das letzte Element aus seiner Sequenz.

```
tweise@weise-laptop: ~
>>> next(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
1
>>> w = iter(x)
>>> next(w)
0
>>> next(v)
```

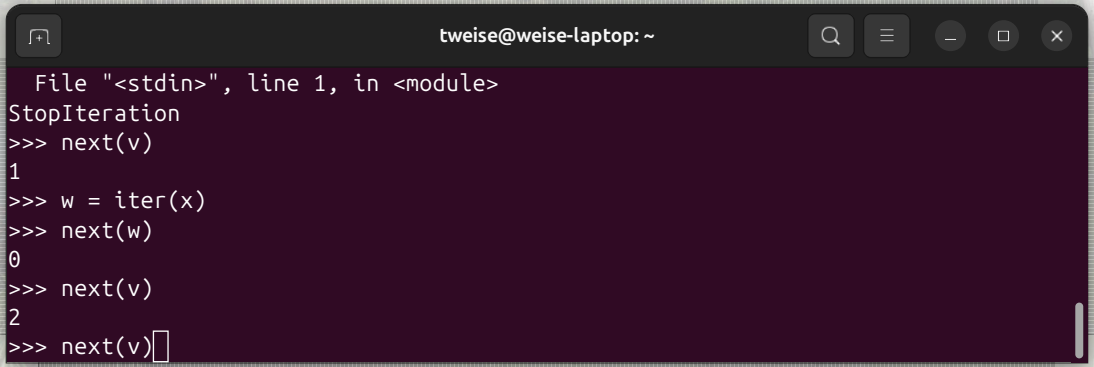
Iterator über Ranges

- `next(v)` liefert uns jetzt das letzte Element aus seiner Sequenz.
- ... nämlich `2`.

```
tweise@weise-laptop: ~
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(v)
1
>>> w = iter(x)
>>> next(w)
0
>>> next(v)
2
>>>
```

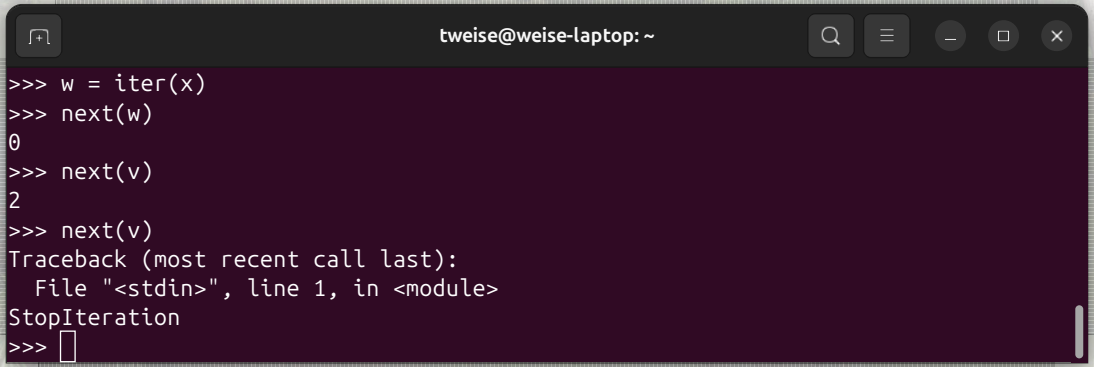
Iterator über Ranges

- ... nämlich `2`.
- Und wenn wir nochmal `next(v)` machen...



Iterator über Ranges

- ... nämlich `2`.
- Und wenn wir nochmal `next(v)` machen...
- ... bekommen wir wieder eine `StopIteration`-Ausnahme.



Zusammenfassung

Zusammenfassung

- Jetzt wissen wir, wie `for`-Schleifen in Python funktionieren.

Zusammenfassung

- Jetzt wissen wir, wie `for`-Schleifen in Python funktionieren.
- Sie erzeugen einen Iterator über eine Sequenz und konsumieren dann dessen Elemente eins nach dem Anderen, wobei der Schleifenkörper für jedes Element einmal ausgeführt wird – so lange, bis eine `StopIteration`-Ausnahme empfangen wird.

Zusammenfassung

- Jetzt wissen wir, wie `for`-Schleifen in Python funktionieren.
- Sie erzeugen einen Iterator über eine Sequenz und konsumieren dann dessen Elemente eins nach dem Anderen, wobei der Schleifenkörper für jedes Element einmal ausgeführt wird – so lange, bis eine `StopIteration`-Ausnahme empfangen wird.
- Alle Kollektionen in Python die eine sequenzielle Sicht auf ihre Daten ermöglichen, implementieren daher diese `Iterable`/`Iterator`-API[28].

Zusammenfassung

- Jetzt wissen wir, wie `for`-Schleifen in Python funktionieren.
- Sie erzeugen einen Iterator über eine Sequenz und konsumieren dann dessen Elemente eins nach dem Anderen, wobei der Schleifenkörper für jedes Element einmal ausgeführt wird – so lange, bis eine `StopIteration`-Ausnahme empfangen wird.
- Alle Kollektionen in Python die eine sequenzielle Sicht auf ihre Daten ermöglichen, implementieren daher diese `Iterable`/`Iterator`-API[28].
- Dank dieser API-Struktur ist es auch gar nicht notwendig, alle Elemente einer Kollektion im Speicher zu halten, so lange wir sie bei Zugriff erzeugen können.

Zusammenfassung

- Jetzt wissen wir, wie `for`-Schleifen in Python funktionieren.
- Sie erzeugen einen Iterator über eine Sequenz und konsumieren dann dessen Elemente eins nach dem Anderen, wobei der Schleifenkörper für jedes Element einmal ausgeführt wird – so lange, bis eine `StopIteration`-Ausnahme empfangen wird.
- Alle Kollektionen in Python die eine sequenzielle Sicht auf ihre Daten ermöglichen, implementieren daher diese `Iterable`/`Iterator`-API[28].
- Dank dieser API-Struktur ist es auch gar nicht notwendig, alle Elemente einer Kollektion im Speicher zu halten, so lange wir sie bei Zugriff erzeugen können.
- Ein Beispiel dafür sind `ranges`, die uns `int`-Sequenzen mit nahezu beliebig vielen Zahlen anbieten, die während der Iteration eine nach der Anderen erzeugt (und freigegeben) werden.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 159, 160).

[2] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 159).

[3] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 159).

[4] "Built-in Types: `Iterator` Types". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#iterator-types (besucht am 2025-09-16) (siehe S. 22–37, 41–44).

[5] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 160).

[6] *Python 3 Documentation. Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html (besucht am 2025-09-16).

[7] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 159).

[8] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 159).

[9] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 160).

[10] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 160).

[11] "`Iterable`". In: *Python 3 Documentation. Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html#term-iterable (besucht am 2025-09-16) (siehe S. 22–35, 41–58).

References II

[12] "`Iterator`". In: *Python 3 Documentation. Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html#term-iterator (besucht am 2025-09-16) (siehe S. 22–37, 41–44).

[13] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 160).

[14] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 160).

[15] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: https://peps.python.org/pep-0586 (besucht am 2024-12-17) (siehe S. 159).

[16] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra" Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 160).

[17] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 160).

[18] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 159).

[19] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 159).

[20] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: 978-0-596-15448-6 (siehe S. 159).

[21] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 159).

[22] ."Literals". In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (besucht am 2025-08-29) (siehe S. 159).

References III

[23] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 159).

[24] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 159, 160).

[25] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 160).

[26] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 159).

[27] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 160).

[28] Ka-Ping Yee und Guido van Rossum. *IteratorS*. Python Enhancement Proposal (PEP) 234. Beaverton, OR, USA: Python Software Foundation (PSF), 30. Jan.–30. Apr. 2001. URL: https://peps.python.org/pep-0234 (besucht am 2025-02-02) (siehe S. 22–28, 41–44, 150–154).

[29] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 159).

Glossary (in English) I

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another[7].

Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs[3,18,29]. Learn more at `https://www.gnu.org/software/bash`.

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[21,24]. Learn more at `https://git-scm.com`.

GitHub is a website where software projects can be hosted and managed via the Git VCS[19,24]. Learn more at `https://github.com`.

IT information technology

Linux is the leading open source operating system, i.e., a free alternative for Microsoft Windows[1,8,20,23,26]. We recommend using it for this course, for software development, and for research. Learn more at `https://www.linux.org`. Its variant Ubuntu is particularly easy to use and install.

literal A literal is a specific concrete value, something that is written down as-is[15,22]. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

Microsoft Windows is a commercial proprietary operating system[2]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at `https://www.microsoft.com/windows`.

Glossary (in English) II

Mypy is a static type checking tool for Python[16] that makes use of type hints. Learn more at https://github.com/python/mypy and in[27].

Python The Python programming language[10,14,17,27], i.e., what you will learn about in our book[27]. Learn more at https://python.org.

terminal A terminal is a text-based window where you can enter commands and execute them[1,5]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[13,25]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Ubuntu is a variant of the open source operating system Linux[5,9]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[24]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.