合肥大學
HEFEI UNIVERSITY



Programming with Python
36. Zwischenspiel: Doctests

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn · https://thomasweise.github.io

School of Artificial Intelligence and Big Data
Hefei University
Hefei, Anhui, China

人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2026-11-05

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



1. Einleitung
2. Beispiel
3. Zusammenfassung



Einleitung

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.
- Bei Unit Tests können wir sehen, welche Ausgabe wir für bestimmte, ausgewählte Eingabedaten erwarten können.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.
- Bei Unit Tests können wir sehen, welche Ausgabe wir für bestimmte, ausgewählte Eingabedaten erwarten können.
- Der Docstring einer `sqrt`-Funktion sagt uns, dass die Funktion die Quadratwurzel einer Zahl berechnet.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.
- Bei Unit Tests können wir sehen, welche Ausgabe wir für bestimmte, ausgewählte Eingabedaten erwarten können.
- Der Docstring einer `sqrt`-Funktion sagt uns, dass die Funktion die Quadratwurzel einer Zahl berechnet.
- Die Unit Tests zeigen uns, das `sqrt` genau `2.0` für die Eingabe `4` zurückliefert und `3.0` für `9`.

Docstrings und Unit Tests (1)



- Wir haben bereits gelernt, das Unit Tests Teil jedes Softwareentwicklungsprozesses sind und in jede vernünftigt Continuous Integration (CI) gehören.
- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.
- Bei Unit Tests können wir sehen, welche Ausgabe wir für bestimmte, ausgewählte Eingabedaten erwarten können.
- Der Docstring einer `sqrt`-Funktion sagt uns, dass die Funktion die Quadratwurzel einer Zahl berechnet.
- Die Unit Tests zeigen uns, das `sqrt` genau `2.0` für die Eingabe `4` zurückliefert und `3.0` für `9`.
- Ein Docstring sagt uns vielleicht, dass die Funktion einen `ArithmeticError` auslöst, wenn das Argument negativ ist.

Docstrings und Unit Tests (1)



- Wenn wir darüber nachdenken, erkennen wir, dass Unit Tests auch Teil der Dokumentation von Software sind.
- Der Docstring einer Funktion sagt uns, was die Funktion im Grunde macht, welche Parameter sie braucht, und welche Ausnahmen sie auslösen könnte.
- Dies wird durch Unit Tests ergänzt, welche uns sehr ähnliche Information in einer viel umfänglicheren Art bereitstellen.
- Bei Unit Tests können wir sehen, welche Ausgabe wir für bestimmte, ausgewählte Eingabedaten erwarten können.
- Der Docstring einer `sqrt`-Funktion sagt uns, dass die Funktion die Quadratwurzel einer Zahl berechnet.
- Die Unit Tests zeigen uns, das `sqrt` genau `2.0` für die Eingabe `4` zurückliefert und `3.0` für `9`.
- Ein Docstring sagt uns vielleicht, dass die Funktion einen `ArithmeticError` auslöst, wenn das Argument negativ ist.
- Die Unit Tests zeigen uns, dass sie einen `ArithmeticError` mit Fehlermeldung *„Invalid input -1.“* auslöst, wenn wir `-1` hineingeben.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.
- Natürlich würden wir nicht *alle* Unit Tests in den Docstring schreiben, weil wir oft viele verschiedene Testfälle haben, und unsere Dokumentation dann schnell unleserlich würde.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.
- Natürlich würden wir nicht *alle* Unit Tests in den Docstring schreiben, weil wir oft viele verschiedene Testfälle haben, und unsere Dokumentation dann schnell unleserlich würde.
- Aber eine Handvoll ausgewählter Tests könnten dem Leser schon helfen.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.
- Natürlich würden wir nicht *alle* Unit Tests in den Docstring schreiben, weil wir oft viele verschiedene Testfälle haben, und unsere Dokumentation dann schnell unleserlich würde.
- Aber eine Handvoll ausgewählter Tests könnten dem Leser schon helfen.
- Natürlich hält uns nichts davon ab, diese einfach hinzuschreiben.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.
- Natürlich würden wir nicht *alle* Unit Tests in den Docstring schreiben, weil wir oft viele verschiedene Testfälle haben, und unsere Dokumentation dann schnell unleserlich würde.
- Aber eine Handvoll ausgewählter Tests könnten dem Leser schon helfen.
- Natürlich hält uns nichts davon ab, diese einfach hinzuschreiben.
- Aber im Idealfall würde pytest dieses Tests auch in den Docstrings finden, sie ausführen und prüfen.

Docstrings und Unit Tests (2)

- Es ist leicht zu sehen, dass sich Docstrings und Unit Tests ergänzen.
- Wäre es nicht toll, wenn wir einige der Informationen aus den Unit Tests in die Docstrings schreiben könnten?
- Zum Beispiel, dass `sqrt(16)` als Ergebnis `4.0` ergibt, würde gut auf eine einzelne Zeile passen.
- Es wäre, nun ja, ein schönes Beispiel für jeden, der die Dokumentation unserer Funktion liest.
- Natürlich würden wir nicht *alle* Unit Tests in den Docstring schreiben, weil wir oft viele verschiedene Testfälle haben, und unsere Dokumentation dann schnell unleserlich würde.
- Aber eine Handvoll ausgewählter Tests könnten dem Leser schon helfen.
- Natürlich hält uns nichts davon ab, diese einfach hinzuschreiben.
- Aber im Idealfall würde pytest dieses Tests auch in den Docstrings finden, sie ausführen und prüfen.
- Tatsächlich ist so eine perfekte Synthese von Dokumentation und Testsen möglich – mit den so-genannten Doctests[5].

Beispiel

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.
- Wir werden eine Funktion `flatten` schreiben, die sogar eine noch eine allgemeinere Variante dieser Idee implementiert.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.
- Wir werden eine Funktion `flatten` schreiben, die sogar eine noch eine allgemeinere Variante dieser Idee implementiert.
- Sie soll ein `Iterable` von `Iterables` als Parameter akzeptieren.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.
- Wir werden eine Funktion `flatten` schreiben, die sogar eine noch eine allgemeinere Variante dieser Idee implementiert.
- Sie soll ein `Iterable` von `Iterables` als Parameter akzeptieren.
- Weil Listen ja auch `Iterables` sind, können wir also eine `list` von `listss` als argument hereingeben.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.
- Wir werden eine Funktion `flatten` schreiben, die sogar eine noch eine allgemeinere Variante dieser Idee implementiert.
- Sie soll ein `Iterable` von `Iterables` als Parameter akzeptieren.
- Weil Listen ja auch `Iterables` sind, können wir also eine `list` von `listss` als argument hereingeben.
- Wir können aber auch ein `tuple` von `sets` angeben, wenn wir wollen.

Flatten Iterables



- Wir probieren das aus ... mit einem letzten Beispiel für List Comprehension.
- Stellen Sie sich vor, dass sie mehrere Listen haben.
- Sie wollen eine neue Liste erstellen, in der alle Elemente von jeder der existierenen Listen drin sind.
- Wir werden eine Funktion `flatten` schreiben, die sogar eine noch eine allgemeinere Variante dieser Idee implementiert.
- Sie soll ein `Iterable` von `Iterables` als Parameter akzeptieren.
- Weil Listen ja auch `Iterables` sind, können wir also eine `list` von `listss` als argument hereingeben.
- Wir können aber auch ein `tuple` von `sets` angeben, wenn wir wollen.
- Der Rückgabewert von `flatten` soll jedenfalls eine `list` mit allen Elementen aus den „inneren“ `Iterables` sein.

Implementierung und Doctests

- Implementieren wir das also.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Implementieren wir das also.
- `flatten` erstellt eine Liste von seinem Parameter `iterables` über den List Comprehension Ausdruck `[value for subiterable in iterables for value in subiterable]`.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Implementieren wir das also.
- `flatten` erstellt eine Liste von seinem Parameter `iterables` über den List Comprehension Ausdruck `[value for subiterable in iterables for value in subiterable]`.
- Die Variable `subiterable` iterateriert über die `iterables`, entspricht also jeweils einer der Unterlisten.

```
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Implementieren wir das also.
- `flatten` erstellt eine Liste von seinem Parameter `iterables` über den List Comprehension Ausdruck `[value for subiterable in iterables for value in subiterable]`.
- Die Variable `subiterable` iterateriert über die `iterables`, entspricht also jeweils einer der Unterlisten.
- Dann iteriert `value` über die Elements des `subiterable`.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Implementieren wir das also.
- `flatten` erstellt eine Liste von seinem Parameter `iterables` über den List Comprehension Ausdruck `[value for subiterable in iterables for value in subiterable]`.
- Die Variable `subiterable` iterateriert über die `iterables`, entspricht also jeweils einer der Unterlisten.
- Dann iteriert `value` über die Elements des `subiterable`.
- Es nimmt also nach und nach jeden Wert in der Unterliste an,

```
1  """A utility for flatten sequences of sequences."""
2
3  from typing import Iterable
4
5
6  def flatten(iterables: Iterable[Iterable]) -> list:
7      """
8      Flatten an :class:`Iterable` of `Iterable`s to a flat list.
9
10     :param iterables: the `Iterable` containing other `Iterable`s.
11     :return: a list with all the contents of the nested `Iterable`s.
12
13     >>> flatten([[1, 2, 3], [4, 5, 6]])
14     [1, 2, 3, 4, 5, 6]
15
16     >>> flatten([[1, 2, 3], [], [4, 5, 6]])
17     [1, 2, 3, 4, 5, 6]
18
19     >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
20     [[1], [2], [3], [4], [5], [6]]
21
22     >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
23     [1, 2, 3, 4, 5, 6, 'a', 'b']
24     """
25     return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- `flatten` erstellt eine Liste von seinem Parameter `iterables` über den List Comprehension Ausdruck `[value for subiterable in iterables for value in subiterable]`.
- Die Variable `subiterable` iterateriert über die `iterables`, entspricht also jeweils einer der Unterlisten.
- Dann iteriert `value` über die Elements des `subiterable`.
- Es nimmt also nach und nach jeden Wert in der Unterliste an,
- Da wir eine `list` mit all diesen Werten zurückliefern, flachen wir die Liste von Listen.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Die Variable `subiterable` iterateriert über die `iterables`, entspricht also jeweils einer der Unterlisten.
- Dann iteriert `value` über die Elements des `subiterable`.
- Es nimmt also nach und nach jeden Wert in der Unterliste an,
- Da wir eine `list` mit all diesen Werten zurückliefern, flachen wir die Liste von Listen.
- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Dann iteriert `value` über die Elements des `subiterable`.
- Es nimmt also nach und nach jeden Wert in der Unterliste an,
- Da wir eine `list` mit all diesen Werten zurückliefern, flachen wir die Liste von Listen.
- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.

```
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Es nimmt also nach und nach jeden Wert in der Unterliste an,
- Da wir eine `list` mit all diesen Werten zurückliefern, flachen wir die Liste von Listen.
- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.

```
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Da wir eine `list` mit all diesen Werten zurückliefern, flachen wir die Liste von Listen.
- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Wir fügen dafür ein kleines Schnipsel Python-Kode gefolgt von seinem erwarteten Output ein.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Wir fügen dafür ein kleines Schnipsel Python-Kode gefolgt von seinem erwarteten Output ein. Die erste Zeile des Kodes hat das Präfix `>>>`.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Wir fügen dafür ein kleines Schnipsel Python-Kode gefolgt von seinem erwarteten Output ein. Die erste Zeile des Kodes hat das Präfix `>>>`. Wenn der Kode mehrere Zeilen braucht, dann haben weitere Zeilen das Präfix `...`.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Wir fügen dafür ein kleines Schnipsel Python-Kode gefolgt von seinem erwarteten Output ein. Die erste Zeile des Kodes hat das Präfix `>>>`. Wenn der Kode mehrere Zeilen braucht, dann haben weitere Zeilen das Präfix `...`. Nach dem Schnipsel, schreiben wir die erwartete Ausganbe.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Die Doctests können von Modulen wie `doctest`[5] oder Werkzeugen wie pytest[9] ausgeführt werden.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Die Doctests können von Modulen wie `doctest`[5] oder Werkzeugen wie pytest[9] ausgeführt werden. Diese extrahieren den Kode, führen ihn aus, und vergleichen die Ausgabe mit der erwarteten Ausgabe.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Die Doctests können von Modulen wie `doctest`[5] oder Werkzeugen wie pytest[9] ausgeführt werden. Diese extrahieren den Kode, führen ihn aus, und vergleichen die Ausgabe mit der erwarteten Ausgabe. Wenn sie nicht übereinstimmen, dann schlägt der Test fehl.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.

Nützliches Werkzeug

Ein Doctest ist Unit Test, der direkt in den Docstring einer Funktion, Klasse, oder eines Moduls geschrieben ist. Die Doctests können von Modulen wie `doctest`[5] oder Werkzeugen wie pytest[9] ausgeführt werden. Diese extrahieren den Kode, führen ihn aus, und vergleichen die Ausgabe mit der erwarteten Ausgabe. Wenn sie nicht übereinstimmen, dann schlägt der Test fehl. Wir nehmen hier immer pytest.

Implementierung und Doctests

- Wahrscheinlich ist es verwirrend, dass die innere `for`-Schleife eigentlich als „äußere“ Schleife ausgeführt wird und umgekehrt.
- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.
- Doctests zu benutzen hat einen weiteren einzigartigen Vorteil.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wir haben dieses Verhalten aber schon gesehen, als wir seiner Zeit `[f"{m}{n}"for m in "abc" for n in "xy"]` berechnet hatten.
- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.
- Doctests zu benutzen hat einen weiteren einzigartigen Vorteil:
- Es erlaubt uns, Beispiele wie unser Kode zu benutzen ist direkt in den Docstring einzufügen.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Normalerweise würden wir jetzt Kode bereitstellen, der `flatten` ausführt und ihre Ausgabe präsentiert.
- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.
- Doctests zu benutzen hat einen weiteren einzigartigen Vorteil:
- Es erlaubt uns, Beispiele wie unser Kode zu benutzen ist direkt in den Docstring einzufügen.
- Und diese Beispiele dienen auch noch gleich als Unit Tests!

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Diesmal machen wir etwas anderes.
- Wir zeigen Doctests für `flatten`.
- Doctests zu benutzen hat einen weiteren einzigartigen Vorteil:
- Es erlaubt uns, Beispiele wie unser Kode zu benutzen ist direkt in den Docstring einzufügen.
- Und diese Beispiele dienen auch noch gleich als Unit Tests!
- Schauen wir uns die Beispiele mal gleich an.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wir zeigen Doctests für `flatten`.
- Doctests zu benutzen hat einen weiteren einzigartigen Vorteil:
- Es erlaubt uns, Beispiele wie unser Kode zu benutzen ist direkt in den Docstring einzufügen.
- Und diese Beispiele dienen auch noch gleich als Unit Tests!
- Schauen wir uns die Beispiele mal gleich an.
- Der erste Doctest sagt uns, dass wenn wir `flatten([[1, 2, 3], [4, 5, 6]])` aufrufen, wir das Ergebnis `[1, 2, 3, 4, 5, 6]` bekommen werden.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Es erlaubt uns, Beispiele wie unser Kode zu benutzen ist direkt in den Docstring einzufügen.
- Und diese Beispiele dienen auch noch gleich als Unit Tests!
- Schauen wir uns die Beispiele mal gleich an.
- Der erste Doctest sagt uns, dass wenn wir `flatten([[1, 2, 3], [4, 5, 6]])` aufrufen, wir das Ergebnis `[1, 2, 3, 4, 5, 6]` bekommen werden.
- Mit anderen Worten, unsere Funktion wird die Liste aus zwei Listen in eine einzige flache Liste zusammenfügen.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Schauen wir uns die Beispiele mal gleich an.
- Der erste Doctest sagt uns, dass wenn wir `flatten([[1, 2, 3], [4, 5, 6]])` aufrufen, wir das Ergebnis `[1, 2, 3, 4, 5, 6]` bekommen werden.
- Mit anderen Worten, unsere Funktion wird die Liste aus zwei Listen in eine einzige flache Liste zusammenfügen.
- Die Funktion hat eine einzige flache Liste aus 6 Elementen von der Liste mit zwei Listen zu je 3 Elementen erstellt.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Mit anderen Worten, unsere Funktion wird die Liste aus zwei Listen in eine einzige flache Liste zusammenfügen.
- Die Funktion hat eine einzige flache Liste aus 6 Elementen von der Liste mit zwei Listen zu je 3 Elementen erstellt.
- Als Nächstes sehen wir, dass `flatten([[1, 2, 3], [], [4, 5, 6]])` ebenfalls `[1, 2, 3, 4, 5, 6]` produzieren soll.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Mit anderen Worten, unsere Funktion wird die Liste aus zwei Listen in eine einzige flache Liste zusammenfügen.
- Die Funktion hat eine einzige flache Liste aus 6 Elementen von der Liste mit zwei Listen zu je 3 Elementen erstellt.
- Als Nächstes sehen wir, dass `flatten([[1, 2, 3], [], [4, 5, 6]])` ebenfalls `[1, 2, 3, 4, 5, 6]` produzieren soll.
- Die leere Liste, die in zweiter Stelle in der „großen“ Liste vorkommt, verschwindet einfach.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Mit anderen Worten, unsere Funktion wird die Liste aus zwei Listen in eine einzige flache Liste zusammenfügen.
- Die Funktion hat eine einzige flache Liste aus 6 Elementen von der Liste mit zwei Listen zu je 3 Elementen erstellt.
- Als Nächstes sehen wir, dass `flatten([[1, 2, 3], [], [4, 5, 6]])` ebenfalls `[1, 2, 3, 4, 5, 6]` produzieren soll.
- Die leere Liste, die in zweiter Stelle in der „großen“ Liste vorkommt, verschwindet einfach.
- Sie hatte ja auch keine Elemente.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Die Funktion hat eine einzige flache Liste aus 6 Elementen von der Liste mit zwei Listen zu je 3 Elementen erstellt.
- Als Nächstes sehen wir, dass `flatten([[1, 2, 3], [], [4, 5, 6]])` ebenfalls `[1, 2, 3, 4, 5, 6]` produzieren soll.
- Die leere Liste, die in zweiter Stelle in der „großen“ Liste vorkommt, verschwindet einfach.
- Sie hatte ja auch keine Elemente.
- `flatten` reduziert zwei Listen-Ebenen zu einer.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Als Nächstes sehen wir, dass `flatten([[1, 2, 3], [], [4, 5, 6]])` ebenfalls `[1, 2, 3, 4, 5, 6]` produzieren soll.
- Die leere Liste, die in zweiter Stelle in der „großen“ Liste vorkommt, verschwindet einfach.
- Sie hatte ja auch keine Elemente.
- `flatten` reduziert zwei Listen-Ebenen zu einer.
- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Die leere Liste, die in zweiter Stelle in der „großen“ Liste vorkommt, verschwindet einfach.
- Sie hatte ja auch keine Elemente.
- `flatten` reduziert zwei Listen-Ebenen zu einer.
- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.
- Unsere Funktion reduziert eine Listenebene hinweg.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Sie hatte ja auch keine Elemente.
- `flatten` reduziert zwei Listen-Ebenen zu einer.
- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.
- Unsere Funktion reduziert eine Listenebene hinweg.
- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).

```
1  """A utility for flatten sequences of sequences."""
2
3  from typing import Iterable
4
5
6  def flatten(iterables: Iterable[Iterable]) -> list:
7      """
8      Flatten an :class:`Iterable` of `Iterable`s to a flat list.
9
10     :param iterables: the `Iterable` containing other `Iterable`s.
11     :return: a list with all the contents of the nested `Iterable`s.
12
13     >>> flatten([[1, 2, 3], [4, 5, 6]])
14     [1, 2, 3, 4, 5, 6]
15
16     >>> flatten([[1, 2, 3], [], [4, 5, 6]])
17     [1, 2, 3, 4, 5, 6]
18
19     >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
20     [[1], [2], [3], [4], [5], [6]]
21
22     >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
23     [1, 2, 3, 4, 5, 6, 'a', 'b']
24     """
25     return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Sie hatte ja auch keine Elemente.
- `flatten` reduziert zwei Listen-Ebenen zu einer.
- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.
- Unsere Funktion reduziert eine Listenebene hinweg.
- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).
- Flatten soll ja mit beliebigen `Iterables` funktionieren.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- `flatten` reduziert zwei Listen-Ebenen zu einer.
- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.
- Unsere Funktion reduziert eine Listenebene hinweg.
- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).
- Flatten soll ja mit beliebigen `Iterables` funktionieren.
- Es akzeptiert also auch gemischten Input.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wir übergeben eine Liste-von-Listen-von-Listen an `flatten` als dritten Test, machen also `flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`.
- Unsere Funktion reduziert eine Listenebene hinweg.
- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).
- Flatten soll ja mit beliebigen `Iterables` funktionieren.
- Es akzeptiert also auch gemischten Input.
- Der letzte Doctest symbolisiert das.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Unsere Funktion reduziert eine Listenebene hinweg.
- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).
- Flatten soll ja mit beliebigen `Iterables` funktionieren.
- Es akzeptiert also auch gemischten Input.
- Der letzte Doctest symbolisiert das.
- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wir bekommen eine Liste-von-Listen: `[[1], [2], [3], [4], [5], [6]]`).
- Flatten soll ja mit beliebigen `Iterables` funktionieren.
- Es akzeptiert also auch gemischten Input.
- Der letzte Doctest symbolisiert das.
- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.
- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Flatten soll ja mit beliebigen `Iterables` funktionieren.
- Es akzeptiert also auch gemischten Input.
- Der letzte Doctest symbolisiert das.
- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.
- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.
- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Es akzeptiert also auch gemischten Input.
- Der letzte Doctest symbolisiert das.
- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.
- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.
- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.
- Die `for`-Schleife macht ja genau das implizit.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Der letzte Doctest symbolisiert das.
- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.
- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.
- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.
- Die `for`-Schleife macht ja genau das implizit.
- Daher tauchen dann auch nur die Schlüssel in unserer flachen Liste auf.

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))` ergibt `[1, 2, 3, 4, 5, 6, 'a', 'b']` als Ergebnis.
- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.
- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.
- Die `for`-Schleife macht ja genau das implizit.
- Daher tauchen dann auch nur die Schlüssel in unserer flachen Liste auf.
- . . . oder sollen zumindest auftauchen. . .

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Beachten Sie, dass nur die Schlüssel des Dictionaries auftauchen.
- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.
- Die `for`-Schleife macht ja genau das implizit.
- Daher tauchen dann auch nur die Schlüssel in unserer flachen Liste auf.
- . . . oder sollen zumindest auftauchen. . .
- . . . wenn unsere Funktion die Doctests auch besteht. . .

```python
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Implementierung und Doctests

- Wie wir gelernt haben, gibt uns `iter`, wenn wir es auf ein `dict` anwenden, einen `Iterator` nur über die Schlüssel des Dictionaries.
- Die `for`-Schleife macht ja genau das implizit.
- Daher tauchen dann auch nur die Schlüssel in unserer flachen Liste auf.
- . . . oder sollen zumindest auftauchen. . .
- . . . wenn unsere Funktion die Doctests auch besteht. . .
- Wir führen darum jetzt pytest mit dem zusätzlichen Parameter `--doctest-modules` aus.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ list_flatten_iterables.py
2 ========================== test session starts
     ↪ ==========================
3 collected 1 item
4 list_flatten_iterables.py .                                              [100%]
5 ========================== 1 passed in 0.01s
     ↪ ==========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Implementierung und Doctests

- Die `for`-Schleife macht ja genau das implizit.
- Daher tauchen dann auch nur die Schlüssel in unserer flachen Liste auf.
- . . . oder sollen zumindest auftauchen. . .
- . . . wenn unsere Funktion die Doctests auch besteht. . .
- Wir führen darum jetzt pytest mit dem zusätzlichen Parameter `--doctest-modules` aus.
- Das volle Kommando spezifiziert auch wieder ein Timeout von 10 Sekunden mit `--timeout=10`.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ list_flatten_iterables.py
2 ========================= test session starts
     ↪ ==========================
3 collected 1 item
4 list_flatten_iterables.py .                                          [100%]
5 ========================== 1 passed in 0.01s
     ↪ ==========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Implementierung und Doctests

- . . . oder sollen zumindest auftauchen. . .
- . . . wenn unsere Funktion die Doctests auch besteht. . .
- Wir führen darum jetzt pytest mit dem zusätzlichen Parameter `--doctest-modules` aus.
- Das volle Kommando spezifiziert auch wieder ein Timeout von 10 Sekunden mit `--timeout=10`.
- Wir haben auch zwei Argumente (`--no-header` und `--tb=short`) um die Ausgaben etwas abzukürzen, damit sie auf den Slides und im Buch gut aussehen.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ list_flatten_iterables.py
2 ========================= test session starts
     ↪ ==========================
3 collected 1 item
4 list_flatten_iterables.py .                                          [100%]
5 ========================== 1 passed in 0.01s
     ↪ ==========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Implementierung und Doctests

- ...wenn unsere Funktion die Doctests auch besteht...
- Wir führen darum jetzt pytest mit dem zusätzlichen Parameter `--doctest-modules` aus.
- Das volle Kommando spezifiziert auch wieder ein Timeout von 10 Sekunden mit `--timeout=10`.
- Wir haben auch zwei Argumente (`--no-header` und `--tb=short`) um die Ausgaben etwas abzukürzen, damit sie auf den Slides und im Buch gut aussehen.
- Eigentlich würde `pytest --doctest-modules fileOrDirToTest` alleine auch schon reichen.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ list_flatten_iterables.py
2 ========================= test session starts
     ↪ ==========================
3 collected 1 item
4 list_flatten_iterables.py .                                          [100%]
5 ========================== 1 passed in 0.01s
     ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Implementierung und Doctests

- Wir führen darum jetzt pytest mit dem zusätzlichen Parameter `--doctest-modules` aus.
- Das volle Kommando spezifiziert auch wieder ein Timeout von 10 Sekunden mit `--timeout=10`.
- Wir haben auch zwei Argumente (`--no-header` und `--tb=short`) um die Ausgaben etwas abzukürzen, damit sie auf den Slides und im Buch gut aussehen.
- Eigentlich würde `pytest --doctest-modules fileOrDirToTest` alleine auch schon reichen.
- Die Ausgabe bescheinigt uns, dass alle Doctests bestanden haben.

```
1 $ pytest --timeout=10 --no-header --tb=short --doctest-modules
     ↪ list_flatten_iterables.py
2 ========================== test session starts
     ↪ ==========================
3 collected 1 item
4 list_flatten_iterables.py .                                            [100%]
5 ========================== 1 passed in 0.01s
     ↪ ==========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Zusammenfassung

Zusammenfassung



Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten.

Zusammenfassung



Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll.

Zusammenfassung



Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll. Weil Doctests normalerweise klein sind, sind sie eine schnelle und elegante Methode, um größere Unit Tests in anderen Dateien zu ergänzen.

Zusammenfassung

Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll. Weil Doctests normalerweise klein sind, sind sie eine schnelle und elegante Methode, um größere Unit Tests in anderen Dateien zu ergänzen.

- Wir haben hier Doctests über die Kommandozeile ausgeführt.

Zusammenfassung



Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll. Weil Doctests normalerweise klein sind, sind sie eine schnelle und elegante Methode, um größere Unit Tests in anderen Dateien zu ergänzen.

- Wir haben hier Doctests über die Kommandozeile ausgeführt.
- Sie können diese aber auch in PyCharm direkt ausführen.

Zusammenfassung

Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll. Weil Doctests normalerweise klein sind, sind sie eine schnelle und elegante Methode, um größere Unit Tests in anderen Dateien zu ergänzen.

- Wir haben hier Doctests über die Kommandozeile ausgeführt.
- Sie können diese aber auch in PyCharm direkt ausführen.
- Das gucken wir uns ein ander Mal an.

Zusammenfassung

Gute Praxis

Immer wenn möglich sollten Docstrings von Funktionen, Klassen, und Modulen Doctests beinhalten. Damit stellen wir Unit Tests als Beispiele zur Verfügung, die zeigen, wie unser Kode verwendet werden soll. Weil Doctests normalerweise klein sind, sind sie eine schnelle und elegante Methode, um größere Unit Tests in anderen Dateien zu ergänzen.

- Wir haben hier Doctests über die Kommandozeile ausgeführt.
- Sie können diese aber auch in PyCharm direkt ausführen.
- Das gucken wir uns ein ander Mal an.
- Auf jeden Fall sollten wir immer Doctests verwenden.

谢谢您们！

Thank you!

Vielen Dank!



References I

[1] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 95).

[2] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dez. 2024. ISBN: 978-1-990783-55-5 (siehe S. 94).

[3] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 95).

[4] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 94).

[5] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/doctest.html (besucht am 2024-11-07) (siehe S. 14–22, 48–51, 94).

[6] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: https://peps.python.org/pep-0257 (besucht am 2024-07-27) (siehe S. 94).

[7] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 95).

[8] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (besucht am 2024-08-23) (siehe S. 94).

[9] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: https://docs.pytest.org/en/stable/how-to/doctest.html (besucht am 2024-11-07) (siehe S. 48–51, 94).

[10] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: https://readthedocs.org/projects/pytest/downloads/pdf/latest (besucht am 2024-11-07) (siehe S. 95).

References II

[11] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 95).

[12] Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4281-0 (siehe S. 94).

[13] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 95).

[14] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 95).

[15] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: https://docs.astral.sh/ruff (besucht am 2024-08-23) (siehe S. 95).

[16] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 95).

[17] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 95).

[18] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: https://www.researchgate.net/publication/255673967 (besucht am 2025-09-05) (siehe S. 95).

[19] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 95).

[20] 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. 94).

References III

[21] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 94).

[22] Ernest E. Rothman, Rich Rosen und Brian Jepson. *Mac OS X for Unix Geeks*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2008. ISBN: 978-0-596-52062-5 (siehe S. 95).

[23] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (siehe S. 95).

[24] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (siehe S. 94).

[25] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: 978-1-80602-393-6 (siehe S. 95).

[26] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 94).

[27] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials (2025)*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: 978-1-80580-309-6 (siehe S. 94, 95).

[28] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: https://www.researchgate.net/publication/220094077 (besucht am 2024-10-01) (siehe S. 95).

[29] 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. 94, 95).

References IV

[30] Bruce M. Van Horn II und Quan Nguyen. *Hands-On Application Development with PyCharm*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-235-0 (siehe S. 95).

[31] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: https://peps.python.org/pep-0008 (besucht am 2024-07-27) (siehe S. 94).

[32] Thomas Weise (汤卫思). *Programming with C*. 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), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 94).

[33] 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), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 95).

[34] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 95).

[35] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 95).

Glossary (in English) I



C is a programming language, which is very successful in system programming situations[4,21,32].

CI *Continuous Integration* is a software development process where developers integrate new code into a codebase hosted in a Version Control System (VCS), after which automated tools run an automated build process including code analysis (such as linters) and unit test execution[12]. If the build succeeds and no errors or problems with the code are, the code may automatically be deployed (if the CI system is configured to do so).

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[6]. They must be delimited by `"""..."""`[6,31].

doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by `....`. These snippets can be executed by modules like `doctest`[5] or tools such as pytest[9]. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[26,29]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS[20,29]. Learn more at https://github.com.

IDE An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For Python, we recommend using PyCharm. On Apple systems, Xcode is often used.

iOS is the operating system that powers Apple iPhones[2,27]. Learn more at https://www.apple.com/ios.

iPadOS is the operating system that powers Apple iPads[2]. Learn more at https://www.apple.com/ipados.

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[8,24]. Ruff is an example of a linter used in the Python world.

Glossary (in English) II



macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers[22,27]. Learn more at https://www.apple.com/macos.

PyCharm is the convenient Python Integrated Development Environment (IDE) that we recommend for this course[30,34,35]. It comes in a free edition, so it can be downloaded and used at no cost. Learn more at https://www.jetbrains.com/pycharm.

pytest is a framework for writing and executing unit tests in Python[3,10,17,19,34]. Learn more at https://pytest.org.

Python The Python programming language[7,11,13,33], i.e., what you will learn about in our book[33]. Learn more at https://python.org.

Ruff is a linter and code formatting tool for Python[14,15]. Learn more at https://docs.astral.sh/ruff or in [33].

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification[1,16,18,19,23,28]. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[29]. 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.

Xcode is offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as macOS and iOS[25].