

合肥大學
HEFEI UNIVERSITY

Programming with Python

19. Tupel

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: 2026-11-04

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. Beispiele
3. Zusammenfassung

Einleitung

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden.

Einleitung

- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*).

Einleitung

- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.

Einleitung

- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten.

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s.

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s. (Der Python-Interpreter erzwingt das aber nicht.)

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s. (Der Python-Interpreter erzwingt das aber nicht.) Tupel sind von Anfang auch dazu gedacht, Objekte verschiedenen Typs zu beinhalten.

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s. (Der Python-Interpreter erzwingt das aber nicht.) Tupel sind von Anfang auch dazu gedacht, Objekte verschiedenen Typs zu beinhalten. Weil sie unveränderlich sind, ist immer klar, welchen Typ das Objekt an einem bestimmten Index hat.

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `ints`. (Der Python-Interpreter erzwingt das aber nicht.) Tupel sind von Anfang auch dazu gedacht, Objekte verschiedenen Typs zu beinhalten. Weil sie unveränderlich sind, ist immer klar, welchen Typ das Objekt an einem bestimmten Index hat.
  3. Tupel-Literale werden mit runden Klammern anstelle von eckigen Klammern definiert, also mit `(...)`.

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s. (Der Python-Interpreter erzwingt das aber nicht.) Tupel sind von Anfang auch dazu gedacht, Objekte verschiedenen Typs zu beinhalten. Weil sie unveränderlich sind, ist immer klar, welchen Typ das Objekt an einem bestimmten Index hat.
  3. Tupel-Literale werden mit runden Klammern anstelle von eckigen Klammern definiert, also mit `(...)`.

Gute Praxis

Wenn Sie eine indizierbare Sequenz von Objekten brauchen, benutzen Sie `list` nur wenn Sie die Sequenz auch verändern wollen. Wenn Sie nicht vor haben, die Sequenz zu verändern, benutzen Sie ein Tupel (`tuple`).

Einleitung



- Tupel (EN: *tuples*) sind Listen sehr ähnlich, mit drei wichtigen Unterschieden:
  1. Tupel sind unveränderlich (EN: *immutable*). Sie können keine Elemente zu einem Tupel hinzufügen, löschen, oder ändern.
  2. Listen sind dafür gedacht, Objekte des selben Typs zu beinhalten. Der Type-Hint `list[int]` definiert eine Liste von `int`s. (Der Python-Interpreter erzwingt das aber nicht.) Tupel sind von Anfang auch dazu gedacht, Objekte verschiedenen Typs zu beinhalten. Weil sie unveränderlich sind, ist immer klar, welchen Typ das Objekt an einem bestimmten Index hat.
  3. Tupel-Literale werden mit runden Klammern anstelle von eckigen Klammern definiert, also mit `(...)`.
- Und das lernen wir jetzt.

Gute Praxis

Wenn Sie eine indizierbare Sequenz von Objekten brauchen, benutzen Sie `list` nur wenn Sie die Sequenz auch verändern wollen. Wenn Sie nicht vor haben, die Sequenz zu verändern, benutzen Sie ein Tupel (`tuple`).

Beispiele



Erstellen, Type Hints, Indizieren

- Tupel-Literale werden mit runden Klammern deklariert.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ python3 tuples_1.py ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Tupel-Literale werden mit runden Klammern deklariert.
- Der Type-Hint `tuple[A, B]` deklariert ein Tupel aus zwei Elementen, wobei das erste Element vom Typ `A` und das zweite vom Typ `B` ist[5].

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Tupel-Literale werden mit runden Klammern deklariert.
- Der Type-Hint `tuple[A, B]` deklariert ein Tupel aus zwei Elementen, wobei das erste Element vom Typ `A` und das zweite vom Typ `B` ist[5].
- Wir können so Tupel mit beliebig vielen typisierten Elementen deklarieren.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Tupel-Literale werden mit runden Klammern deklariert.
- Der Type-Hint `tuple[A, B]` deklariert ein Tupel aus zwei Elementen, wobei das erste Element vom Typ `A` und das zweite vom Typ `B` ist[5].
- Wir können so Tupel mit beliebig vielen typisierten Elementen deklarieren.
- `len(t)` liefert die Anzahl der Elemente im Tupel `t`.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Der Type-Hint `tuple[A, B]` deklariert ein Tupel aus zwei Elementen, wobei das erste Element vom Typ `A` und das zweite vom Typ `B` ist[s].
- Wir können so Tupel mit beliebig vielen typisierten Elementen deklarieren.
- `len(t)` liefert die Anzahl der Elemente im Tupel `t`.
- Der Type-Hint `tuple[A, ...]` deklariert ein Tupel dessen Elemente alle vom Typ `A` sind, das aber ohne feste Längenangabe deklariert wird.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Wir können so Tupel mit beliebig vielen typisierten Elementen deklarieren.
- `len(t)` liefert die Anzahl der Elemente im Tupel `t`.
- Der Type-Hint `tuple[A, ...]` deklariert ein Tupel dessen Elemente alle vom Typ `A` sind, das aber ohne feste Längenangabe deklariert wird.
- Tupel können genau wie Listen und Strings indiziert und gesliced werden.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- Wir können so Tupel mit beliebig vielen typisierten Elementen deklarieren.
- `len(t)` liefert die Anzahl der Elemente im Tupel `t`.
- Der Type-Hint `tuple[A, ...]` deklariert ein Tupel dessen Elemente alle vom Typ `A` sind, das aber ohne feste Längenangabe deklariert wird.
- Tupel können genau wie Listen und Strings indiziert und gesliced werden.
- Auch der `in`- und der `not in`-Operator funktionieren.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Erstellen, Type Hints, Indizieren

- `len(t)` liefert die Anzahl der Elemente im Tupel `t`.
- Der Type-Hint `tuple[A, ...]` deklariert ein Tupel dessen Elemente alle vom Typ `A` sind, das aber ohne feste Längenangabe deklariert wird.
- Tupel können genau wie Listen und Strings indiziert und gesliced werden.
- Auch der `in`- und der `not in`-Operator funktionieren.
- Die Funktion `t.index(v)` liefert den Index des Elements `v` im Tupel `t`.

```python
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")    # first element of `veggies`.
print(f"{veggies[1] = }")    # second element of `veggies`.
print(f"{veggies[-1] = }")   # last element of `veggies`.
print(f"{veggies[-2] = }")   # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Tupel mit Elementen verschiedener Typen

- Wir können auch Tupel mit Elementen verschiedener Typen erstellen.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Wir können auch Tupel mit Elementen verschiedener Typen erstellen.
- `tuple[str, int, float]` ist ein Tupel wo das erste Element ein String, das zweite eine Ganzzahl und das dritte eine Fließkommazahl ist.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Wir können auch Tupel mit Elementen verschiedener Typen erstellen.
- `tuple[str, int, float]` ist ein Tupel wo das erste Element ein String, das zweite eine Ganzzahl und das dritte eine Fließkommazahl ist.
- Tupel (und Listen) unterstützen alle sechs Vergleichsoperationen `<`, `<=`, `==`, `>=`, `>`, und `!=`.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Wir können auch Tupel mit Elementen verschiedener Typen erstellen.
- `tuple[str, int, float]` ist ein Tupel wo das erste Element ein String, das zweite eine Ganzzahl und das dritte eine Fließkommazahl ist.
- Tupel (und Listen) unterstützen alle sechs Vergleichsoperationen `<`, `<=`, `==`, `>=`, `>`, und `!=`.
- Natürlich können wir Tupel auch in Listen oder Tupel reintun.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Wir können auch Tupel mit Elementen verschiedener Typen erstellen.
- `tuple[str, int, float]` ist ein Tupel wo das erste Element ein String, das zweite eine Ganzzahl und das dritte eine Fließkommazahl ist.
- Tupel (und Listen) unterstützen alle sechs Vergleichsoperationen `<`, `<=`, `==`, `>=`, `>`, und `!=`.
- Natürlich können wir Tupel auch in Listen oder Tupel reintun.
- Der Type-Hint `list[tuple[str, int, float]]` definiert eine Liste, die Tupel es oben genannten Typs beinhalten.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- `tuple[str, int, float]` ist ein Tupel wo das erste Element ein String, das zweite eine Ganzzahl und das dritte eine Fließkommazahl ist.
- Tupel (und Listen) unterstützen alle sechs Vergleichsoperationen `<`, `<=`, `==`, `>=`, `>`, und `!=`.
- Natürlich können wir Tupel auch in Listen oder Tupel reintun.
- Der Type-Hint `list[tuple[str, int, float]]` definiert eine Liste, die Tupel es oben genannten Typs beinhalted.
- Diese arbeiten dann lexikographisch elementweise.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Tupel (und Listen) unterstützen alle sechs Vergleichsoperationen `<`, `<=`, `==`, `>=`, `>`, und `!=`.
- Natürlich können wir Tupel auch in Listen oder Tupel reintun.
- Der Type-Hint `list[tuple[str, int, float]]` definiert eine Liste, die Tupel es oben genannten Typs beinhalten.
- Diese arbeiten dann lexikographisch elementweise.
- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Natürlich können wir Tupel auch in Listen oder Tupel reintun.
- Der Type-Hint `list[tuple[str, int, float]]` definiert eine Liste, die Tupel es oben genannten Typs beinhalted.
- Diese arbeiten dann lexikographisch elementweise.
- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Der Type-Hint `list[tuple[str, int, float]]` definiert eine Liste, die Tupel es oben genannten Typs beinhalted.
- Diese arbeiten dann lexikographisch elementweise.
- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`. Wenn das erste Element von `x` größer als das erste Element von `y` ist, dann gilt `x > y`.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Diese arbeiten dann lexikographisch elementweise.
- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`. Wenn das erste Element von `x` größer als das erste Element von `y` ist, dann gilt `x > y`. Wenn das erste Element von `x` gleich dem erste Element von `y` ist, dann geht der Vergleich mit den zweiten Elementen weiter, wenn beide Tupel mindestens 2 Elemente haben (andernfalls ist das kürzere Tupel kleiner).

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`. Wenn das erste Element von `x` größer als das erste Element von `y` ist, dann gilt `x > y`. Wenn das erste Element von `x` gleich dem erste Element von `y` ist, dann geht der Vergleich mit den zweiten Elementen weiter, wenn beide Tupel mindestens 2 Elemente haben (andernfalls ist das kürzere Tupel kleiner). Und so weiter.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`. Wenn das erste Element von `x` größer als das erste Element von `y` ist, dann gilt `x > y`. Wenn das erste Element von `x` gleich dem erste Element von `y` ist, dann geht der Vergleich mit den zweiten Elementen weiter, wenn beide Tupel mindestens 2 Elemente haben (andernfalls ist das kürzere Tupel kleiner). Und so weiter.
- Deshalb können wir Listen von Tupel sortieren.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Angenommen, wir vergleichen wir zwei Tupel `x` und `y`. Wenn das erste Element von `x` kleiner als das erste Element von `y` ist, dann gilt `x < y`. Wenn das erste Element von `x` größer als das erste Element von `y` ist, dann gilt `x > y`. Wenn das erste Element von `x` gleich dem erste Element von `y` ist, dann geht der Vergleich mit den zweiten Elementen weiter, wenn beide Tupel mindestens 2 Elemente haben (andernfalls ist das kürzere Tupel kleiner). Und so weiter.
- Deshalb können wir Listen von Tupel sortieren.
- Genau wie Listen können wir Tupel auf mehrere Variablen auspacken.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Tupel mit Elementen verschiedener Typen

- Deshalb können wir Listen von Tupel sortieren.
- Genau wie Listen können wir Tupel auf mehrere Variablen auspacken.
- In Fällen, wo es keine Verwechslung geben kann, können wir Tupel sogar ohne die runden Klammern definieren.

```python
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ python3 tuples_2.py ↓

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2),
    ↪ ('pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear',
    ↪ -2, 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ python3 tuples_3.py ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
       ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ python3 tuples_3.py ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
       ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ `python3 tuples_3.py` ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
       ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ `python3 tuples_3.py` ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
        ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit



- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.

Gute Praxis

Wenn Sie wirklich unveränderliche Datenstrukturen brauchen, dann packen Sie immer nur unveränderliche Objekte in Tupel.

Unveränderlichkeit



- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.

Gute Praxis

Wenn Sie wirklich unveränderliche Datenstrukturen brauchen, dann packen Sie immer nur unveränderliche Objekte in Tupel. Veränderliche Objekte in Tupel können immer noch verändert werden, wodurch die Tupel dann keine Konstanten mehr sind.

Unveränderlichkeit



- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.

Gute Praxis

Wenn Sie wirklich unveränderliche Datenstrukturen brauchen, dann packen Sie immer nur unveränderliche Objekte in Tupel. Veränderliche Objekte in Tupel können immer noch verändert werden, wodurch die Tupel dann keine Konstanten mehr sind. Programmierer nehmen aber oft an, dass Tupel Konstanten sind.

Unveränderlichkeit



- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.

Gute Praxis

Wenn Sie wirklich unveränderliche Datenstrukturen brauchen, dann packen Sie immer nur unveränderliche Objekte in Tupel. Veränderliche Objekte in Tupel können immer noch verändert werden, wodurch die Tupel dann keine Konstanten mehr sind. Programmierer nehmen aber oft an, dass Tupel Konstanten sind. Wenn man diese Annahme verletzt, dann kann das zu sehr eigenartigen Fehlern führen.

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ `python3 tuples_3.py` ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
        ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ `python3 tuples_3.py` ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
        ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Diese Liste `[2]` könne wir natürlich später verändern.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.
- Was sagt Mypy dazu?

```python
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ `python3 tuples_3.py` ↓

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError
        ↪ exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.
- Was sagt Mypy dazu? Es findet den Fehler.

```
$ mypy tuples_3.py --no-strict-optional --check-untyped-defs
tuples_3.py:11: error: Unsupported target for indexed assignment ("tuple
    ↪ [int, list[int]]")  [index]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.
- Was sagt Mypy dazu? Es findet den Fehler.
- Und was sagt Ruff dazu?

```
$ mypy tuples_3.py --no-strict-optional --check-untyped-defs
tuples_3.py:11: error: Unsupported target for indexed assignment ("tuple
    ↪ [int, list[int]]")  [index]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.

```
$ mypy tuples_3.py --no-strict-optional --check-untyped-defs
tuples_3.py:11: error: Unsupported target for indexed assignment ("tuple
    ↪ [int, list[int]]")  [index]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 tuples_3.py
All checks passed!
# ruff 0.17.3 succeeded with exit code 0.
```

- Was sagt Mypy dazu? Es findet den Fehler.
- Und was sagt Ruff dazu? Es findet ihn nicht, denn es sucht nicht nach Typ-Fehlern.

Unveränderlichkeit

- Prüfen wir nun, ob Tupel wirklich unveränderlich sind.
- Wir können z. B. ein veränderliches Objekt (die Liste) in ein Tupel tun.
- Das Tupel beinhaltet dann immer noch genau das selbe Listenobjekt, nur dass sich eben der Inhalt dieses Objekts geändert hat.
- Was aber nicht geht, ist ein anderes Objekt in ein Tupel reinzuschreiben.
- Dann crashed das Programm.

```
$ mypy tuples_3.py --no-strict-optional --check-untyped-defs
tuples_3.py:11: error: Unsupported target for indexed assignment ("tuple
    ↪ [int, list[int]]")  [index]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 tuples_3.py
All checks passed!
# ruff 0.17.3 succeeded with exit code 0.
```

- Was sagt Mypy dazu? Es findet den Fehler.
- Und was sagt Ruff dazu? Es findet ihn nicht, denn es sucht nicht nach Typ-Fehlern.
- Es ist also immer sinnvoll, unseren Kode mit mehreren Werkzeugen zu prüfen.

Zusammenfassung

Zusammenfassung

- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.
- Beide unterstützen slicing.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.
- Beide unterstützen slicing.
- Tupel sind unveränderlich, wohingegen Listen verändert werden können.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.
- Beide unterstützen slicing.
- Tupel sind unveränderlich, wohingegen Listen verändert werden können.
- Tupel-Literale werden mit runden Klammern (manchmal auch ganz ohne Klammern) deklariert, Listen-Literale mit eckigen Klammern.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.
- Beide unterstützen slicing.
- Tupel sind unveränderlich, wohingegen Listen verändert werden können.
- Tupel-Literale werden mit runden Klammern (manchmal auch ganz ohne Klammern) deklariert, Listen-Literale mit eckigen Klammern.
- Listen sind dafür gedacht, beliebig viele Elemente eines Datentyps zu beinhalten.

Zusammenfassung



- Tupel und Listen sind beides Kontainer für beliebige Anzahlen von Objekten.
- Beide können mit Ganzzahlen indiziert werden.
- Beide unterstützen slicing.
- Tupel sind unveränderlich, wohingegen Listen verändert werden können.
- Tupel-Literale werden mit runden Klammern (manchmal auch ganz ohne Klammern) deklariert, Listen-Literale mit eckigen Klammern.
- Listen sind dafür gedacht, beliebig viele Elemente eines Datentyps zu beinhalten.
- Tupel können Elemente verschiedener Datentypen beinhalten.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] 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. 65).

[2] 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. 65).

[3] 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. 65).

[4] 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. 66).

[5] Łukasz Langa. *Type Hinting Generics In Standard Collections*. Python Enhancement Proposal (PEP) 585. Beaverton, OR, USA: Python Software Foundation (PSF), 3. März 2019. URL: https://peps.python.org/pep-0585 (besucht am 2024-10-09) (siehe S. 17–21).

[6] 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. 65).

[7] 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. 65).

[8] 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. 65).

[9] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 65).

[10] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 65).

[11] 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. 65).

References II

[12] 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. 65).

[13] *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. 65).

[14] 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. 65).

[15] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 65).

[16] ."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. 65).

[17] 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. 65, 66).

[18] 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. 66).

[19] 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. 65).

[20] 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. 65).

Glossary (in English) I



C is a programming language, which is very successful in system programming situations [1,13,19].

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 [15,17]. Learn more at https://git-scm.com.

GitHub is a website where software projects can be hosted and managed via the Git VCS [12,17]. Learn more at https://github.com.

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles [3,14]. Ruff is an example of a linter used in the Python world.

literal A literal is a specific concrete value, something that is written down as-is [7,16]. 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`.

Mypy is a static type checking tool for Python [8] that makes use of type hints. Learn more at https://github.com/python/mypy and in [20].

Python The Python programming language [2,6,9,20], i.e., what you will learn about in our book [20]. Learn more at https://python.org.

Ruff is a linter and code formatting tool for Python [10,11]. Learn more at https://docs.astral.sh/ruff or in [20].

Glossary (in English) II



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 [4,18]. 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.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code [17]. 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.