合肥大學
HEFEI UNIVERSITY

Programming with Python

45. Klassen/Dunder: __hash__

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

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

应用优化研究所
人工智能与大数据学院
合肥大学
中国安徽省合肥市

Version: 2025-10-30

Programming with Python

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

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

Outline

Einleitung

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr. Ich wollte es nur nochmal sagen.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr. Ich wollte es nur nochmal sagen.
  2. Es muss eine Dunder-Methode `__eq__` geben, die eine Instanz von `Points` auf Gleichheit mit anderen Objekten vergleicht.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr. Ich wollte es nur nochmal sagen.
  2. Es muss eine Dunder-Methode `__eq__` geben, die eine Instanz von `Points` auf Gleichheit mit anderen Objekten vergleicht. Auch die habe wir schon.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr. Ich wollte es nur nochmal sagen.
  2. Es muss eine Dunder-Methode `__eq__` geben, die eine Instanz von `Points` auf Gleichheit mit anderen Objekten vergleicht. Auch die habe wir schon.
  3. Die Dunder-Methode `__hash__` muss implementiert sein, die einen Hash-Wert eines `Point`-Objekts in Form eines `int` liefert. Die fehlt noch.

Einleitung

- Unsere `Point`-Objekte sind unveränderlich.
- Sie speichern nur zwei numerische Koordination.
- Vielleicht könnte es eine Situation geben, wo jemand solche Punkte als Schlüssel für ein Dictionary verwenden will.
- Oder jemand will eine Menge von Punkten erstellen.
- Damit so etwas geht, müssen drei Bedingungen erfüllt werden:
  1. Instanzen von `Point` müssen unveränderlich sein. Das ist schon wahr. Ich wollte es nur nochmal sagen.
  2. Es muss eine Dunder-Methode `__eq__` geben, die eine Instanz von `Points` auf Gleichheit mit anderen Objekten vergleicht. Auch die habe wir schon.
  3. Die Dunder-Methode `__hash__` muss implementiert sein, die einen Hash-Wert eines `Point`-Objekts in Form eines `int` liefert. Die fehlt noch.
- Wenn wir diese Kriterien erfüllen, dann können `Points` in Mengen gespeichert oder als Schlüssel in Dictionaries verwendet werden.

__eq__ und __hash__

`__eq__` und `__hash__`

- Wir müssten also nur eine weitere Methode, implementieren nämlich `__hash__`.

`__eq__` und `__hash__`

- Wir müssten also nur eine weitere Methode, implementieren nämlich `__hash__`.
- Für die zwei Dunder-Methoden `__eq__` und `__hash__` muss immer gelten[3]:

`__eq__` und `__hash__`

- Wir müssten also nur eine weitere Methode, implementieren nämlich `__hash__`.
- Für die zwei Dunder-Methoden `__eq__` und `__hash__` muss immer gelten[3]:

$$\texttt{a.\_\_eq\_\_(b)} \Rightarrow \texttt{a.\_\_hash\_\_()} = \texttt{b.\_\_hash\_\_()} \tag{1}$$

`__eq__` und `__hash__`

- Wir müssten also nur eine weitere Methode, implementieren nämlich `__hash__`.
- Für die zwei Dunder-Methoden `__eq__` und `__hash__` muss immer gelten[3]:

$$\texttt{a.\_\_eq\_\_(b)} \Rightarrow \texttt{a.\_\_hash\_\_()} = \texttt{b.\_\_hash\_\_()} \tag{1}$$

- Das ist das selbe wie[3,5]:

$$\texttt{a == b} \Rightarrow \texttt{hash(a)} = \texttt{hash(b)} \tag{2}$$

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.

Ganzzahlige Hash-Tabellen

- Machen wir mal einen Schritt zurück.
- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.

Ganzzahlige Hash-Tabellen

- Was ist überhaup ein Hash-Wert?
- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].

Ganzzahlige Hash-Tabellen

- Warum brauchen wir diese ganzzahligen Hash-Werte?
- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.

Ganzzahlige Hash-Tabellen

- Und warum müssen gleiche Objekte die gleichen Hash-Werte haben?
- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.

Ganzzahlige Hash-Tabellen

- Dictionaries in Python (und Java) benutzen intern Tabellen, in denen die Schlüssel-Wert-Paare gespeichert sind[9,21].
- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}($`len(l)`$)$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?

Ganzzahlige Hash-Tabellen

- Mengen machen das selbe, speichern aber nur Schlüssel.
- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}(\texttt{len(l)})$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}(\texttt{len(l)})$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?

Ganzzahlige Hash-Tabellen

- Die internen Tabellen funktionieren so ähnlich wie lineare Listen.
- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}(\texttt{len(l)})$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}(\texttt{len(l)})$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.

Ganzzahlige Hash-Tabellen

- Anders als bei Listen werden neue Elemente nicht am Ende angehängt.
- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}($`len(l)`$)$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.

Ganzzahlige Hash-Tabellen

- Stattdessen funktionieren sie eher wie Listen fester Länge, bei denen neue Elemente an bestimmten Indizes gespeichert werden, wo sie wieder gefunden werden können.
- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}($`len(l)`$)$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.

Ganzzahlige Hash-Tabellen

- Hash-Tabellen[7,14,26] sind sehr schnell.
- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}($`len(l)`$)$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}($`len(l)`$)$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).

Ganzzahlige Hash-Tabellen

- Sie haben eine Element-weise Lese-, Such-, und Update-Komplexität von $\mathcal{O}(1)$[1,10,23].
- Ein Element in einer Liste `l` kann in $\mathcal{O}(\texttt{len(l)})$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}(\texttt{len(l)})$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.

Ganzzahlige Hash-Tabellen

- Ein Element in einer Liste `l` kann in $\mathcal{O}(\texttt{len(l)})$ gesucht werden, also ist ein bestimmtes Element in einer Liste zu finden viel langsamer.
- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}(\texttt{len(l)})$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.
- Natürlich war das eine sehr krasse Vereinfachung[9,21].

Ganzzahlige Hash-Tabellen

- Wie gesagt können Sie sich vorstellen, dass Hash-Tabellen intern Listen als Speicher verwenden.
- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}($`len(l)`$)$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.
- Natürlich war das eine sehr krasse Vereinfachung[9,21].
- Es könnte z. B. verschiedene Ganzzahlen mit dem selben Ergebnis für `i % len(l)` geben.

Ganzzahlige Hash-Tabellen

- Wir können wir also eine Liste `l` mit Such-Komplexität $\mathcal{O}(\texttt{len(l)})$ in eine Hash-Tabelle mit Such-Komplexität $\mathcal{O}(1)$ umbauen?
- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.
- Natürlich war das eine sehr krasse Vereinfachung[9,21].
- Es könnte z. B. verschiedene Ganzzahlen mit dem selben Ergebnis für `i % len(l)` geben.
- Dictionaries und Sets müssen also mit solchen Kollisionen umgehen können.

Ganzzahlige Hash-Tabellen

- Stellen wir uns erstmal vor, das wir nur Ganzzahlen speichern wollen und das unsere Liste viel größer ist, als die Anzahl der Elemente die wir speichern wollen.
- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.
- Natürlich war das eine sehr krasse Vereinfachung[9,21].
- Es könnte z. B. verschiedene Ganzzahlen mit dem selben Ergebnis für `i % len(l)` geben.
- Dictionaries und Sets müssen also mit solchen Kollisionen umgehen können.
- Sie müssen auch wachsen können, wenn sie langsam gefüllt werden.

Ganzzahlige Hash-Tabellen

- Wir benutzen eine Liste von `int | None` Werten und sie ist anfänglich nur mit `None` gefüllt.
- Wenn wir eine Ganzzahl `i` speichern wollen, dann berechnen wir den Rest der Division durch die Listenlänge, also `i % len(l)`, und platzieren das Element an diesem Index.
- Wenn wir wissen wollen, ob eine Ganzzahl `j` in der Liste ist, dann berechnen wir wieder `j % len(l)` und prüfen, ob es gleich dem Element an diesem Index ist (via `__eq__`).
- Beide Operationen, Einfügen und Suchen, funktionieren nun in $\mathcal{O}(1)$.
- Natürlich war das eine sehr krasse Vereinfachung[9,21].
- Es könnte z. B. verschiedene Ganzzahlen mit dem selben Ergebnis für `i % len(l)` geben.
- Dictionaries und Sets müssen also mit solchen Kollisionen umgehen können.
- Sie müssen auch wachsen können, wenn sie langsam gefüllt werden.
- Aber das ist zumindest die grundlegende Idee.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.

Warum Hash-Werte?

- Weil wir auch andere Objekte, die keine Ganzzahlen sind, als Schlüssel verwenden wollen, brauchen wir eine Möglichkeit, diese zu Ganzzahlen umzurechnen.
- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.

Warum Hash-Werte?

- Das ist, was `__hash__` machen soll.
- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.

Warum Hash-Werte?

- Es muss keine bijektive Funktion sein, also die „Umrechnung“ muss nur in Richtung `int` functionieren.
- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.

Warum Hash-Werte?

- Es muss auch nicht injektiv sein, also verschiedene Objekte dürfen den gleichen Hash-Wert haben.
- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.

Warum Hash-Werte?

- Das sollte aber so weit wie möglich vermieden werden.
- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.
- Wenn nicht, dann nicht.

Warum Hash-Werte?

- Die Dictionary- oder Mengen-Implementierung ist dann dafür verantwortlich, diese ganzzahligen Hash-Werte in Indizes für ihre internen Tabellen umzurechnen.
- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.
- Wenn nicht, dann nicht.
- Und wie gesagt, das ist komplizierter in der Realität, z. B. kann es verschiedene Objekte mi den selben Hash-Kodes geben und die Menge muss solche Kollisionen irgendwie auflösen ... aber für uns reicht diese prinzipielle Idee.

Warum Hash-Werte?

- Also werden die Hash-Werte gebraucht, um Objekte in Dictionaries und Mengen zu finden.
- Wir wollen wissen, ob ein Objekt `a in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.
- Wenn nicht, dann nicht.
- Und wie gesagt, das ist komplizierter in der Realität, z. B. kann es verschiedene Objekte mi den selben Hash-Kodes geben und die Menge muss solche Kollisionen irgendwie auflösen ... aber für uns reicht diese prinzipielle Idee.
- Vereinfacht gesagt gilt also, dass wenn wir ein Objekt in die Menge `s` tun wollen, dessen Hash-Wert berechnet und dann in einen Index für die interne Tabelle umgerechnet wird.

Warum Hash-Werte?

- Wir wollen wissen, ob ein Objekt `a` `in` der Menge `s` ist?
- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.
- Wenn nicht, dann nicht.
- Und wie gesagt, das ist komplizierter in der Realität, z. B. kann es verschiedene Objekte mi den selben Hash-Kodes geben und die Menge muss solche Kollisionen irgendwie auflösen ... aber für uns reicht diese prinzipielle Idee.
- Vereinfacht gesagt gilt also, dass wenn wir ein Objekt in die Menge `s` tun wollen, dessen Hash-Wert berechnet und dann in einen Index für die interne Tabelle umgerechnet wird.
- Dictionaries funktionieren genauso, nur speichern sie Schlüssel-Wert-Paare und berechnen den Hash-Code nur von den Schlüsseln.

Warum Hash-Werte?

- Dann benutzt die Menge `s` die Funktion `hash(a)`, welche dann `a.__hash__()` aufruft, um den Hash-Wert von `a` zu bekommen.
- Das funktioniert so ähnlich wie `repr(a)`, das `a.__repr__()` aufruft, wenn es definiert ist.
- Dann übersetzt `s` den Hash-Wert in einen Index.
- Dann prüft es, ob da ein Objekt `b` an diesem Index ist mit `b == a`.
- Wenn ja, dann ist `a` in der Menge `s` und pythonila in s liefert `True`.
- Wenn nicht, dann nicht.
- Und wie gesagt, das ist komplizierter in der Realität, z. B. kann es verschiedene Objekte mi den selben Hash-Kodes geben und die Menge muss solche Kollisionen irgendwie auflösen ... aber für uns reicht diese prinzipielle Idee.
- Vereinfacht gesagt gilt also, dass wenn wir ein Objekt in die Menge `s` tun wollen, dessen Hash-Wert berechnet und dann in einen Index für die interne Tabelle umgerechnet wird.
- Dictionaries funktionieren genauso, nur speichern sie Schlüssel-Wert-Paare und berechnen den Hash-Code nur von den Schlüsseln.
- Sie können die Details in [9, 21] nachlesen.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.
- Deshalb ist auch klar, dass sich Objekte, die wir als Schlüssel in Dictionaries verwenden oder in Mengen speichern niemals verändern dürfen, also unveränderlich seien müssen.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.
- Deshalb ist auch klar, dass sich Objekte, die wir als Schlüssel in Dictionaries verwenden oder in Mengen speichern niemals verändern dürfen, also unveränderlich seien müssen.
- Es ist auch klar, dass wenn zwei Objekte `a` und `b` gleich sind, sie dann auch den selben Hash-Wert haben müssen.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.
- Deshalb ist auch klar, dass sich Objekte, die wir als Schlüssel in Dictionaries verwenden oder in Mengen speichern niemals verändern dürfen, also unveränderlich seien müssen.
- Es ist auch klar, dass wenn zwei Objekte `a` und `b` gleich sind, sie dann auch den selben Hash-Wert haben müssen.
- Das folgt schon daraus, das für zwei gleich Objekte `a` und `b` gelten muss, das `a in s` = `b in s`.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.
- Deshalb ist auch klar, dass sich Objekte, die wir als Schlüssel in Dictionaries verwenden oder in Mengen speichern niemals verändern dürfen, also unveränderlich seien müssen.
- Es ist auch klar, dass wenn zwei Objekte `a` und `b` gleich sind, sie dann auch den selben Hash-Wert haben müssen.
- Das folgt schon daraus, das für zwei gleich Objekte `a` und `b` gelten muss, das `a in s` = `b in s`.
- Sonst könnte es ja sein, dass `"123" in s` für das String Literal `"123"` das Ergebnis `True` liefert, aber `False` für `str(123)in s`.

`__eq__` und `__hash__`

- Es ist klar, dass wenn wir `__hash__` zweimal für das selbe Objekt `a` aufrufen, wir auch zweimal den selben Wert bekommen müssen.
- Weil der Hash-Wert benutzt wird, um Objekte in Mengen und Dictionaries zu finden, darf er sich niemals ändern.
- Deshalb ist auch klar, dass sich Objekte, die wir als Schlüssel in Dictionaries verwenden oder in Mengen speichern niemals verändern dürfen, also unveränderlich seien müssen.
- Es ist auch klar, dass wenn zwei Objekte `a` und `b` gleich sind, sie dann auch den selben Hash-Wert haben müssen.
- Das folgt schon daraus, das für zwei gleich Objekte `a` und `b` gelten muss, das `a in s` = `b in s`.
- Sonst könnte es ja sein, dass `"123" in s` für das String Literal `"123"` das Ergebnis `True` liefert, aber `False` für `str(123)in s`.
- Das würde ja überhaupt keinen Sinn ergeben.

Objekte „Hash-bar“ machen

> *. . . The* `__hash__()` *method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a* `tuple` *and hashing the tuple.*
>
> — *[3], 2001*

Objekte „Hash-bar“ machen

> *. . . The* `__hash__()` *method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a* `tuple` *and hashing the tuple.*
>
> — *[3], 2001*

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3].

Objekte „Hash-bar“ machen

> *. . . The* `__hash__()` *method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a* `tuple` *and hashing the tuple.*
>
> — *[3], 2001*

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Nur unveränderliche Klassen dürfen `__hash__` implementieren, also nur Klassen, bei denen alle Attribute den Type Hint `Final` haben und ihre Werte nur im Initialisierer `__init__` bekommen.

Objekte „Hash-bar“ machen

> *. . . The* `__hash__()` *method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a* `tuple` *and hashing the tuple.*
>
> — *[3], 2001*

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Nur unveränderliche Klassen dürfen `__hash__` implementieren, also nur Klassen, bei denen alle Attribute den Type Hint `Final` haben und ihre Werte nur im Initialisierer `__init__` bekommen.
- Das Ergebnis von `a.__hash__()` darf sich nie ändern (weil `a` sich auch nie ändern darf).

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Nur unveränderliche Klassen dürfen `__hash__` implementieren, also nur Klassen, bei denen alle Attribute den Type Hint `Final` haben und ihre Werte nur im Initialisierer `__init__` bekommen.
- Das Ergebnis von `a.__hash__()` darf sich nie ändern (weil `a` sich auch nie ändern darf).
- Wenn eine Klasse nicht `__eq__` definiert, dann kann sie auch nicht `__hash__` implementieren.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Nur unveränderliche Klassen dürfen `__hash__` implementieren, also nur Klassen, bei denen alle Attribute den Type Hint `Final` haben und ihre Werte nur im Initialisierer `__init__` bekommen.
- Das Ergebnis von `a.__hash__()` darf sich nie ändern (weil `a` sich auch nie ändern darf).
- Wenn eine Klasse nicht `__eq__` definiert, dann kann sie auch nicht `__hash__` implementieren.
- Instanzen einer Klasse, die `__eq__` implementiert aber nicht `__hash__`, können nicht als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Das Ergebnis von `a.__hash__()` darf sich nie ändern (weil `a` sich auch nie ändern darf).
- Wenn eine Klasse nicht `__eq__` definiert, dann kann sie auch nicht `__hash__` implementieren.
- Instanzen einer Klasse, die `__eq__` implementiert aber nicht `__hash__`, können nicht als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Nur Instanzen einer Klasse, die sowohl `__eq__` als auch `__hash__` implementiert, können als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Wenn eine Klasse nicht `__eq__` definiert, dann kann sie auch nicht `__hash__` implementieren.
- Instanzen einer Klasse, die `__eq__` implementiert aber nicht `__hash__`, können nicht als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Nur Instanzen einer Klasse, die sowohl `__eq__` als auch `__hash__` implementiert, können als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Die Ergebnisse von `__eq__` und `__hash__` müssen auf Basis der selben Attribute berechnet werden.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Instanzen einer Klasse, die `__eq__` implementiert aber nicht `__hash__`, können nicht als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Nur Instanzen einer Klasse, die sowohl `__eq__` als auch `__hash__` implementiert, können als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Die Ergebnisse von `__eq__` und `__hash__` müssen auf Basis der selben Attribute berechnet werden. Mit anderen Worten, die Attribute eines Objekts `a`, die das Ergebnis von `a.__eq__(...)` bestimmen, müssen genau die gleichen sein wie die, die das Ergebnis von `a.__hash__()` bestimmen.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Instanzen einer Klasse, die `__eq__` implementiert aber nicht `__hash__`, können nicht als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Nur Instanzen einer Klasse, die sowohl `__eq__` als auch `__hash__` implementiert, können als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Die Ergebnisse von `__eq__` und `__hash__` müssen auf Basis der selben Attribute berechnet werden. Mit anderen Worten, die Attribute eines Objekts `a`, die das Ergebnis von `a.__eq__(...)` bestimmen, müssen genau die gleichen sein wie die, die das Ergebnis von `a.__hash__()` bestimmen.
- Es ist am Besten, die Ergebnisse von `a.__hash__()` zu berechnen, in dem man einfach alle diese Attribute in ein `tuple` tut und dann dieses Tupel an `hash` übergibt.

Objekte „Hash-bar“ machen

Gute Praxis

Um `__eq__` und `__hash__` zu implementieren, sind die folgenden Regeln zu beachten[3]:

- Nur Instanzen einer Klasse, die sowohl `__eq__` als auch `__hash__` implementiert, können als Schlüssel in Dictionaries oder Elemente von Mengen verwendet werden.
- Die Ergebnisse von `__eq__` und `__hash__` müssen auf Basis der selben Attribute berechnet werden. Mit anderen Worten, die Attribute eines Objekts `a`, die das Ergebnis von `a.__eq__(...)` bestimmen, müssen genau die gleichen sein wie die, die das Ergebnis von `a.__hash__()` bestimmen.
- Es ist am Besten, die Ergebnisse von `a.__hash__()` zu berechnen, in dem man einfach alle diese Attribute in ein `tuple` tut und dann dieses Tupel an `hash` übergibt.
- Zwei Objekte, die gleich sind, müssen den gleichen Hash-Wert haben.

Beispiel: `Point` mit `__hash__`

- Nun können wir endlich unsere Klasse `Point` „hashbar“ machen.

```python
"""A class for points, with equals and hash dunder methods."""

from math import isfinite
from types import NotImplementedType
from typing import Final


class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Nun können wir endlich unsere Klasse `Point` „hashbar“ machen.
- Wir erstellen eine neue Datei `point_with_hash.py`.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Nun können wir endlich unsere Klasse `Point` „hashbar“ machen.
- Wir erstellen eine neue Datei `point_with_hash.py`.
- Wir behalten die vorherige Implementierung von `Point` mit der Methode `__eq__`.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Nun können wir endlich unsere Klasse `Point` „hashbar“ machen.
- Wir erstellen eine neue Datei `point_with_hash.py`.
- Wir behalten die vorherige Implementierung von `Point` mit der Methode `__eq__`.
- Alles, was wir machen müssen, ist eine neue Methode `__hash__` hinzuzufügen.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Nun können wir endlich unsere Klasse `Point` „hashbar“ machen.
- Wir erstellen eine neue Datei `point_with_hash.py`.
- Wir behalten die vorherige Implementierung von `Point` mit der Methode `__eq__`.
- Alles, was wir machen müssen, ist eine neue Methode `__hash__` hinzuzufügen.
- Die einzigen Attribute, die in unserer `__eq__`-Methode eine Rolle spielen, sind die beiden Koordinaten des Punktes, `self.x` und `self.y`.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wir erstellen eine neue Datei `point_with_hash.py`.
- Wir behalten die vorherige Implementierung von `Point` mit der Methode `__eq__`.
- Alles, was wir machen müssen, ist eine neue Methode `__hash__` hinzuzufügen.
- Die einzigen Attribute, die in unserer `__eq__`-Methode eine Rolle spielen, sind die beiden Koordinaten des Punktes, `self.x` und `self.y`.
- Also kann das Ergebnis von `__hash__` einfach `hash((self.x, self.y))` sein.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Alles, was wir machen müssen, ist eine neue Methode `__hash__` hinzuzufügen.
- Die einzigen Attribute, die in unserer `__eq__`-Methode eine Rolle spielen, sind die beiden Koordinaten des Punktes, `self.x` und `self.y`.
- Also kann das Ergebnis von `__hash__` einfach `hash((self.x, self.y))` sein.
- Wir brauchen die inneren Klammern um ein Tupel zu erstellen, also wir machen implizit `t = (self.x, self.y)` und dann `hash(t)`.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Die einzigen Attribute, die in unserer `__eq__`-Methode eine Rolle spielen, sind die beiden Koordinaten des Punktes, `self.x` und `self.y`.
- Also kann das Ergebnis von `__hash__` einfach `hash((self.x, self.y))` sein.
- Wir brauchen die inneren Klammern um ein Tupel zu erstellen, also wir machen implizit `t = (self.x, self.y)` und dann `hash(t)`.
- Wenn wir das so hinschreiben, bekommen wir plötzlich einen Schreck.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Also kann das Ergebnis von `__hash__` einfach `hash((self.x, self.y))` sein.
- Wir brauchen die inneren Klammern um ein Tupel zu erstellen, also wir machen implizit `t = (self.x, self.y)` und dann `hash(t)`.
- Wenn wir das so hinschreiben, bekommen wir plötzlich einen Schreck.
- Wir erlauben ja, dass die Koordinaten der Punkte entweder `int`s oder `float`s seien können.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Also kann das Ergebnis von `__hash__` einfach `hash((self.x, self.y))` sein.
- Wir brauchen die inneren Klammern um ein Tupel zu erstellen, also wir machen implizit `t = (self.x, self.y)` und dann `hash(t)`.
- Wenn wir das so hinschreiben, bekommen wir plötzlich einen Schreck.
- Wir erlauben ja, dass die Koordinaten der Punkte entweder `int`s oder `float`s seien können.
- Nun wissen wir, dass `5.0 == 5` natürlich `True` ergibt.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wir brauchen die inneren Klammern um ein Tupel zu erstellen, also wir machen implizit `t = (self.x, self.y)` und dann `hash(t)`.
- Wenn wir das so hinschreiben, bekommen wir plötzlich einen Schreck.
- Wir erlauben ja, dass die Koordinaten der Punkte entweder `int`s oder `float`s seien können.
- Nun wissen wir, dass `5.0 == 5` natürlich `True` ergibt.
- Deshalb ist ja `Point(5.0, 3).__eq__(Point(5, 3))` auch `True`.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn wir das so hinschreiben, bekommen wir plötzlich einen Schreck.
- Wir erlauben ja, dass die Koordinaten der Punkte entweder `int`s oder `float`s seien können.
- Nun wissen wir, dass `5.0 == 5` natürlich `True` ergibt.
- Deshalb ist ja `Point(5.0, 3).__eq__(Point(5, 3))` auch `True`.
- Ist aber `hash((5.0, 3))` wirklich das gleiche wie `hash((5, 3))`?

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wir erlauben ja, dass die Koordinaten der Punkte entweder `ints` oder `floats` seien können.
- Nun wissen wir, dass `5.0 == 5` natürlich `True` ergibt.
- Deshalb ist ja `Point(5.0, 3).__eq__(Point(5, 3))` auch `True`.
- Ist aber `hash((5.0, 3))` wirklich das gleiche wie `hash((5, 3))`?
- Oder, noch einfacher, ist `hash(5.0)== hash(5)` wirklich wahr?

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Nun wissen wir, dass `5.0 == 5` natürlich `True` ergibt.
- Deshalb ist ja `Point(5.0, 3).__eq__(Point(5, 3))` auch `True`.
- Ist aber `hash((5.0, 3))` wirklich das gleiche wie `hash((5, 3))`?
- Oder, noch einfacher, ist `hash(5.0)== hash(5)` wirklich wahr?
- Wenn nicht, dann hätten wir den Vertrag von `__hash__` und `__eq__` auf eine sehr eigenartige und unerwartete Weise gebrochen.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Deshalb ist ja `Point(5.0, 3).__eq__(Point(5, 3))` auch `True`.
- Ist aber `hash((5.0, 3))` wirklich das gleiche wie `hash((5, 3))`?
- Oder, noch einfacher, ist `hash(5.0)== hash(5)` wirklich wahr?
- Wenn nicht, dann hätten wir den Vertrag von `__hash__` und `__eq__` auf eine sehr eigenartige und unerwartete Weise gebrochen.
- `Point(5.0, 3)` wäre gleich zu `Point(5, 3)`, aber ihre Hash-Werte wären verschieden.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Oder, noch einfacher, ist `hash(5.0)== hash(5)` wirklich wahr?
- Wenn nicht, dann hätten wir den Vertrag von `__hash__` und `__eq__` auf eine sehr eigenartige und unerwartete Weise gebrochen.
- `Point(5.0, 3)` wäre gleich zu `Point(5, 3)`, aber ihre Hash-Werte wären verschieden.
- Wenn wir eine Menge `s` von Punkten hätten und `Point(5.0, 3)` in dieser Menge speichern, dann könnte das Ergebnis von `Point(5, 3)in s` entweder `True` oder `False` sein.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn nicht, dann hätten wir den Vertrag von `__hash__` und `__eq__` auf eine sehr eigenartige und unerwartete Weise gebrochen.
- `Point(5.0, 3)` wäre gleich zu `Point(5, 3)`, aber ihre Hash-Werte wären verschieden.
- Wenn wir eine Menge `s` von Punkten hätten und `Point(5.0, 3)` in dieser Menge speichern, dann könnte das Ergebnis von `Point(5, 3)in s` entweder `True` oder `False` sein.
- Wenn das Layout der internen Tabelle, deren Größe von allen vorherigen Einfügungen und Löschungen abhängt, so ist, dass die verschiedenen Hash-Werte auf den

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- `Point(5.0, 3)` wäre gleich zu `Point(5, 3)`, aber ihre Hash-Werte wären verschieden.
- Wenn wir eine Menge `s` von Punkten hätten und `Point(5.0, 3)` in dieser Menge speichern, dann könnte das Ergebnis von `Point(5, 3)in s` entweder `True` oder `False` sein.
- Wenn das Layout der internen Tabelle, deren Größe von allen vorherigen Einfügungen und Löschungen abhängt, so ist, dass die verschiedenen Hash-Werte auf den selben Index gemappt werden, dann ist das Ergebnis `True`.
- Im viel Wahrscheinlicheren Fall dass das nicht zutrifft, ist das Ergebnis

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn wir eine Menge `s` von Punkten hätten und `Point(5.0, 3)` in dieser Menge speichern, dann könnte das Ergebnis von `Point(5, 3)in s` entweder `True` oder `False` sein.
- Wenn das Layout der internen Tabelle, deren Größe von allen vorherigen Einfügungen und Löschungen abhängt, so ist, dass die verschiedenen Hash-Werte auf den selben Index gemappt werden, dann ist das Ergebnis `True`.
- Im viel Wahrscheinlicheren Fall dass das nicht zutrifft, ist das Ergebnis `False`.
- Wenn `hash(5.0)!= hash(5)` wirklich wahr wäre, dann würde

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn das Layout der internen Tabelle, deren Größe von allen vorherigen Einfügungen und Löschungen abhängt, so ist, dass die verschiedenen Hash-Werte auf den selben Index gemappt werden, dann ist das Ergebnis `True`.
- Im viel Wahrscheinlicheren Fall dass das nicht zutrifft, ist das Ergebnis `False`.
- Wenn `hash(5.0)!= hash(5)` wirklich wahr wäre, dann würde unsere Implementierung Programme erzeugen, die sich unerwartet anders verhalten, in seltenen Situationen, die wir wahrscheinlich nicht deterministisch reproduzieren könnten.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn das Layout der internen Tabelle, deren Größe von allen vorherigen Einfügungen und Löschungen abhängt, so ist, dass die verschiedenen Hash-Werte auf den selben Index gemappt werden, dann ist das Ergebnis `True`.
- Im viel Wahrscheinlicheren Fall dass das nicht zutrifft, ist das Ergebnis `False`.
- Wenn `hash(5.0)!= hash(5)` wirklich wahr wäre, dann würde unsere Implementierung Programme erzeugen, die sich unerwartet anders verhalten, in seltenen Situationen, die wir wahrscheinlich nicht deterministisch reproduzieren könnten.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Wenn `hash(5.0) != hash(5)` wirklich wahr wäre, dann würde unsere Implementierung Programme erzeugen, die sich unerwartet anders verhalten, in seltenen Situationen, die wir wahrscheinlich nicht deterministisch reproduzieren könnten.
- Das ist eine der schrecklichsten Arten von Bugs.
- Gleichzeitig könnten wir dann Mengen erzeugen, die das gleiche Element mehrfach beinhalten können.
- Das würde dann die Definition von Mengen verletzen.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Das ist eine der schrecklichsten Arten von Bugs.
- Gleichzeitig könnten wir dann Mengen erzeugen, die das gleiche Element mehrfach beinhalten können.
- Das würde dann die Definition von Mengen verletzen.
- Kein Wunder, das Python mit den Regeln für `__hash__` und `__eq__` recht streng ist...

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Das ist eine der schrecklichsten Arten von Bugs.
- Gleichzeitig könnten wir dann Mengen erzeugen, die das gleiche Element mehrfach beinhalten können.
- Das würde dann die Definition von Mengen verletzen.
- Kein Wunder, das Python mit den Regeln für `__hash__` und `__eq__` recht streng ist...
- Zum Glück haben die Entwickler von Python an dieses Problem gedacht.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__`

- Gleichzeitig könnten wir dann Mengen erzeugen, die das gleiche Element mehrfach beinhalten können.
- Das würde dann die Definition von Mengen verletzen.
- Kein Wunder, das Python mit den Regeln für `__hash__` und `__eq__` recht streng ist...
- Zum Glück haben die Entwickler von Python an dieses Problem gedacht.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """

        return hash((self.x, self.y))  # hash over the tuple of values
```

> *Numeric values that compare equal have the same* ***`hash`*** *value (even if they are of different types, as is the case for* `1` *and* `1.0`*).*
>
> — *[6], 2001*

Beispiel: `Point` mit `__hash__`

- Gleichzeitig könnten wir dann Mengen erzeugen, die das gleiche Element mehrfach beinhalten können.
- Das würde dann die Definition von Mengen verletzen.
- Kein Wunder, das Python mit den Regeln für `__hash__` und `__eq__` recht streng ist...
- Zum Glück haben die Entwickler von Python an dieses Problem gedacht.
- Daher können wir also wirklich unsere `__hash__`-Funktion genauso implementieren, wie wir es uns überlegt haben.

```python
class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir benutzen unsere neue Klasse im Programm `point_with_hash_user.py`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")   # True
print(f"{p2 in points = }")   # True
print(f"{p3 in points = }")   # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir benutzen unsere neue Klasse im Programm `point_with_hash_user.py`.
- Wir erstellen zuerst drei Punkte `p1 = Point(3, 5)`, `p2 = Point(7, 8)` und `p3 = Point(3, 5.0)`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir benutzen unsere neue Klasse im Programm `point_with_hash_user.py`.
- Wir erstellen zuerst drei Punkte `p1 = Point(3, 5)`, `p2 = Point(7, 8)` und `p3 = Point(3, 5.0)`.
- `p1 == p2` ist `False`, aber `p1 == p3` ist `True`, obwohl die `y`-Koordinate von `p1` ein `int` ist und die von `p2` ein `float` (aber mit dem selben Wert).

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir benutzen unsere neue Klasse im Programm `point_with_hash_user.py`.
- Wir erstellen zuerst drei Punkte `p1 = Point(3, 5)`, `p2 = Point(7, 8)` und `p3 = Point(3, 5.0)`.
- `p1 == p2` ist `False`, aber `p1 == p3` ist `True`, obwohl die `y`-Koordinate von `p1` ein `int` ist und die von `p2` ein `float` (aber mit dem selben Wert).
- Dann erstellen wir die Menge `points` als `{p1, p2, p3}`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir benutzen unsere neue Klasse im Programm `point_with_hash_user.py`.
- Wir erstellen zuerst drei Punkte `p1 = Point(3, 5)`, `p2 = Point(7, 8)` und `p3 = Point(3, 5.0)`.
- `p1 == p2` ist `False`, aber `p1 == p3` ist `True`, obwohl die `y`-Koordinate von `p1` ein `int` ist und die von `p2` ein `float` (aber mit dem selben Wert).
- Dann erstellen wir die Menge `points` als `{p1, p2, p3}`.
- Sie hat Größe 2.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir erstellen zuerst drei Punkte `p1 = Point(3, 5)`, `p2 = Point(7, 8)` und `p3 = Point(3, 5.0)`.
- `p1 == p2` ist `False`, aber `p1 == p3` ist `True`, obwohl die `y`-Koordinate von `p1` ein `int` ist und die von `p2` ein `float` (aber mit dem selben Wert).
- Dann erstellen wir die Menge `points` als `{p1, p2, p3}`.
- Sie hat Größe 2.
- Weil `p1 == p3` wird nur eines dieser beiden Objekte in der Menge gespeicher.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- `p1 == p2` ist `False`, aber `p1 == p3` ist `True`, obwohl die `y`-Koordinate von `p1` ein `int` ist und die von `p2` ein `float` (aber mit dem selben Wert).
- Dann erstellen wir die Menge `points` als `{p1, p2, p3}`.
- Sie hat Größe 2.
- Weil `p1 == p3` wird nur eines dieser beiden Objekte in der Menge gespeicher.
- `p1 in points`, `p2 in points` und `p3 in points` sind aber alle `True`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Dann erstellen wir die Menge `points` als `{p1, p2, p3}`.
- Sie hat Größe 2.
- Weil `p1 == p3` wird nur eines dieser beiden Objekte in der Menge gespeicher.
- `p1 in points`, `p2 in points` und `p3 in points` sind aber alle `True`.
- Das ist weil `p1` und `p3` gleich sind und auch den selben Hash-Wert haben.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Sie hat Größe 2.
- Weil `p1 == p3` wird nur eines dieser beiden Objekte in der Menge gespeicher.
- `p1 in points`, `p2 in points` und `p3 in points` sind aber alle `True`.
- Das ist weil `p1` und `p3` gleich sind und auch den selben Hash-Wert haben.
- Wenn einen nuen Punkt `p4` mit den gleichen Koordinaten wie `p2` erstellen wirden, dann würde `p4 in points` auch gelten.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Weil `p1 == p3` wird nur eines dieser beiden Objekte in der Menge gespeicher.
- `p1 in points`, `p2 in points` und `p3 in points` sind aber alle `True`.
- Das ist weil `p1` und `p3` gleich sind und auch den selben Hash-Wert haben.
- Wenn einen nuen Punkt `p4` mit den gleichen Koordinaten wie `p2` erstellen wirden, dann würde `p4 in points` auch gelten.
- Ein Punkt `p5` mit Koordinaten, die anders als die von `p1` und `p2` sind, wäre kein Element von `points`, also `p5 in points` wäre dann `False`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- `p1 in points`, `p2 in points` und `p3 in points` sind aber alle `True`.
- Das ist weil `p1` und `p3` gleich sind und auch den selben Hash-Wert haben.
- Wenn einen nuen Punkt `p4` mit den gleichen Koordinaten wie `p2` erstellen wirden, dann würde `p4 in points` auch gelten.
- Ein Punkt `p5` mit Koordinaten, die anders als die von `p1` und `p2` sind, wäre kein Element von `points`, also `p5 in points` wäre dann `False`.
- Wir können nun Instanzen der Klasse `Point` auch als Schlüssel für ein Dictionary `point_vals` verwenden.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")     # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Das ist weil `p1` und `p3` gleich sind und auch den selben Hash-Wert haben.
- Wenn einen nuen Punkt `p4` mit den gleichen Koordinaten wie `p2` erstellen wirden, dann würde `p4 in points` auch gelten.
- Ein Punkt `p5` mit Koordinaten, die anders als die von `p1` und `p2` sind, wäre kein Element von `points`, also `p5 in points` wäre dann `False`.
- Wir können nun Instanzen der Klasse `Point` auch als Schlüssel für ein Dictionary `point_vals` verwenden.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wenn einen nuen Punkt `p4` mit den gleichen Koordinaten wie `p2` erstellen wirden, dann würde `p4 in points` auch gelten.
- Ein Punkt `p5` mit Koordinaten, die anders als die von `p1` und `p2` sind, wäre kein Element von `points`, also `p5 in points` wäre dann `False`.
- Wir können nun Instanzen der Klasse `Point` auch als Schlüssel für ein Dictionary `point_vals` verwenden.
- Die selben Dictionary-Operationen wie in Einheit 21 können problemlos verwendet werden.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Ein Punkt `p5` mit Koordinaten, die anders als die von `p1` und `p2` sind, wäre kein Element von `points`, also `p5 in points` wäre dann `False`.
- Wir können nun Instanzen der Klasse `Point` auch als Schlüssel für ein Dictionary `point_vals` verwenden.
- Die selben Dictionary-Operationen wie in Einheit 21 können problemlos verwendet werden.
- Wir assoziieren Wert `"A"` mit Schlüssel `p1` und `"B"` mit Schlüssel `p2`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir können nun Instanzen der Klasse `Point` auch als Schlüssel für ein Dictionary `point_vals` verwenden.
- Die selben Dictionary-Operationen wie in Einheit 21 können problemlos verwendet werden.
- Wir assoziieren Wert `"A"` mit Schlüssel `p1` und `"B"` mit Schlüssel `p2`.
- Dann fügen wir ein weiteres Schlüssel-Wert-Paar hinzu, in dem wir `"C"` unter Schlüssel `Point(7, 9)` speichern.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Die selben Dictionary-Operationen wie in Einheit 21 können problemlos verwendet werden.
- Wir assoziieren Wert `"A"` mit Schlüssel `p1` und `"B"` mit Schlüssel `p2`.
- Dann fügen wir ein weiteres Schlüssel-Wert-Paar hinzu, in dem wir `"C"` unter Schlüssel `Point(7, 9)` speichern.
- Dieses neue Paar taucht im Dictionary wie erwartet auf.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

- Wir assoziieren Wert `"A"` mit Schlüssel `p1` und `"B"` mit Schlüssel `p2`.
- Dann fügen wir ein weiteres Schlüssel-Wert-Paar hinzu, in dem wir `"C"` unter Schlüssel `Point(7, 9)` speichern.
- Dieses neue Paar taucht im Dictionary wie erwartet auf.
- Wenn wir den Wert `"D"` unter Schlüssel `Point(3.0, 5.0)` speichern, dann überschreibt das den Wert `"A"`, der unter Schlüssel `p1` gespeichert ist, weil `p1` nämlich `Point(3, 5)` ist.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Beispiel: `Point` mit `__hash__` benutzen

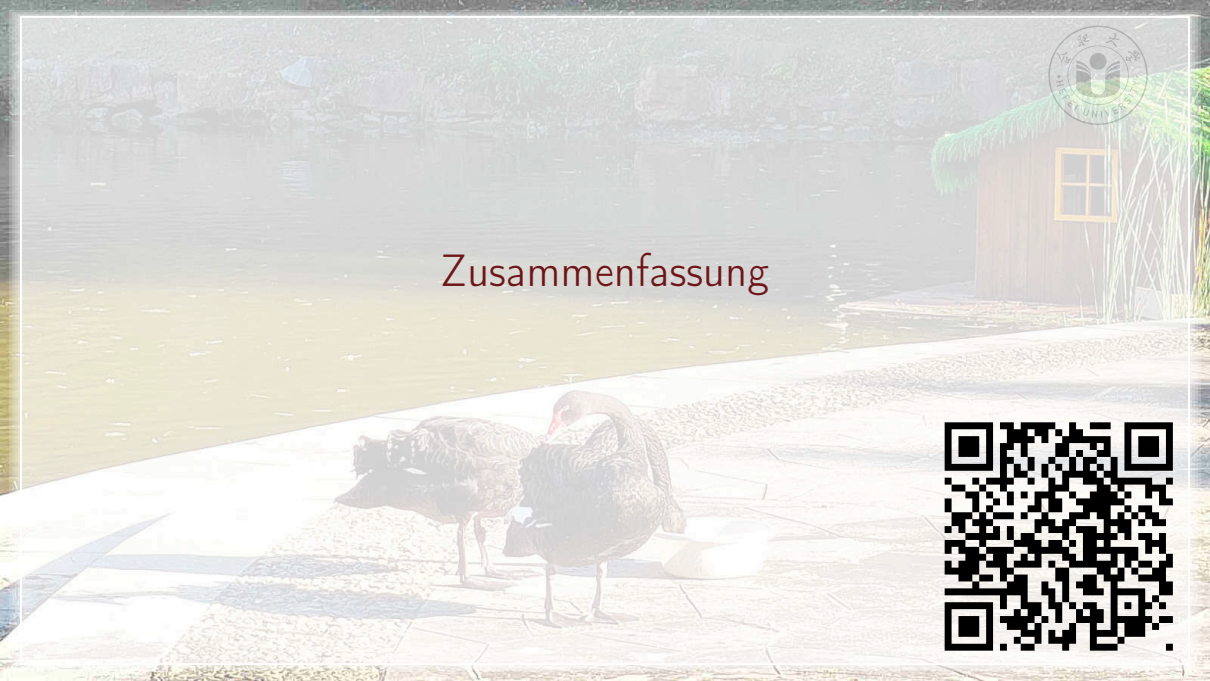
- Dann fügen wir ein weiteres Schlüssel-Wert-Paar hinzu, in dem wir `"C"` unter Schlüssel `Point(7, 9)` speichern.
- Dieses neue Paar taucht im Dictionary wie erwartet auf.
- Wenn wir den Wert `"D"` unter Schlüssel `Point(3.0, 5.0)` speichern, dann überschreibt das den Wert `"A"`, der unter Schlüssel `p1` gespeichert ist, weil `p1` nämlich `Point(3, 5)` ist.
- Wenn wir dann `point_vals[p1]` abfragen, kommt `"D"`.

```python
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)    # Create a first point.
p2: Point = Point(7, 8)    # Create a second, different point.
p3: Point = Point(3, 5.0)  # A third point, which equals the first.

print(f"{(p1 == p2) = }")  # False, since p1 is really != p2
print(f"{(p1 == p3) = }")  # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"  # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

Zusammenfassung



Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.
- Andernfalls können wir sehr schwer verständliche Fehler erzeugen.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.
- Andernfalls können wir sehr schwer verständliche Fehler erzeugen.
- Wir könnten z. B. `__eq__` und `__ne__` aus Versehen so implementieren, dass `x == y` nicht mehr das Gegenteil von `x != y` ist.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.
- Andernfalls können wir sehr schwer verständliche Fehler erzeugen.
- Wir könnten z. B. `__eq__` und `__ne__` aus Versehen so implementieren, dass `x == y` nicht mehr das Gegenteil von `x != y` ist.
- Oder wir könnten `__eq__` und `__hash__` so implementieren, dass Mengen keine Mengen mehr sind...

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.
- Andernfalls können wir sehr schwer verständliche Fehler erzeugen.
- Wir könnten z. B. `__eq__` und `__ne__` aus Versehen so implementieren, dass `x == y` nicht mehr das Gegenteil von `x != y` ist.
- Oder wir könnten `__eq__` und `__hash__` so implementieren, dass Mengen keine Mengen mehr sind...
- Solcherlei Bugs wären schwer zu finden.

Zusammenfassung

- Objekte „hashbar“ zu machen ist ein Thema, welches wir nicht soooo oft antreffen werden.
- Aber es ist ein Thema, von dem wir lernen können.
- Wir haben gelernt, dass die Arbeit mit Dunder-Methoden Vorsicht erfordert.
- Wir müssen wirklich verstehen, was wir tun.
- Andernfalls können wir sehr schwer verständliche Fehler erzeugen.
- Wir könnten z. B. `__eq__` und `__ne__` aus Versehen so implementieren, dass `x == y` nicht mehr das Gegenteil von `x != y` ist.
- Oder wir könnten `__eq__` und `__hash__` so implementieren, dass Mengen keine Mengen mehr sind...
- Solcherlei Bugs wären schwer zu finden.
- Wir müssen die Dokumentation gründlich lesen.

谢谢您们！

Thank you!

Vielen Dank!

References I

[1] AndrewBadr. “TimeComplexity”. In: *The Python Wiki*. Beaverton, OR, USA: Python Software Foundation (PSF), 19. Jan. 2023. URL: https://wiki.python.org/moin/TimeComplexity (besucht am 2024-08-27) (siehe S. 22–41).

[2] Paul Gustav Heinrich Bachmann. *Die Analytische Zahlentheorie / Dargestellt von Paul Bachmann*. Bd. Zweiter Theil der Reihe Zahlentheorie: Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Haupttheilen. Leipzig, Sachsen, Germany: B. G. Teubner, 1894. ISBN: 978-1-4181-6963-3. URL: http://gallica.bnf.fr/ark:/12148/bpt6k994750 (besucht am 2023-12-13) (siehe S. 143).

[3] “Basic Customizations: `object.__hash__(self)`”. In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__hash__ (besucht am 2024-12-09) (siehe S. 18–21, 72–82).

[4] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 142).

[5] “Built-in `Exception`S”. In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/exceptions.html (besucht am 2024-10-29) (siehe S. 18–21).

[6] “Built-in Functions”. In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html (besucht am 2024-12-09) (siehe S. 107).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald Linn Rivest und Clifford Stein. *Introduction to Algorithms*. 3. Aufl. Cambridge, MA, USA: MIT Press, 2009. ISBN: 978-0-262-03384-8 (siehe S. 22–40).

[8] 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. 142).

[9] Adam Gold. *Python Hash Tables Under the Hood*. San Francisco, CA, USA: GitHub Inc, 30. Juni 2020. URL: https://adamgold.github.io/posts/python-hash-tables-under-the-hood (besucht am 2024-12-09) (siehe S. 22–64).

[10] Trey Hunner. “Python Big $\mathcal{O}$: The Time Complexities of Different Data Structures in Python; Python 3.8-3.12”. In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, 16. Apr. 2024. URL: https://www.pythonmorsels.com/time-complexities (besucht am 2024-08-27) (siehe S. 22–41).

References II

[11] John Hunt. *A Beginners Guide to Python 3 Programming*. 2. Aufl. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (siehe S. 142).

[12] Donald Ervin Knuth. "Big Omicron and Big Omega and Big Theta". *ACM SIGACT News* 8(2):18–24, Apr.–Juni 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5700. doi:10.1145/1008328.1008329 (siehe S. 143).

[13] Donald Ervin Knuth. *Fundamental Algorithms*. 3. Aufl. Bd. 1 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (siehe S. 143).

[14] Donald Ervin Knuth. *Sorting and Searching*. Bd. 3 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1998. ISBN: 978-0-201-89685-5 (siehe S. 22–40).

[15] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig, Sachsen, Germany: B. G. Teubner, 1909. ISBN: 978-0-8218-2650-8 (siehe S. 143).

[16] 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. 143).

[17] Kent D. Lee und Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (siehe S. 142).

[18] 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. 142).

[19] 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. 142).

[20] Marc Loy, Patrick Niemeyer und Daniel Leuck. *Learning Java*. 5. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2020. ISBN: 978-1-4920-5627-0 (siehe S. 142).

[21] Laurent Luce. *Python Dictionary Implementation*. Belmont, MA, USA, 29. Aug. 2011–30. Mai 2020. URL: https://www.laurentluce.com/posts/python-dictionary-implementation (besucht am 2024-12-09) (siehe S. 22–64).

References III

[22] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 142).

[23] nishkarsh146. *Complexity Cheat Sheet for Python Operations*. Noida, Uttar Pradesh, India: GeeksforGeeks – Sanchhaya Education Private Limited, 17. Aug. 2022. URL: https://www.geeksforgeeks.org/complexity-cheat-sheet-for-python-operations (besucht am 2024-08-27) (siehe S. 22–41).

[24] 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. 142).

[25] *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. 142).

[26] Abraham „Avi" Silberschatz, Henry F. „Hank" Korth und S. Sudarshan. *Database System Concepts*. 7. Aufl. New York, NY, USA: McGraw-Hill, März 2019. ISBN: 978-0-07-802215-9 (siehe S. 22–40).

[27] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 142).

[28] *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library (besucht am 2025-04-27).

[29] ."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. 142).

[30] 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. 142, 143).

[31] 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. 143).

References IV

[32] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: https://thomasweise.github.io/programmingWithPython (besucht am 2025-01-05) (siehe S. 142).

Glossary (in English) I

C is a programming language, which is very successful in system programming situations[8,25].

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[27,30]. Learn more at `https://git-scm.com`.

GitHub is a website where software projects can be hosted and managed via the Git VCS[24,30]. Learn more at `https://github.com`.

Java is another very successful programming language, with roots in the C family of languages[4,20].

literal A literal is a specific concrete value, something that is written down as-is[18,29]. 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`.

modulo division is, in Python, done by the operator `%` that computes the remainder of a division. `15 % 6` gives us `3`.

Mypy is a static type checking tool for Python[19] that makes use of type hints. Learn more at `https://github.com/python/mypy` and in[32].

Python The Python programming language[11,17,22,32], i.e., what you will learn about in our book[32]. Learn more at `https://python.org`.

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[16,31]. 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[30]. 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.

$\Omega(g(x))$ If $f(x) = \Omega(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $f(x) \geq c * g(x) \geq 0 \forall x \geq x_0$[12,13]. In other words, $\Omega(g(x))$ describes a lower bound for function growth.

$\mathcal{O}(g(x))$ If $f(x) = \mathcal{O}(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $0 \leq f(x) \leq c * g(x) \forall x \geq x_0$[2,12,13,15]. In other words, $\mathcal{O}(g(x))$ describes an upper bound for function growth.

$\Theta(g(x))$ If $f(x) = \Theta(g(x))$, then $f(x) = \mathcal{O}(g(x))$ and $f(x) = \Omega(g(x))$[12,13]. In other words, $\Theta(g(x))$ describes an exact order of function growth.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{R}^+$ the set of the positive real numbers, i.e., $\mathbb{R}^+ = \{x \in \mathbb{R} : x > 0\}$.