



合肥大學
HEFEI UNIVERSITY



Programming with Python

41. Klassen: Grundlagen

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

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-Code unten rechts). Dort können Sie das Kursbuch (in Englisch) und diese Slides finden. Das Repository mit den Beispielprogrammen in Python finden Sie unter <https://github.com/thomasWeise/programmingWithPythonCode>.



Outline



1. Einleitung
2. Gründe für Klassen
3. Klassen Definieren
4. Beispiel
5. Veränderbar vs. Unveränderbar
6. Zusammenfassung





Einleitung



Einleitung

- Wir haben bereits viele einfache Datentypen kennengelernt.



Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.

Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.
- In vielen Situationen haben wir es jedoch mit Daten zu tun, von keinem der obigen Strukturen vernünftig repräsentiert werden können.

Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.
- In vielen Situationen haben wir es jedoch mit Daten zu tun, von keinem der obigen Strukturen vernünftig repräsentiert werden können.
- Viele Datentypen sind im Grunde Strukturen, die mehrere Elemente, die miteinander in einer semantischen Beziehung stehen, verbinden.

Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.
- In vielen Situationen haben wir es jedoch mit Daten zu tun, von keinem der obigen Strukturen vernünftig repräsentiert werden können.
- Viele Datentypen sind im Grunde Strukturen, die mehrere Elemente, die miteinander in einer semantischen Beziehung stehen, verbinden.
- Die Elemente von Listen oder Tupeln, z. B., stehen nur in so fern mit einander in einer Beziehung, dass sie in der selben Kollektion auftauchen.

Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.
- In vielen Situationen haben wir es jedoch mit Daten zu tun, von keinem der obigen Strukturen vernünftig repräsentiert werden können.
- Viele Datentypen sind im Grunde Strukturen, die mehrere Elemente, die miteinander in einer semantischen Beziehung stehen, verbinden.
- Die Elemente von Listen oder Tupeln, z. B., stehen nur in so fern mit einander in einer Beziehung, dass sie in der selben Kollektion auftauchen.
- Die Elemente *Tag*, *Monat*, und *Jahr* eines *Datums* haben dagegen eine viel engere Beziehung mit einer klaren Bedeutung.

Einleitung



- Wir haben bereits viele einfache Datentypen kennengelernt.
- Darüber hinaus haben wir auch verschiedene Arten von Kollektionen gelernt, die mehrere Elemente enthalten können.
- In vielen Situationen haben wir es jedoch mit Daten zu tun, von keinem der obigen Strukturen vernünftig repräsentiert werden können.
- Viele Datentypen sind im Grunde Strukturen, die mehrere Elemente, die miteinander in einer semantischen Beziehung stehen, verbinden.
- Die Elemente von Listen oder Tupeln, z. B., stehen nur in so fern mit einander in einer Beziehung, dass sie in der selben Kollektion auftauchen.
- Die Elemente *Tag*, *Monat*, und *Jahr* eines *Datums* haben dagegen eine viel engere Beziehung mit einer klaren Bedeutung.
- Oftmals formen solche Datentypen und die Operationen auf ihnen eine semantische Einheit.



Gründe für Klassen



Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.
- Alles was wir direkt sehen würden ist, dass sie mit Tupeln von zwei `floats` arbeitet.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.
- Alles was wir direkt sehen würden ist, dass sie mit Tupeln von zwei `floats` arbeitet.
- Auf der anderen Seite haben die beiden Teile einer komplexen Zahl, der Realteil und der Imaginärteil, zwei verschiedene und wohldefinierte Bedeutungen.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Stellen Sie sich vor, wir würden die komplexen Zahlen in Python implementieren wollen.
- (Python hat schon den Datentyp `complex` genau dafür, aber stellen Sie sich vor, es hätte ihn nicht.)
- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.
- Alles was wir direkt sehen würden ist, dass sie mit Tupeln von zwei `floats` arbeitet.
- Auf der anderen Seite haben die beiden Teile einer komplexen Zahl, der Realteil und der Imaginärteil, zwei verschiedene und wohldefinierte Bedeutungen.
- Es wäre aber nicht sofort klar, ob die erste Zahl im Tupel der Realteil oder der Imaginärteil ist.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Nun könnten Sie hingehen und eine komplexe Zahl einfach als `tuple[float, float]` darstellen.
- Das hat aber mehrere Nachteile.
- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.
- Alles was wir direkt sehen würden ist, dass sie mit Tupeln von zwei `floats` arbeitet.
- Auf der anderen Seite haben die beiden Teile einer komplexen Zahl, der Realteil und der Imaginärteil, zwei verschiedene und wohldefinierte Bedeutungen.
- Es wäre aber nicht sofort klar, ob die erste Zahl im Tupel der Realteil oder der Imaginärteil ist.
- Genaugenommen könnten wir komplexe Zahlen auch in Polarform darstellen, wobei dann die Teile des Tupels wieder andere Bedeutungen hätten.

Probleme mit Datenstrukturen am Beispiel Komplexe Zahlen



- Auf der einen Seite könnte dann *jedes* Tupel von zwei `floats` als komplexe Zahl interpretiert werden.
- Von der Signatur einer Funktion, also basierend auf ihren Parameter- und Rückgabe-Datentyp, wäre es dann also nicht klar, ob diese Funktion mit komplexen Zahlen arbeitet oder nicht.
- Alles was wir direkt sehen würden ist, dass sie mit Tupeln von zwei `floats` arbeitet.
- Auf der anderen Seite haben die beiden Teile einer komplexen Zahl, der Realteil und der Imaginärteil, zwei verschiedene und wohldefinierte Bedeutungen.
- Es wäre aber nicht sofort klar, ob die erste Zahl im Tupel der Realteil oder der Imaginärteil ist.
- Genaugenommen könnten wir komplexe Zahlen auch in Polarform darstellen, wobei dann die Teile des Tupels wieder andere Bedeutungen hätten.
- Ebenso wäre die normale textuelle Repräsentation eines Tuples von zwei `floats` so etwas wie `"(3.0, 4.0)"`, wobei wir für komplexe Zahlen eher so etwas wie `"3+4i"` haben wollten.

Use Case: Group Data and Operations



- Der erste wichtige Use Case für Klassen (`classes`) in Python ist, dass sie uns eine Möglichkeit bieten, eine Datenstruktur zusammen mit den Operationen für die Datenstruktur zu definieren¹⁹.

Use Case: Group Data and Operations



- Der erste wichtige Use Case für Klassen (`classes`) in Python ist, dass sie uns eine Möglichkeit bieten, eine Datenstruktur zusammen mit den Operationen für die Datenstruktur zu definieren¹⁹.
- Das erlaubt es uns z. B. eine `class` für komplexe Zahlen zu definieren, die die Attribute `real_part` und `imaginary_part` hat.

Use Case: Group Data and Operations



- Der erste wichtige Use Case für Klassen (`classes`) in Python ist, dass sie uns eine Möglichkeit bieten, eine Datenstruktur zusammen mit den Operationen für die Datenstruktur zu definieren¹⁹.
- Das erlaubt es uns z. B. eine `class` für komplexe Zahlen zu definieren, die die Attribute `real_part` und `imaginary_part` hat.
- Wir können Operatoren wie Addition und Subtraktion definieren, die mit dieser Klasse arbeiten, wodurch sofort klar wird, wie diese zu benutzen sind.

Use Case: Group Data and Operations



- Der erste wichtige Use Case für Klassen (`classes`) in Python ist, dass sie uns eine Möglichkeit bieten, eine Datenstruktur zusammen mit den Operationen für die Datenstruktur zu definieren¹⁹.
- Das erlaubt es uns z. B. eine `class` für komplexe Zahlen zu definieren, die die Attribute `real_part` und `imaginary_part` hat.
- Wir können Operatoren wie Addition und Subtraktion definieren, die mit dieser Klasse arbeiten, wodurch sofort klar wird, wie diese zu benutzen sind.
- Und die Klasse kann eine von uns gewählte textuelle Repräsentation haben.

Probleme mit APIs am Beispiel von Dokumenten



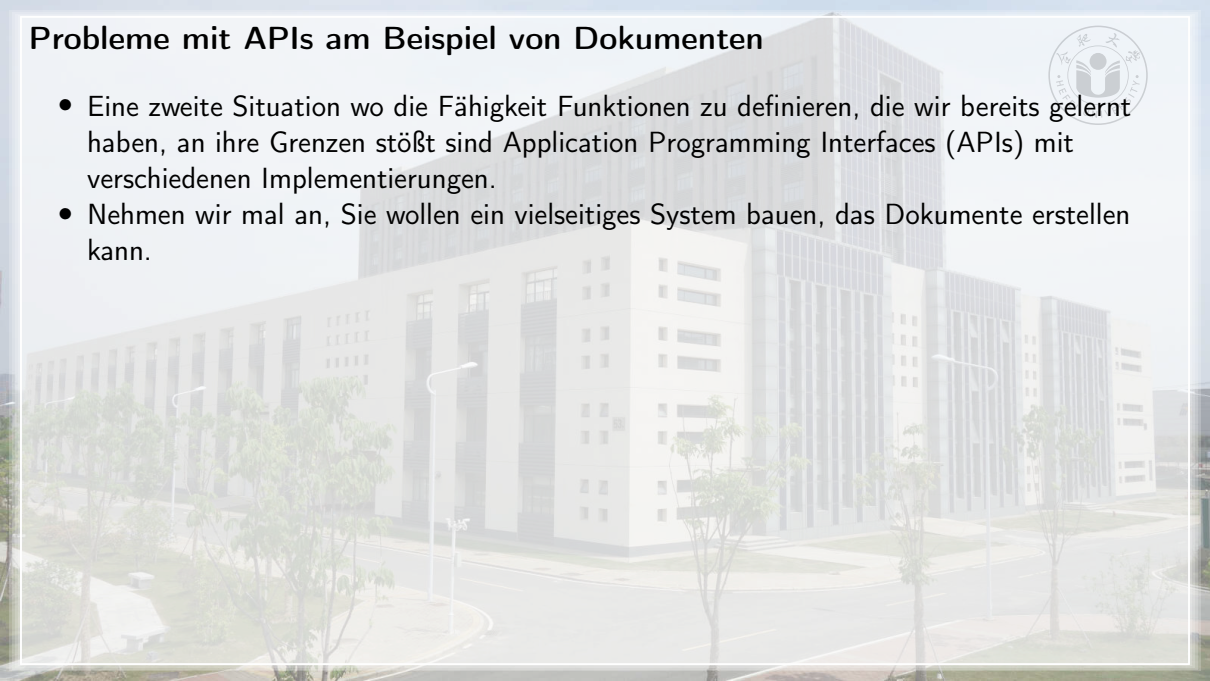
- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.



Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.



Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.

Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.

Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.

Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.

Probleme mit APIs am Beispiel von Dokumenten



- Eine zweite Situation wo die Fähigkeit Funktionen zu definieren, die wir bereits gelernt haben, an ihre Grenzen stößt sind Application Programming Interfaces (APIs) mit verschiedenen Implementierungen.
- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.

Probleme mit APIs am Beispiel von Dokumenten



- Nehmen wir mal an, Sie wollen ein vielseitiges System bauen, das Dokumente erstellen kann.
- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.

Probleme mit APIs am Beispiel von Dokumenten



- Auf der Ausgabeseite wollen Sie verschiedene Formate unterstützen, z. B. LibreOffice^{35,58}, Microsoft Word^{29,69}, und Adobe PDF^{33,110}.
- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.

Probleme mit APIs am Beispiel von Dokumenten



- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.

Probleme mit APIs am Beispiel von Dokumenten



- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.
- Das wäre jedoch ein gräßlicher Ansatz.

Probleme mit APIs am Beispiel von Dokumenten



- Auf der Eingabeseite wollen Sie dem Benutzer/Programmierer eine einheitliche Art zum Dokumente erstellen bereitstellen.
- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.
- Das wäre jedoch ein gräßlicher Ansatz.
- Das größte Problem wäre, dass es keine Möglich gibt, zu definieren, „wie die API aussieht.“

Probleme mit APIs am Beispiel von Dokumenten



- Die API dafür sollte natürlich für alle Ausgabeformate gleich sein.
- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.
- Das wäre jedoch ein gräßlicher Ansatz.
- Das größte Problem wäre, dass es keine Möglich gibt, zu definieren, „wie die API aussieht.“
- Das kann schnell zu Inkonsistenzen im Softwarelebenszyklus führen.

Probleme mit APIs am Beispiel von Dokumenten



- Sie würde nicht aus einer einzelnen Funktion bestehen, sondern aus mehreren Gruppen von Funktionen.
- Es könnte sogar verschachtelte Hierarchien von Funktionen geben, die z. B. das Erstellen von Kapiteln und Absätzen von Text erlauben, oder das Formattieren von Strings mit verschiedenen Fonts.
- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.
- Das wäre jedoch ein gräßlicher Ansatz.
- Das größte Problem wäre, dass es keine Möglichkeit gibt, zu definieren, „wie die API aussieht.“
- Das kann schnell zu Inkonsistenzen im Softwarelebenszyklus führen.
- Wenn wir die Signatur von einer Funktion geringfügig verändern, müssten wir das manuell in alle anderen Module ebenfalls einpflegen.

Probleme mit APIs am Beispiel von Dokumenten



- Natürlich müssten wir diese Operationen verschieden für die verschiedenen Ausgabeformate implementieren.
- Wir könnten das versuchen, in dem wir in verschiedene Module für verschiedene Ausgabeformate implementieren.
- In den Modulen könnten wir dann Funktionen mit dem selben Namen und der selben Signatur implementieren, die das jeweils benötigte Verhalten implementieren.
- Das wäre jedoch ein gräßlicher Ansatz.
- Das größte Problem wäre, dass es keine Möglichkeit gibt, zu definieren, „wie die API aussieht.“
- Das kann schnell zu Inkonsistenzen im Softwarelebenszyklus führen.
- Wenn wir die Signatur von einer Funktion geringfügig verändern, müssten wir das manuell in alle anderen Module ebenfalls einpflegen.
- Es gäbe auch keine Möglichkeit, dass ein Linter wie Ruff uns informieren könnte, wenn der Kode in einem Modul nicht mehr synchron mit den anderen ist.

Klassenhierarchien für APIs

- Klassen bieten uns die notwendige Abstraktion.



Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.
- Für jedes Ausgabeformat könnten wir eine Unterklasse von dieser Basisklasse ableiten, die die Operationen dann entsprechend implementiert.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.
- Für jedes Ausgabeformat könnten wir eine Unterklasse von dieser Basisklasse ableiten, die die Operationen dann entsprechend implementiert.
- Der Benutzercode könnte dann Dokumente aller Typen einheitlich benutzen, denn alle wären Instanzen von `Document` with exactly the same operations.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.
- Für jedes Ausgabeformat könnten wir eine Unterklasse von dieser Basisklasse ableiten, die die Operationen dann entsprechend implementiert.
- Der Benutzercode könnte dann Dokumente aller Typen einheitlich benutzen, denn alle wären Instanzen von `Document` with exactly the same operations.
- Alle formatspezifischen Dinge wären unsichtbar für den Benutzer, genauso, wie es auch sein sollte.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.
- Für jedes Ausgabeformat könnten wir eine Unterklasse von dieser Basisklasse ableiten, die die Operationen dann entsprechend implementiert.
- Der Benutzercode könnte dann Dokumente aller Typen einheitlich benutzen, denn alle wären Instanzen von `Document` with exactly the same operations.
- Alle formatspezifischen Dinge wären unsichtbar für den Benutzer, genauso, wie es auch sein sollte.
- Linter können uns dann auch sagen, wenn eine Unterklasse der Spezifikation der API in der Basisklasse nicht richtig folgt.

Klassenhierarchien für APIs



- Klassen bieten uns die notwendige Abstraktion.
- Wir könnten eine Basisklasse `Document` für Dokument-Objekte definieren, die die notwendigen Operationen zum Einfügen von Text oder Grafiken definiert.
- Jede dieser Operationen könnte einfach einen `NotImplementedError` auslösen.
- Für jedes Ausgabeformat könnten wir eine Unterklasse von dieser Basisklasse ableiten, die die Operationen dann entsprechend implementiert.
- Der Benutzercode könnte dann Dokumente aller Typen einheitlich benutzen, denn alle wären Instanzen von `Document` with exactly the same operations.
- Alle formatspezifischen Dinge wären unsichtbar für den Benutzer, genauso, wie es auch sein sollte.
- Linter können uns dann auch sagen, wenn eine Unterklasse der Spezifikation der API in der Basisklasse nicht richtig folgt.
- Der zweite wichtige Use Case für Klassen ist daher, dass sie uns eine Abstraktion zum Definieren und Implementieren von APIs bereitstellen.

Use Case Zusammenfassung



- Klassen können daher zwei wichtige Probleme lösen, bei denen einfache Datentypen, Kollektionen, und Funktionen nicht wirklich geeignet sind.

- Klassen können daher zwei wichtige Probleme lösen, bei denen einfache Datentypen, Kollektionen, und Funktionen nicht wirklich geeignet sind:
 1. Sie erlauben es uns, klar und semantisch Daten und die dazugehörigen Operationen zusammen zu gruppieren.

- Klassen können daher zwei wichtige Probleme lösen, bei denen einfache Datentypen, Kollektionen, und Funktionen nicht wirklich geeignet sind:
 1. Sie erlauben es uns, klar und semantisch Daten und die dazugehörigen Operationen zusammen zu gruppieren.
 2. Sie geben uns eine einfache Möglichkeit, mehrere Operationen in eine API zu gruppieren, die dann – auf transparente Art – auf verschiedene Art implementiert werden kann.

- Klassen können daher zwei wichtige Probleme lösen, bei denen einfache Datentypen, Kollektionen, und Funktionen nicht wirklich geeignet sind:
 1. Sie erlauben es uns, klar und semantisch Daten und die dazugehörigen Operationen zusammen zu gruppieren.
 2. Sie geben uns eine einfache Möglichkeit, mehrere Operationen in eine API zu gruppieren, die dann – auf transparente Art – auf verschiedene Art implementiert werden kann.
- Wir werden nun also Klassen diskutieren.



Klassen Definieren



Klassen Definieren: Syntax

- Klassen sind Datentypen die Datenelemente und den Code, der auf diesen arbeitet, miteinander verbinden¹⁹.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen sind Datentypen die Datenelemente und den Code, der auf diesen arbeitet, miteinander verbinden¹⁹.
- Eine Klasse ist dabei im Grunde eine Blaupause, ein Konzept, wohingegen ein Objekt eine konkrete Instanz einer Klasse ist.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen sind Datentypen die Datenelemente und den Code, der auf diesen arbeitet, miteinander verbinden¹⁹.
- Eine Klasse ist dabei im Grunde eine Blaupause, ein Konzept, wohingegen ein Objekt eine konkrete Instanz einer Klasse ist.
- Zum Beispiel ist `int` im Grunde eine Klasse für Ganzzahlen, wohingegen `5` eine konkrete Instanz dieser Klasse ist.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen sind Datentypen die Datenelemente und den Code, der auf diesen arbeitet, miteinander verbinden¹⁹.
- Eine Klasse ist dabei im Grunde eine Blaupause, ein Konzept, wohingegen ein Objekt eine konkrete Instanz einer Klasse ist.
- Zum Beispiel ist `int` im Grunde eine Klasse für Ganzzahlen, wohingegen `5` eine konkrete Instanz dieser Klasse ist.
- Klassen werden mit dem Schlüsselwort `class` gefolgt vom Klassennamen und dem Doppelpunkt (`„:“`) deklariert.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, or `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the ":!")
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Klassen sind Datentypen die Datenelemente und den Code, der auf diesen arbeitet, miteinander verbinden¹⁹.
- Eine Klasse ist dabei im Grunde eine Blaupause, ein Konzept, wohingegen ein Objekt eine konkrete Instanz einer Klasse ist.
- Zum Beispiel ist `int` im Grunde eine Klasse für Ganzzahlen, wohingegen `5` eine konkrete Instanz dieser Klasse ist.
- Klassen werden mit dem Schlüsselwort `class` gefolgt vom Klassennamen und dem Doppelpunkt (`„:“`) deklariert.
- Der Körper der Klasse ist dann mit vier Leerzeichen eingerückt.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:    # or `class MyClass(MyBaseClass)`
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the ":!")
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Eine Klasse ist dabei im Grunde eine Blaupause, ein Konzept, wohingegen ein Objekt eine konkrete Instanz einer Klasse ist.
- Zum Beispiel ist `int` im Grunde eine Klasse für Ganzzahlen, wohingegen `5` eine konkrete Instanz dieser Klasse ist.
- Klassen werden mit dem Schlüsselwort `class` gefolgt vom Klassennamen und dem Doppelpunkt (,:") deklariert.
- Der Körper der Klasse ist dann mit vier Leerzeichen eingerückt.
- Er beinhaltet alles, was zur Klasse gehört, die Dokumentation, die Methoden, und die Attribute.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Zum Beispiel ist `int` im Grunde eine Klasse für Ganzzahlen, wohingegen `5` eine konkrete Instanz dieser Klasse ist.
- Klassen werden mit dem Schlüsselwort `class` gefolgt vom Klassennamen und dem Doppelpunkt (,:") deklariert.
- Der Körper der Klasse ist dann mit vier Leerzeichen eingerückt.
- Er beinhaltet alles, was zur Klasse gehört, die Dokumentation, die Methoden, und die Attribute.
- Das Erste, was nach der Klassendeklaration kommt, ist normalerweise der Docstring der Klasse.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11          #: Documentation of the meaning of attribute 1 (notice the "(!)
12          self.attribute_1: type_hint = initial_value
13          # ...
14
15      def my_method(self, param1: type_hint, param2: type_hint) -> result:
16          """
17          Docstring of my_method.
18
19          :param param1: the documentation of the first parameter.
20          :param param2: the documentation of the second parameter.
21          :returns: the documentation of the result of the method.
22          """
23          # compute something using the attributes
24          self.attribute_1 = ... # Assign value to attribute.
25          x = self.attribute_1 # Use the value of an attribute.
26          self.my_other_method(12) # Call other methods of the class.
27
28          # ... more methods
29
30
31  # Instantiating MyClass creates a new instance of MyClass.
32  # We can use MyClass as type hint for variables.
33  newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Er beinhaltet alles, was zur Klasse gehört, die Dokumentation, die Methoden, und die Attribute.
- Das Erste, was nach der Klassendeklaration kommt, ist normalerweise der Docstring der Klasse.
- Das kann eine einzelne, aussagekräftige Zeile sein oder eine mehrzeilige Dokumentation, die dann erst mit einer einzelnen Zusammenfassung anfängt, gefolgt von einer Leerzeile, gefolgt von mehreren Zeilen ausführlicher Dokumentation.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

Gute Praxis

Klassennamen sollten der „CapWords“-Konvention folgen, die auch oft *Camel Case* genannt wird, also aussehen wir `MyClass` oder `UniversityDepartment`, aber nicht wie `my_class` der `university_department`¹⁰⁶.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value for param1 of __init__)
```


Klassen Definieren: Syntax

Gute Praxis

Klassennamen sollten der „CapWords“-Konvention folgen, die auch oft *Camel Case* genannt wird, also aussehen wir `MyClass` oder `UniversityDepartment`, aber nicht wie `my_class` der `university_department`¹⁰⁶.

- Klassen können eine Initialisierer-Methode namens `__init__` haben.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)"
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value for param1 of __init__)
```

Klassen Definieren: Syntax

Gute Praxis

Klassennamen sollten der „CapWords“-Konvention folgen, die auch oft *Camel Case* genannt wird, also aussehen wir `MyClass` oder `UniversityDepartment`, aber nicht wie `my_class` der `university_department`¹⁰⁶.

- Klassen können eine Initialisierer-Methode namens `__init__` haben.
- Diese Spezialmethode kann beliebig viele Parameter haben, liefert aber niemals einen Rückgabewert zurück.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:    # or `class MyClass(MyBaseClass)`
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)"
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ...    # Assign value to attribute.
25         x = self.attribute_1    # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31     # Instantiating MyClass creates a new instance of MyClass.
32     # We can use MyClass as type hint for variables.
33     newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen können eine Initialisierer-Methode namens `__init__` haben.
- Diese Spezialmethode kann beliebig viele Parameter haben, liefert aber niemals einen Rückgabewert zurück.
- Sie wird mit Type Hints und einem Docstring wie eine normale Methode annotiert.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen können eine Initialisierer-Methode namens `__init__` haben.
- Diese Spezialmethode kann beliebig viele Parameter haben, liefert aber niemals einen Rückgabewert zurück.
- Sie wird mit Type Hints und einem Docstring wie eine normale Methode annotiert.
- Nur der Spezialparameter `self` wird nicht annotiert.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen können eine Initialisierer-Methode namens `__init__` haben.
- Diese Spezialmethode kann beliebig viele Parameter haben, liefert aber niemals einen Rückgabewert zurück.
- Sie wird mit Type Hints und einem Docstring wie eine normale Methode annotiert.
- Nur der Spezialparameter `self` wird nicht annotiert.
- Der Initialisierer `__init__` wird benutzt, um alle (nicht-geerbten) Attribute einer Klasseninstanz zu deklarieren und um ihnen anfängliche Werte zuzuweisen.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:    # or `class MyClass(MyBaseClass)`
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)"
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Diese Spezialmethode kann beliebig viele Parameter haben, liefert aber niemals einen Rückgabewert zurück.
- Sie wird mit Type Hints und einem Docstring wie eine normale Methode annotiert.
- Nur der Spezialparameter `self` wird nicht annotiert.
- Der Initialisierer `__init__` wird benutzt, um alle (nicht-geerbten) Attribute einer Klasseninstanz zu deklarieren und um ihnen anfängliche Werte zuzuweisen.
- In diesem Schritt geben wir auch Type Hints für die Attribute an.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Sie wird mit Type Hints und einem Docstring wie eine normale Methode annotiert.
- Nur der Spezialparameter `self` wird nicht annotiert.
- Der Initialisierer `__init__` wird benutzt, um alle (nicht-geerbten) Attribute einer Klasseninstanz zu deklarieren und um ihnen anfängliche Werte zuzuweisen.
- In diesem Schritt geben wir auch Type Hints für die Attribute an.
- In allen Methoden der Klasse wird die aktuelle Instanz der Klasse, das aktuelle Objekt, über den Name `self` referenziert.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30 # Instantiating MyClass creates a new instance of MyClass.
31 # We can use MyClass as type hint for variables.
32 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Nur der Spezialparameter `self` wird nicht annotiert.
- Der Initialisierer `__init__` wird benutzt, um alle (nicht-geerbten) Attribute einer Klasseninstanz zu deklarieren und um ihnen anfängliche Werte zuzuweisen.
- In diesem Schritt geben wir auch Type Hints für die Attribute an.
- In allen Methoden der Klasse wird die aktuelle Instanz der Klasse, das aktuelle Objekt, über den Name `self` referenziert.
- Wenn wir auf ein Attribut oder eine Methode der Klasse zugreifen, tun wir das immer über das Präfix `self..`

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Der Initialisierer `__init__` wird benutzt, um alle (nicht-geerbten) Attribute einer Klasseninstanz zu deklarieren und um ihnen anfängliche Werte zuzuweisen.
- In diesem Schritt geben wir auch Type Hints für die Attribute an.
- In allen Methoden der Klasse wird die aktuelle Instanz der Klasse, das aktuelle Objekt, über den Name `self` referenziert.
- Wenn wir auf ein Attribut oder eine Methode der Klasse zugreifen, tun wir das immer über das Präfix `self.`
- Wir deklarieren daher `self` immer als ersten Parameter jeder Methode.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the ":!")
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- In diesem Schritt geben wir auch Type Hints für die Attribute an.
- In allen Methoden der Klasse wird die aktuelle Instanz der Klasse, das aktuelle Objekt, über den Name `self` referenziert.
- Wenn wir auf ein Attribut oder eine Methode der Klasse zugreifen, tun wir das immer über das Präfix `self.`
- Wir deklarieren daher `self` immer als ersten Parameter jeder Methode.
- Eine Zeile wie `self.x: int = 5` in `__init__` erzeugt das Instanzenattribut `x`, type-hinted es als Ganzzahl, und weist ihm den Initialwert 5 zu.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Wenn wir auf ein Attribut oder eine Methode der Klasse zugreifen, tun wir das immer über das Präfix `self.`
- Wir deklarieren daher `self` immer als ersten Parameter jeder Methode.
- Eine Zeile wie `self.x: int = 5` in `__init__` erzeugt das Instanzenattribut `x`, type-hinted es als Ganzzahl, und weist ihm den Initialwert 5 zu.
- Wir können auch ein kurzes Kommentar, dass die Bedeutung des Attributes beschreibt, in die Zeile vor seiner Deklaration schreiben.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Wir deklarieren daher `self` immer als ersten Parameter jeder Methode.
- Eine Zeile wie `self.x: int = 5` in `__init__` erzeugt das Instanzenattribut `x`, type-hintet es als Ganzzahl, und weist ihm den Initialwert 5 zu.
- Wir können auch ein kurzes Kommentar, dass die Bedeutung des Attributes beschreibt, in die Zeile vor seiner Deklaration schreiben.
- Dieser Spezialkommentar fängt immer mit einem Doppelpunkt nach dem Hashmark an, also mit `#:`.⁹¹

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31     # Instantiating MyClass creates a new instance of MyClass.
32     # We can use MyClass as type hint for variables.
33     newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Wir können auch ein kurzes Kommentar, dass die Bedeutung des Attributes beschreibt, in die Zeile **vor** seiner Deklaration schreiben.
- Dieser Spezialkommentar fängt immer mit einem Doppelpunkt nach dem Hashmark an, also mit `#: 91`.
- Wenn wir `#: This is the x-coordinate.` in die Zeile vor die Deklaration von `self.x` schreiben, dann annotiert dies das Atribut mit einer Dokumentation die sagt, dass es, nun ja, eine x-Koordinate ist.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value for param1 of __init__)
```

Klassen Definieren: Syntax

- Wir können auch ein kurzes Kommentar, dass die Bedeutung des Attributes beschreibt, in die Zeile **vor** seiner Deklaration schreiben.
- Dieser Spezialkommentar fängt immer mit einem Doppelpunkt nach dem Hashmark an, also mit `#: 91`.
- Wenn wir `#: This is the x-coordinate.` in die Zeile vor die Deklaration von `self.x` schreiben, dann annotiert dies das Atribut mit einer Dokumentation die sagt, dass es, nun ja, eine x-Koordinate ist.
- Klassen können beliebig viele Methoden haben.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31     #: Instantiating MyClass creates a new instance of MyClass.
32     #: We can use MyClass as type hint for variables.
33     newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Dieser Spezialkommentar fängt immer mit einem Doppelpunkt nach dem Hashmark an, also mit `#:` ⁹¹.
- Wenn wir `#: This is the x-coordinate.` in die Zeile vor die Deklaration von `self.x` schreiben, dann annotiert dies das Attribut mit einer Dokumentation die sagt, dass es, nun ja, eine x-Koordinate ist.
- Klassen können beliebig viele Methoden haben.
- Eine Methode ist eine Funktion, die mit den Attributwerten einer Klasseninstanz arbeitet.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Wenn wir

`#: This is the x-coordinate.` in die Zeile vor die Deklaration von `self.x` schreiben, dann annotiert dies das Attribut mit einer Dokumentation die sagt, dass es, nun ja, eine x-Koordinate ist.

- Klassen können beliebig viele Methoden haben.
- Eine Methode ist eine Funktion, die mit den Attributwerten einer Klasseninstanz arbeitet.
- Jede Methode hat als ersten Parameter `self`, welches für das Objekt/die Instanz der Klasse steht, auf der die methode arbeitet.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:    # or `class MyClass(MyBaseClass)`
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)"
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen können beliebig viele Methoden haben.
- Eine Methode ist eine Funktion, die mit den Attributwerten einer Klasseninstanz arbeitet.
- Jede Methode hat als ersten Parameter `self`, welches für das Objekt/die Instanz der Klasse steht, auf der die methode arbeitet.
- Eine Methode kann beliebig viele andere Parameter und einen Rückgabewert haben.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the ":!")
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Klassen können beliebig viele Methoden haben.
- Eine Methode ist eine Funktion, die mit den Attributwerten einer Klasseninstanz arbeitet.
- Jede Methode hat als ersten Parameter `self`, welches für das Objekt/die Instanz der Klasse steht, auf der die methode arbeitet.
- Eine Methode kann beliebig viele andere Parameter und einen Rückgabewert haben.
- Alle Parameter außer `self` werden natürlich mit Type Hints annotiert.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the ":!")
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Eine Methode ist eine Funktion, die mit den Attributwerten einer Klasseninstanz arbeitet.
- Jede Methode hat als ersten Parameter `self`, welches für das Objekt/die Instanz der Klasse steht, auf der die methode arbeitet.
- Eine Methode kann beliebig viele andere Parameter und einen Rückgabewert haben.
- Alle Parameter außer `self` werden natürlich mit Type Hints annotiert.
- Methoden haben auch Docstrings, wie normale Funktionen.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Jede Methode hat als ersten Parameter `self`, welches für das Objekt/die Instanz der Klasse steht, auf der die Methode arbeitet.
- Eine Methode kann beliebig viele andere Parameter und einen Rückgabewert haben.
- Alle Parameter außer `self` werden natürlich mit Type Hints annotiert.
- Methoden haben auch Docstrings, wie normale Funktionen.
- In einer Methode können sowohl auf die Attribute als auch auf die Methoden einer Instanz über das `self`-Präfix zugreifen.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Eine Methode kann beliebig viele andere Parameter und einen Rückgabewert haben.
- Alle Parameter außer `self` werden natürlich mit Type Hints annotiert.
- Methoden haben auch Docstrings, wie normale Funktionen.
- In einer Methode können sowohl auf die Attribute als auch auf die Methoden einer Instanz über das `self`-Präfix zugreifen.
- Nachdem wir die Klasse definiert haben, können wir sie instantiieren.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Alle Parameter außer `self` werden natürlich mit Type Hints annotiert.
- Methoden haben auch Docstrings, wie normale Funktionen.
- In einer Methode können sowohl auf die Attribute als auch auf die Methoden einer Instanz über das `self`-Präfix zugreifen.
- Nachdem wir die Klasse definiert haben, können wir sie instantiieren.
- Dafür verwenden wir den Klassennamen wie eine normale Funktion.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Methoden haben auch Docstrings, wie normale Funktionen.
- In einer Methode können sowohl auf die Attribute als auch auf die Methoden einer Instanz über das `self`-Präfix zugreifen.
- Nachdem wir die Klasse definiert haben, können wir sie instantiieren.
- Dafür verwenden wir den Klassennamen wie eine normale Funktion.
- Dabei müssen wir Werte für alle Parameter von `__init__` angeben, außer für `self`.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- In einer Methode können sowohl auf die Attribute als auch auf die Methoden einer Instanz über das `self`-Präfix zugreifen.
- Nachdem wir die Klasse definiert haben, können wir sie instantiieren.
- Dafür verwenden wir den Klassennamen wie eine normale Funktion.
- Dabei müssen wir Werte für alle Parameter von `__init__` angeben, außer für `self`.
- Wir können Objekte der Klasse dann genauso wie normale Werte verwenden und sie z. B. in Variablen speichern.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Nachdem wir die Klasse definiert haben, können wir sie instantiieren.
- Dafür verwenden wir den Klassennamen wie eine normale Funktion.
- Dabei müssen wir Werte für alle Parameter von `__init__` angeben, außer für `self`.
- Wir können Objekte der Klasse dann genauso wie normale Werte verwenden und sie z. B. in Variablen speicher.
- Wir können den Klassennamen auch als Type Hint verwenden, denn entsteht ja für einen normalen Datentyp.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```


Klassen Definieren: Syntax

- Dafür verwenden wir den Klassennamen wie eine normale Funktion.
- Dabei müssen wir Werte für alle Parameter von `__init__` angeben, außer für `self`.
- Wir können Objekte der Klasse dann genauso wie normale Werte verwenden und sie z. B. in Variablen speichern.
- Wir können den Klassennamen auch als Type Hint verwenden, denn entsteht ja für einen normalen Datentyp.
- So viel zur Struktur von Klassen.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```

Klassen Definieren: Syntax

- Dabei müssen wir Werte für alle Parameter von `__init__` angeben, außer für `self`.
- Wir können Objekte der Klasse dann genauso wie normale Werte verwenden und sie z. B. in Variablen speichern.
- Wir können den Klassennamen auch als Type Hint verwenden, denn entsteht ja für einen normalen Datentyp.
- So viel zur Struktur von Klassen.
- Lassen Sie uns nun ohne weiteres Vorgeplänkel mit einem Beispiel beginnen.

```
1  """The basic syntax for defining classes in Python."""
2
3  class MyClass:
4      """The docstring of the class."""
5
6      def __init__(self, param1: type_hint) -> None:
7          """The docstring of the initializer __init__."""
8          # In this method, we initialize all the attributes of the class.
9          # Each attribute should get an initial value, `None` if needed be.
10
11         #: Documentation of the meaning of attribute 1 (notice the "(!)
12         self.attribute_1: type_hint = initial_value
13         # ...
14
15     def my_method(self, param1: type_hint, param2: type_hint) -> result:
16         """
17         Docstring of my_method.
18
19         :param param1: the documentation of the first parameter.
20         :param param2: the documentation of the second parameter.
21         :returns: the documentation of the result of the method.
22         """
23         # compute something using the attributes
24         self.attribute_1 = ... # Assign value to attribute.
25         x = self.attribute_1 # Use the value of an attribute.
26         self.my_other_method(12) # Call other methods of the class.
27
28     # ... more methods
29
30
31 # Instantiating MyClass creates a new instance of MyClass.
32 # We can use MyClass as type hint for variables.
33 newVar: MyClass = MyClass(value_for_param1_of__init__)
```



Beispiel



Beispiel: Punkte in der 2D-Ebene

- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.



Beispiel: Punkte in der 2D-Ebene

- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y -Koordinate definiert.



Beispiel: Punkte in der 2D-Ebene



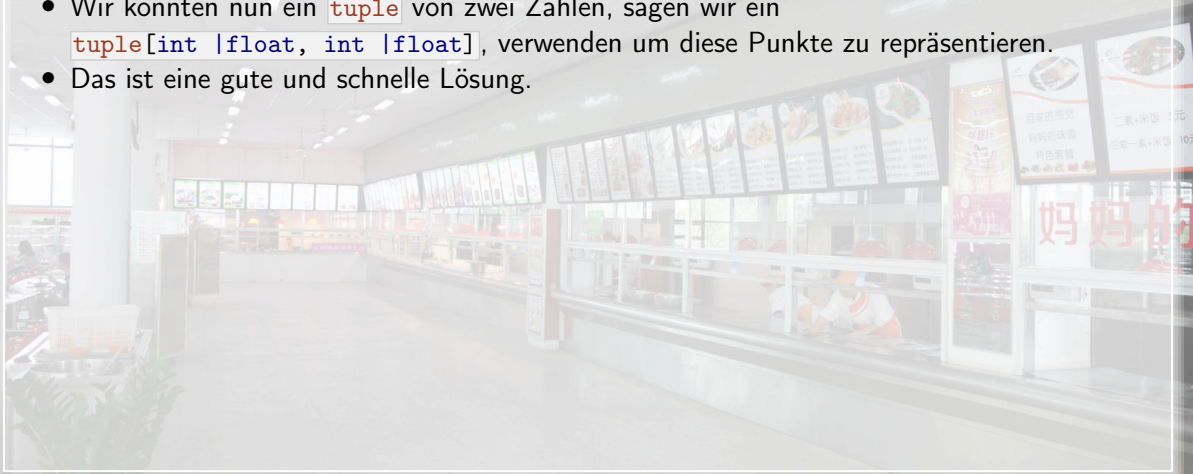
- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int | float, int | float]`, verwenden um diese Punkte zu repräsentieren.



Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int | float, int | float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.



Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.

Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.

Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.

Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.

Beispiel: Punkte in der 2D-Ebene



- Stellen Sie sich vor, das wir ein Programm zum Verarbeiten von Punkten in der zweidimensionalen Euklidischen Ebene schreiben wollen.
- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.

Beispiel: Punkte in der 2D-Ebene



- Jeder Punkt sei durch seine x und y-Koordinate definiert.
- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int | float, int | float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int | float, int | float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.

Beispiel: Punkte in der 2D-Ebene



- Wir könnten nun ein `tuple` von zwei Zahlen, sagen wir ein `tuple[int |float, int |float]`, verwenden um diese Punkte zu repräsentieren.
- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.

Beispiel: Punkte in der 2D-Ebene



- Das ist eine gute und schnelle Lösung.
- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int | float, int | float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern. . .

Beispiel: Punkte in der 2D-Ebene

- Aber dieser Lösung fehlt die Semantik, sie hat keine klare und offensichtliche Bedeutung.
- Nichts sagt dass ein `tuple[int |float, int |float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern. . .
- Das Ergebnis würde dann wenig Sinn ergeben.

Beispiel: Punkte in der 2D-Ebene



- Nichts sagt dass ein `tuple[int | float, int | float]` ein Punkt in der zweidimensionalen Euklidischen Ebene ist.
- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern...
- Das Ergebnis würde dann wenig Sinn ergeben.
- Trotzdem können solche Situation entstehen, z. B. durch missverstandene Dokumentation.

Beispiel: Punkte in der 2D-Ebene



- Es könnte genauso gut ein Tupel von Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing sein.
- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern...
- Das Ergebnis würde dann wenig Sinn ergeben.
- Trotzdem können solche Situation entstehen, z. B. durch missverstandene Dokumentation.
- Wenn ich mit Punkten der zweidimensionalen Euklidischen Ebene arbeite, dann habe ich im Idealfall eine Datenstruktur die klar und unverständlich für solche Punkte und nur solche Punkte entwickelt wurde.

Beispiel: Punkte in der 2D-Ebene



- Im Grunde ist es nur eine Gruppierung zweier Zahlen.
- Das selbe Fehlen von Semantik taucht wieder auf, wenn wir Operationen die diese Punkte verarbeiten implementieren wollen.
- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern. . .
- Das Ergebnis würde dann wenig Sinn ergeben.
- Trotzdem können solche Situation entstehen, z. B. durch missverstandene Dokumentation.
- Wenn ich mit Punkten der zweidimensionalen Euklidischen Ebene arbeite, dann habe ich im Idealfall eine Datenstruktur die klar und unverständlich für solche Punkte und nur solche Punkte entwickelt wurde.
- Die Operationen für Punkte sollten nur Instanzen dieser Datenstruktur als Input akzeptieren und Ausnahmen auslösen, wenn etwas anderes hereingegeben wird.

Beispiel: Punkte in der 2D-Ebene



- Eine Funktion die den Abstand zweier solcher Punkte berechnet würde einfach zwei solche Tupel als Input nehmen.
- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern. . .
- Das Ergebnis würde dann wenig Sinn ergeben.
- Trotzdem können solche Situation entstehen, z. B. durch missverstandene Dokumentation.
- Wenn ich mit Punkten der zweidimensionalen Euklidischen Ebene arbeite, dann habe ich im Idealfall eine Datenstruktur die klar und unverstündlich für solche Punkte und nur solche Punkte entwickelt wurde.
- Die Operationen für Punkte sollten nur Instanzen dieser Datenstruktur als Input akzeptieren und Ausnahmen auslösen, wenn etwas anderes hereingegeben wird.
- Wenn ich auf die x-Koordinate eines solchen Punktes zugreife, dann sollte sowohl von der Semantik als auch von den involvierten Namen absolut klar sein, dass das wirklich eine x-Koordinate ist und nicht irgendeine andere Zahl.

Beispiel: Punkte in der 2D-Ebene



- Natürlich sollte man dann nur solche Tupel als Argumente hereingeben, die auch wirklich Punkte in der zweidimensionalen Euklidischen Ebene repräsentieren.
- Aber niemand kann verhindern, dass ich andere Tupel hereingebe, z. B. solche, die wirklich nur eine Reisezeit und Reisekosten für ein Zugticket von Hefei nach Beijing speichern. . .
- Das Ergebnis würde dann wenig Sinn ergeben.
- Trotzdem können solche Situation entstehen, z. B. durch missverstandene Dokumentation.
- Wenn ich mit Punkten der zweidimensionalen Euklidischen Ebene arbeite, dann habe ich im Idealfall eine Datenstruktur die klar und unverständlich für solche Punkte und nur solche Punkte entwickelt wurde.
- Die Operationen für Punkte sollten nur Instanzen dieser Datenstruktur als Input akzeptieren und Ausnahmen auslösen, wenn etwas anderes hereingegeben wird.
- Wenn ich auf die x-Koordinate eines solchen Punktes zugreife, dann sollte sowohl von der Semantik als auch von den involvierten Namen absolut klar sein, dass das wirklich eine x-Koordinate ist und nicht irgendeine andere Zahl.
- Solch klare Semantik kann mit Klassen in Python erreicht werden.

Eine Klasse für Punkte

- Wir implementieren eine Klasse für Punkte in Datei `point.py`.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir implementieren eine Klasse für Punkte in Datei `point.py`.
- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.

```
1 """A simple class for points."""
```

```
2  
3 from math import isfinite, sqrt  
4 from typing import Final
```

```
5  
6  
7 class Point:
```

```
8     """  
9     A class for representing a point in the two-dimensional plane.
```

```
10  
11     >>> p = Point(1, 2.5)
```

```
12     >>> p.x
```

```
13     1
```

```
14     >>> p.y
```

```
15     2.5
```

```
16  
17     >>> try:
```

```
18         ...     Point(1, 1e308 * 1e308)
```

```
19         ... except ValueError as ve:
```

```
20         ...     print(ve)
```

```
21     x=1 and y=inf must both be finite.
```

```
22     """
```

```
23  
24     def __init__(self, x: int | float, y: int | float) -> None:
```

```
25         """
```

```
26         The constructor: Create a point and set its coordinates.
```

```
27  
28         :param x: the x-coordinate of the point
```

```
29         :param y: the y-coordinate of the point
```

```
30         """
```

```
31         if not (isfinite(x) and isfinite(y)):
```

```
32             raise ValueError(f"x={x} and y={y} must both be finite.")
```

```
33         #: the x-coordinate of the point
```

```
34         self.x: Final[int | float] = x
```

```
35         #: the y-coordinate of the point
```

```
36         self.y: Final[int | float] = y
```

```
37  
38     def distance(self, p: "Point") -> float:
```

```
39         """
```

```
40         Get the distance to another point.
```

```
41  
42         :param p: the other point
```

```
43         :return: the distance
```

```
44  
45     >>> Point(1, 1).distance(Point(4, 4))
```

```
46     4.242640687119285
```

```
47     """
```

```
48     return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir implementieren eine Klasse für Punkte in Datei `point.py`.
- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.
- Dann erstellen dann den Körper der Klasse, den wir mit vier Leerzeichen einrücken.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir implementieren eine Klasse für Punkte in Datei `point.py`.
- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.
- Dann erstellen dann den Körper der Klasse, den wir mit vier Leerzeichen einrücken.
- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.
- Dann erstellen dann den Körper der Klasse, den wir mit vier Leerzeichen einrücken.
- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.

Gute Praxis

An den Anfang der Klasse kommt immer ein Docstring, der beschreibt, wozu die Klasse gedacht ist.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.

Gute Praxis

An den Anfang der Klasse kommt immer ein Docstring, der beschreibt, wozu die Klasse gedacht ist. Dieser Docstring kann include Doctests um die Benutzung der Klasse zu verdeutlichen.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.

Gute Praxis

An den Anfang der Klasse kommt immer ein Docstring, der beschreibt, wozu die Klasse gedacht ist. Dieser Docstring kann include Doctests um die Benutzung der Klasse zu verdeutlichen. Solche Tests können aber auch in den Docstring des Moduls gepackt werden.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir implementieren eine Klasse für Punkte in Datei `point.py`.
- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.
- Dann erstellen dann den Körper der Klasse, den wir mit vier Leerzeichen einrücken.
- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.
- Danach definieren wir alle *Methoden* der `class`.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir erstellen die Klasse `Point` in dem wir `class Point:` schreiben.
- Dann erstellen dann den Körper der Klasse, den wir mit vier Leerzeichen einrücken.
- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.
- Danach definieren wir alle *Methoden* der `class`.
- Methoden sind wie Funktionen, nur das ihr erster Parameter immer `self` genannt wird und immer eine Instanz der Klasse, also ein Objekt, ist.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Das Erste, was wir in den Körper der Klasse schreiben, ist immer der Docstring.
- Danach definieren wir alle *Methoden* der `class`.
- Methoden sind wie Funktionen, nur das ihr erster Parameter immer `self` genannt wird und immer eine Instanz der Klasse, also ein Objekt, ist.
- So oder so, alle Methoden kommen in den Körper der Klasse.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Danach definieren wir alle *Methoden* der `class`.
- Methoden sind wie Funktionen, nur das ihr erster Parameter immer `self` genannt wird und immer eine Instanz der Klasse, also ein Objekt, ist.
- So oder so, alle Methoden kommen in den Körper der Klasse.
- Unsere Klasse `Point` bekommt zwei Attribute, `x` und `y`.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
    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 distance(self, p: "Point") -> float:
```

```
        """
```

```
        Get the distance to another point.
```

```
        :param p: the other point
```

```
        :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Danach definieren wir alle *Methoden* der `class`.
- Methoden sind wie Funktionen, nur das ihr erster Parameter immer `self` genannt wird und immer eine Instanz der Klasse, also ein Objekt, ist.
- So oder so, alle Methoden kommen in den Körper der Klasse.
- Unsere Klasse `Point` bekommt zwei Attribute, `x` und `y`.
- Ein Attribut ist eine Variable, die jede einzelne Instanz einer Klasse hat.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Methoden sind wie Funktionen, nur das ihr erster Parameter immer `self` genannt wird und immer eine Instanz der Klasse, also ein Objekt, ist.
- So oder so, alle Methoden kommen in den Körper der Klasse.
- Unsere Klasse `Point` bekommt zwei Attribute, `x` und `y`.
- Ein Attribut ist eine Variable, die jede einzelne Instanz einer Klasse hat.
- Später werden wir eine Instanz von `Point` mit der x-Koordinate 5 und der y-Koordinate 10 erstellen und dann eine andere Instanz mit der x-Koordinate 2 und der y-Koordinate 7.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- So oder so, alle Methoden kommen in den Körper der Klasse.
- Unsere Klasse `Point` bekommt zwei Attribute, `x` und `y`.
- Ein Attribut ist eine Variable, die jede einzelne Instanz einer Klasse hat.
- Später werden wir eine Instanz von `Point` mit der x-Koordinate 5 und der y-Koordinate 10 erstellen und dann eine andere Instanz mit der x-Koordinate 2 und der y-Koordinate 7.
- Jede Instanz von `Point` muss also diese beiden Attribute haben.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Unsere Klasse `Point` bekommt zwei Attribute, `x` und `y`.
- Ein Attribut ist eine Variable, die jede einzelne Instanz einer Klasse hat.
- Später werden wir eine Instanz von `Point` mit der x-Koordinate 5 und der y-Koordinate 10 erstellen und dann eine andere Instanz mit der x-Koordinate 2 und der y-Koordinate 7.
- Jede Instanz von `Point` muss also diese beiden Attribute haben.
- Darum braucht `Point` einen Initialisierer, also eine spezielle Methode, die diese Attribute erstellt und initialisiert.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Ein Attribut ist eine Variable, die jede einzelne Instanz einer Klasse hat.
- Später werden wir eine Instanz von `Point` mit der x-Koordinate 5 und der y-Koordinate 10 erstellen und dann eine andere Instanz mit der x-Koordinate 2 und der y-Koordinate 7.
- Jede Instanz von `Point` muss also diese beiden Attribute haben.
- Darum braucht `Point` einen Initialisierer, also eine spezielle Methode, die diese Attribute erstellt und initialisiert.
- Diese Methode wird `__init__` genannt.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Jede Instanz von `Point` muss also diese beiden Attribute haben.
- Darum braucht `Point` einen Initialisierer, also eine spezielle Methode, die diese Attribute erstellt und initialisiert.
- Diese Methode wird `__init__` genannt.
- Jede Methode einer Klasse muss den Parameter `self` haben, der die Instanz der Klasse (das Objekt), auf die die Methode angewandt wird, beinhaltet.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Jede Instanz von `Point` muss also diese beiden Attribute haben.
- Darum braucht `Point` einen Initialisierer, also eine spezielle Methode, die diese Attribute erstellt und initialisiert.
- Diese Methode wird `__init__` genannt.
- Jede Methode einer Klasse muss den Parameter `self` haben, der die Instanz der Klasse (das Objekt), auf die die Methode angewandt wird, beinhaltet.
- Der Initialisierer `__init__` ist eine spezielle Methode, deshalb hat auch er den Parameter `self`.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Diese Methode wird `__init__` genannt.
- Jede Methode einer Klasse muss den Parameter `self` haben, der die Instanz der Klasse (das Objekt), auf die die Methode angewandt wird, beinhaltet.
- Der Initialisierer `__init__` ist eine spezielle Methode, deshalb hat auch er den Parameter `self`.
- Zusätzlich verlangen wir, dass Werte für die beiden Parameter `x` und `y` angegeben werden, wann immer wir eine Instanz von `Point` erstellen.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Diese Methode wird `__init__` genannt.
- Jede Methode einer Klasse muss den Parameter `self` haben, der die Instanz der Klasse (das Objekt), auf die die Methode angewandt wird, beinhaltet.
- Der Initialisierer `__init__` ist eine spezielle Methode, deshalb hat auch er den Parameter `self`.
- Zusätzlich verlangen wir, dass Werte für die beiden Parameter `x` und `y` angegeben werden, wann immer wir eine Instanz von `Point` erstellen.
- Wir erlauben sowohl `ints` als auch `floats` für deren Typen.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Der Initialisierer `__init__` ist eine spezielle Methode, deshalb hat auch er den Parameter `self`.
- Zusätzlich verlangen wir, dass Werte für die beiden Parameter `x` und `y` angegeben werden, wann immer wir eine Instanz von `Point` erstellen.
- Wir erlauben sowohl `ints` als auch `floats` für deren Typen.
- In jeder Methode der Klasse kann auf die Attribute der Objekte über den Parameter `self` zugegriffen werden.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Der Initialisierer `__init__` ist eine spezielle Methode, deshalb hat auch er den Parameter `self`.
- Zusätzlich verlangen wir, dass Werte für die beiden Parameter `x` und `y` angegeben werden, wann immer wir eine Instanz von `Point` erstellen.
- Wir erlauben sowohl `ints` als auch `floats` für deren Typen.
- In jeder Methode der Klasse kann auf die Attribute der Objekte über den Parameter `self` zugegriffen werden.
- Wir können das Attribut `x` eines Objekts in einer Methode über `self.x` auslesen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Zusätzlich verlangen wir, dass Werte für die beiden Parameter `x` und `y` angegeben werden, wann immer wir eine Instanz von `Point` erstellen.
- Wir erlauben sowohl `ints` als auch `floats` für deren Typen.
- In jeder Methode der Klasse kann auf die Attribute der Objekte über den Parameter `self` zugegriffen werden.
- Wir können das Attribut `x` eines Objekts in einer Methode über `self.x` auslesen.
- Hierbei kann `self.x` wie eine normale lokale Variable verwendet werden.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir erlauben sowohl `ints` als auch `floats` für deren Typen.
- In jeder Methode der Klasse kann auf die Attribute der Objekte über den Parameter `self` zugegriffen werden.
- Wir können das Attribut `x` eines Objekts in einer Methode über `self.x` auslesen.
- Hierbei kann `self.x` wie eine normale lokale Variable verwendet werden.
- Wir können den Wert `a` in einem veränderlichen Attribut `x` des aktuellen Objekts in einer Methode speichern, in dem `self.x = a` schreiben.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Wir können das Attribut `x` eines Objekts in einer Methode über `self.x` auslesen.
- Hierbei kann `self.x` wie eine normale lokale Variable verwendet werden.
- Wir können den Wert `a` in einem veränderlichen Attribut `x` des aktuellen Objekts in einer Methode speichern, in dem `self.x = a` schreiben.
- Dieser Wert wird bleiben, bis er wieder manuell geändert wird, auch wenn die Ausführung der Methode beendet ist.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dieser Wert wird bleiben, bis er wieder manuell geändert wird, auch wenn die Ausführung der Methode beendet ist.

Gute Praxis

Objektattribute dürfen nur im Initialisierer `__init__` erstellt werden.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dieser Wert wird bleiben, bis er wieder manuell geändert wird, auch wenn die Ausführung der Methode beendet ist.

Gute Praxis

Objektattribute dürfen nur im Initialisierer `__init__` erstellt werden. Ein initialer Wert muss dort jedem Attribut sofort zugewiesen werden.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dieser Wert wird bleiben, bis er wieder manuell geändert wird, auch wenn die Ausführung der Methode beendet ist.
- Wir wollen nur endliche Koordinaten für unsere `Points` zulassen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dieser Wert wird bleiben, bis er wieder manuell geändert wird, auch wenn die Ausführung der Methode beendet ist.
- Wir wollen nur endliche Koordinaten für unsere `Points` zulassen.
- Es ist besser, sofort einen Fehler über eine Ausnahme zu signalisieren wenn wir auf ungültige Daten treffen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Wir wollen nur endliche Koordinaten für unsere `Points` zulassen.
- Es ist besser, sofort einen Fehler über eine Ausnahme zu signalisieren wenn wir auf ungültige Daten treffen.
- Wir wollen also sofort nicht-finite Koordinaten aussortieren, wenn ein `Point` erstellt wird.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
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 distance(self, p: "Point") -> float:
```

```
    """
```

```
    Get the distance to another point.
```

```
    :param p: the other point
```

```
    :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir wollen nur endliche Koordinaten für unsere `Points` zulassen.
- Es ist besser, sofort einen Fehler über eine Ausnahme zu signalisieren wenn wir auf ungültige Daten treffen.
- Wir wollen also sofort nicht-finite Koordinaten aussortieren, wenn ein `Point` erstellt wird.
- Daher ist das Erste, was wir im Initialisierer machen, die Koordinaten mit der `isfinite`-Funktion aus dem `math`-Modul zu prüfen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir wollen nur endliche Koordinaten für unsere `Points` zulassen.
- Es ist besser, sofort einen Fehler über eine Ausnahme zu signalisieren wenn wir auf ungültige Daten treffen.
- Wir wollen also sofort nicht-finite Koordinaten aussortieren, wenn ein `Point` erstellt wird.
- Daher ist das Erste, was wir im Initialisierer machen, die Koordinaten mit der `isfinite`-Funktion aus dem `math`-Modul zu prüfen.
- Wenn `x` oder `y` nicht finit sind, dann lösen wir sofort einen `ValueError` aus.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir wollen also sofort nicht-finite Koordinaten aussortieren, wenn ein `Point` erstellt wird.
- Daher ist das Erste, was wir im Initialisierer machen, die Koordinaten mit der `isfinite`-Funktion aus dem `math`-Modul zu prüfen.
- Wenn `x` oder `y` nicht finit sind, dann lösen wir sofort einen `ValueError` aus.
- Streng genommen könnten wir auch die Typen von `x` und `y` prüfen und z. B. einen `TypeError` auslösen, wenn sie nicht passen ... aber ich will das Beispiel nicht noch länger machen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn `x` oder `y` nicht finit sind, dann lösen wir sofort einen `ValueError` aus.
- Streng genommen könnten wir auch die Typen von `x` und `y` prüfen und z. B. einen `TypeError` auslösen, wenn sie nicht passen ... aber ich will das Beispiel nicht noch länger machen.
- Wenn die Koordinaten OK sind, dann setzen wir

```
self.x: Final[int | float] = x
und
self.y: Final[int | float] = y.
```

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
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 distance(self, p: "Point") -> float:
```

```
    """
```

```
    Get the distance to another point.
```

```
    :param p: the other point
```

```
    :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Streng genommen könnten wir auch die Typen von `x` und `y` prüfen und z. B. einen `TypeError` auslösen, wenn sie nicht passen ... aber ich will das Beispiel nicht noch länger machen.
- Wenn die Koordinaten OK sind, dann setzen wir
`self.x: Final[int | float] = x`
und
`self.y: Final[int | float] = y.`
- Diese Zeilen erstellen die Attribute `self.x` und `self.y` für das Objekt, das über den Parameter `self` hereingegeben wurde.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn die Koordinaten OK sind, dann setzen wir
`self.x: Final[int | float] = x`
und
`self.y: Final[int | float] = y.`
- Diese Zeilen erstellen die Attribute `self.x` und `self.y` für das Objekt, das über den Parameter `self` hereingegeben wurde.
- Der Type Hint `Final` aus dem Modul `typing` annotiert eine Variable oder ein Attribut als unveränderlich⁹⁵.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn die Koordinaten OK sind, dann setzen wir
`self.x: Final[int | float] = x`
und
`self.y: Final[int | float] = y.`
- Diese Zeilen erstellen die Attribute `self.x` und `self.y` für das Objekt, das über den Parameter `self` hereingegeben wurde.
- Der Type Hint `Final` aus dem Modul `typing` annotiert eine Variable oder ein Attribut als unveränderlich⁹⁵.
- Wir erlauben also nicht, dass die Koordinaten eines `Points` nachträglich verändert werden können.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Jedes Attribute eines Objekts muss mit einem Type Hint annotiert werden, wenn es im Initialisierer `__init__` erstellt wird⁵⁷.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Jedes Attribute eines Objekts muss mit einem Type Hint annotiert werden, wenn es im Initialisierer `__init__` erstellt wird⁵⁷. Dabei funktionieren Type Hints genau wie bei normalen Variablen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

Gute Praxis

Der Type Hint `Final` markiert eine Variable oder ein Attribut als unveränderlich.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Der Type Hint `Final` markiert eine Variable oder ein Attribut als unveränderlich. Alle Attribute, die Sie nach der Erstellung nicht mehr verändern wollen, sollten mit `Final` annotiert werden.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Der Type Hint `Final` markiert eine Variable oder ein Attribut als unveränderlich. Alle Attribute, die Sie nach der Erstellung nicht mehr verändern wollen, sollten mit `Final` annotiert werden. Dabei ist das natürlich nur ein Type Hint, wird also nicht vom Interpreter durchgesetzt⁹⁵ und bösartiger Code kann die Attribute immer noch verändern.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Der Type Hint `Final` markiert eine Variable oder ein Attribut als unveränderlich. Alle Attribute, die Sie nach der Erstellung nicht mehr verändern wollen, sollten mit `Final` annotiert werden. Dabei ist das natürlich nur ein Type Hint, wird also nicht vom Interpreter durchgesetzt⁹⁵ und bösartiger Code kann die Attribute immer noch verändern. Ein Type Checker wie Mypy kann aber solche falschen Veränderungen erkennen und Warnungen ausgeben.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
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 distance(self, p: "Point") -> float:
```

```
    """
```

```
    Get the distance to another point.
```

```
    :param p: the other point
```

```
    :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Ein Attribute wird mit einer Zeile *über* seiner Initialisierung dokumentiert, und zwar mit einem *Kommentar* das mit `#:` anfängt und die Bedeutung des Attributs erklärt⁹¹.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

Gute Praxis

Ein Attribut wird mit einer Zeile *über* seiner Initialisierung dokumentiert, und zwar mit einem *Kommentar* das mit `#:` anfängt und die Bedeutung des Attributs erklärt⁹¹. Manchmal wird die Dokumentation auch als String direkt unter dem Attribut angegeben³⁸, aber wir bleiben bei der obigen Methode, weil sie auch von Werkzeugen wie z. B. Sphinx unterstützt wird.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Diese Zeilen erstellen die Attribute `self.x` und `self.y` für das Objekt, das über den Parameter `self` hereingegeben wurde.
- Der Type Hint `Final` aus dem Modul `typing` annotiert eine Variable oder ein Attribut als unveränderlich⁹⁵.
- Wir erlauben also nicht, dass die Koordinaten eines `Points` nachträglich verändert werden können.
- Nach dem wir unseren Initialisierer geschrieben haben, können wir nun so etwas wie `p = Point(1, 2)` machen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Diese Zeilen erstellen die Attribute `self.x` und `self.y` für das Objekt, das über den Parameter `self` hereingegeben wurde.
- Der Type Hint `Final` aus dem Modul `typing` annotiert eine Variable oder ein Attribut als unveränderlich⁹⁵.
- Wir erlauben also nicht, dass die Koordinaten eines `Points` nachträglich verändert werden können.
- Nach dem wir unseren Initialisierer geschrieben haben, können wir nun so etwas wie `p = Point(1, 2)` machen.
- So wird ein neues Objekt als Instanz unserer Klasse `Point` erstellt.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Der Type Hint `Final` aus dem Modul `typing` annotiert eine Variable oder ein Attribut als unveränderlich⁹⁵.
- Wir erlauben also nicht, dass die Koordinaten eines `Points` nachträglich verändert werden können.
- Nach dem wir unseren Initialisierer geschrieben haben, können wir nun so etwas wie `p = Point(1, 2)` machen.
- So wird ein neues Objekt als Instanz unserer Klasse `Point` erstellt.
- Dafür wird zuerst der Speicher für das Objekt `p` reserviert.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir erlauben also nicht, dass die Koordinaten eines `Points` nachträglich verändert werden können.
- Nach dem wir unseren Initialisierer geschrieben haben, können wir nun so etwas wie `p = Point(1, 2)` machen.
- So wird ein neues Objekt als Instanz unserer Klasse `Point` erstellt.
- Dafür wird zuerst der Speicher für das Objekt `p` reserviert.
- Dann wird der Initialisierer als `__init__(p, 1, 2)` aufgerufen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Nach dem wir unseren Initialisierer geschrieben haben, können wir nun so etwas wie `p = Point(1, 2)` machen.
- So wird ein neues Objekt als Instanz unserer Klasse `Point` erstellt.
- Dafür wird zuerst der Speicher für das Objekt `p` reserviert.
- Dann wird der Initialisierer als `__init__(p, 1, 2)` aufgerufen.
- Nach dem der Initialisierer fertig ist, wird das Objekt in der Variable gespeichert und `p` zeigt nun auf das neue `Point`-Objekt.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- So wird ein neues Objekt als Instanz unserer Klasse `Point` erstellt.
- Dafür wird zuerst der Speicher für das Objekt `p` reserviert.
- Dann wird der Initialisierer als `__init__(p, 1, 2)` aufgerufen.
- Nach dem der Initialisierer fertig ist, wird das Objekt in der Variable gespeichert und `p` zeigt nun auf das neue `Point`-Objekt.
- Das Attribut `p.x` hat nun den Wert `1` und `p.y` hat den Wert `2`.

```
1 """A simple class for points."""
```

```
2  
3 from math import isfinite, sqrt  
4 from typing import Final
```

```
5  
6 class Point:
```

```
7     """  
8     A class for representing a point in the two-dimensional plane.
```

```
9  
10     >>> p = Point(1, 2.5)
```

```
11     >>> p.x
```

```
12     1
```

```
13     >>> p.y
```

```
14     2.5
```

```
15  
16     >>> try:
```

```
17         ...     Point(1, 1e308 * 1e308)
```

```
18         ... except ValueError as ve:
```

```
19         ...     print(ve)
```

```
20     x=1 and y=inf must both be finite.
```

```
21     """
```

```
22  
23  
24 def __init__(self, x: int | float, y: int | float) -> None:
```

```
25     """
```

```
26     The constructor: Create a point and set its coordinates.
```

```
27  
28     :param x: the x-coordinate of the point
```

```
29     :param y: the y-coordinate of the point
```

```
30     """
```

```
31     if not (isfinite(x) and isfinite(y)):
```

```
32         raise ValueError(f"x={x} and y={y} must both be finite.")
```

```
33     #: the x-coordinate of the point
```

```
34     self.x: Final[int | float] = x
```

```
35     #: the y-coordinate of the point
```

```
36     self.y: Final[int | float] = y
```

```
37  
38 def distance(self, p: "Point") -> float:
```

```
39     """
```

```
40     Get the distance to another point.
```

```
41  
42     :param p: the other point
```

```
43     :return: the distance
```

```
44  
45     >>> Point(1, 1).distance(Point(4, 4))
```

```
46     4.242640687119285
```

```
47     """
```

```
48     return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dafür wird zuerst der Speicher für das Objekt `p` reserviert.
- Dann wird der Initialisierer als `__init__(p, 1, 2)` aufgerufen.
- Nach dem der Initialisierer fertig ist, wird das Objekt in der Variable gespeichert und `p` zeigt nun auf das neue `Point`-Objekt.
- Das Attribut `p.x` hat nun den Wert 1 und `p.y` hat den Wert 2.
- Von dem Wissen, dass `p` eine Instanz von `Point` ist, können wir sofort schlussfolgern, dass `p.x` und `p.y` seine x- und y-Koordinaten sind.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Dann wird der Initialisierer als `__init__(p, 1, 2)` aufgerufen.
- Nach dem der Initialisierer fertig ist, wird das Objekt in der Variable gespeichert und `p` zeigt nun auf das neue `Point`-Objekt.
- Das Attribut `p.x` hat nun den Wert `1` und `p.y` hat den Wert `2`.
- Von dem Wissen, dass `p` eine Instanz von `Point` ist, können wir sofort schlussfolgern, dass `p.x` und `p.y` seine x- und y-Koordinaten sind.
- Es ist fast unmöglich, die Bedeutung dieser Variablen misszuverstehen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Nach dem der Initialisierer fertig ist, wird das Objekt in der Variable gespeichert und `p` zeigt nun auf das neue `Point`-Objekt.
- Das Attribut `p.x` hat nun den Wert 1 und `p.y` hat den Wert 2.
- Von dem Wissen, dass `p` eine Instanz von `Point` ist, können wir sofort schlussfolgern, dass `p.x` und `p.y` seine x- und y-Koordinaten sind.
- Es ist fast unmöglich, die Bedeutung dieser Variablen misszuverstehen.
- Natürlich helfen unsere Docstrings mit Doctest und unsere Type Hints dem Benutzer zusätzlich dabei, die Bedeutung zu verstehen.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Das Attribut `p.x` hat nun den Wert 1 und `p.y` hat den Wert 2.
- Von dem Wissen, dass `p` eine Instanz von `Point` ist, können wir sofort schlussfolgern, dass `p.x` und `p.y` seine x- und y-Koordinaten sind.
- Es ist fast unmöglich, die Bedeutung dieser Variablen misszuverstehen.
- Natürlich helfen unsere Docstrings mit Doctest und unsere Type Hints dem Benutzer zusätzlich dabei, die Bedeutung zu verstehen.
- Das wir nun eine Klasse für die Punkte der zweidimensionalen Ebene haben ist schon sehr schön.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Von dem Wissen, dass `p` eine Instanz von `Point` ist, können wir sofort schlussfolgern, dass `p.x` und `p.y` seine x- und y-Koordinaten sind.
- Es ist fast unmöglich, die Bedeutung dieser Variablen misszuverstehen.
- Natürlich helfen unsere Docstrings mit Doctest und unsere Type Hints dem Benutzer zusätzlich dabei, die Bedeutung zu verstehen.
- Das wir nun eine Klasse für die Punkte der zweidimensionalen Ebene haben ist schon sehr schön.
- Aber diese Klasse erlaubt es uns auch, Operationen auf Punkten durch Methoden zu definieren.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Es ist fast unmöglich, die Bedeutung dieser Variablen misszuverstehen.
- Natürlich helfen unsere Docstrings mit Doctest und unsere Type Hints dem Benutzer zusätzlich dabei, die Bedeutung zu verstehen.
- Das wir nun eine Klasse für die Punkte der zweidimensionalen Ebene haben ist schon sehr schön.
- Aber diese Klasse erlaubt es uns auch, Operationen auf Punkten durch Methoden zu definieren.
- Als Beispiel implementieren wir die Methode `distance`, die den Abstand zwischen zwei Punkten berechnet.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Das wir nun eine Klasse für die Punkte der zweidimensionalen Ebene haben ist schon sehr schön.
- Aber diese Klasse erlaubt es uns auch, Operationen auf Punkten durch Methoden zu definieren.
- Als Beispiel implementieren wir die Methode `distance`, die den Abstand zwischen zwei Punkten berechnet.
- Sie würden einen Punkt `p1` haben und könnten dann `p1.distance(p2)` aufrufen, um den Abstand zu einem anderen Punkt `p2` zu berechnen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Das was wir nun eine Klasse für die Punkte der zweidimensionalen Ebene haben ist schon sehr schön.
- Aber diese Klasse erlaubt es uns auch, Operationen auf Punkten durch Methoden zu definieren.
- Als Beispiel implementieren wir die Methode `distance`, die den Abstand zwischen zwei Punkten berechnet.
- Sie würden einen Punkt `p1` haben und könnten dann `p1.distance(p2)` aufrufen, um den Abstand zu einem anderen Punkt `p2` zu berechnen.
- Die Gleichung dafür kennen wir ja schon aus der letzten Einheit.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Aber diese Klasse erlaubt es uns auch, Operationen auf Punkten durch Methoden zu definieren.
- Als Beispiel implementieren wir die Methode `distance`, die den Abstand zwischen zwei Punkten berechnet.
- Sie würden einen Punkt `p1` haben und könnten dann `p1.distance(p2)` aufrufen, um den Abstand zu einem anderen Punkt `p2` zu berechnen.
- Die Gleichung dafür kennen wir ja schon aus der letzten Einheit.
- Wir importieren dafür die Funktion `sqrt` aus dem Modul `math`.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Sie würden einen Punkt `p1` haben und könnten dann `p1.distance(p2)` aufrufen, um den Abstand zu einem anderen Punkt `p2` zu berechnen.
- Die Gleichung dafür kennen wir ja schon aus der letzten Einheit.
- Wir importieren dafür die Funktion `sqrt` aus dem Modul `math`.
- Unsere Methode `distance` hat dann zwei Parameter, nämlich `self`, welche das Objekt, dessen Methode wir aufrufen, repräsentiert (`p1` im vorigen Beispiel) und `p`, das andere Punkt-Objekt (oder `p2` oben).

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Die Gleichung dafür kennen wir ja schon aus der letzten Einheit.
- Wir importieren dafür die Funktion `sqrt` aus dem Modul `math`.
- Unsere Methode `distance` hat dann zwei Parameter, nämlich `self`, welche das Objekt, dessen Methode wir aufrufen, repräsentiert (`p1` im vorigen Beispiel) und `p`, das andere Punkt-Objekt (oder `p2` oben).

- Sie berechnet dann die Euklidische Distanz als

```
sqrt((self.x - p.x)** 2  
+ (self.y - p.y)** 2).
```

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
    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 distance(self, p: "Point") -> float:
```

```
        """
```

```
        Get the distance to another point.
```

```
        :param p: the other point
```

```
        :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Die Gleichung dafür kennen wir ja schon aus der letzten Einheit.
- Wir importieren dafür die Funktion `sqrt` aus dem Modul `math`.
- Unsere Methode `distance` hat dann zwei Parameter, nämlich `self`, welche das Objekt, dessen Methode wir aufrufen, repräsentiert (`p1` im vorigen Beispiel) und `p`, das andere Punkt-Objekt (oder `p2` oben).
- Sie berechnet dann die Euklidische Distanz als
$$\sqrt{(self.x - p.x)^2 + (self.y - p.y)^2}$$
.
- In der Methode eines Objekts steht `self` immer für das Objekt selbst.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18          ...     Point(1, 1e308 * 1e308)
19          ... except ValueError as ve:
20              ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24      def __init__(self, x: int | float, y: int | float) -> None:
25          """
26          The constructor: Create a point and set its coordinates.
27
28          :param x: the x-coordinate of the point
29          :param y: the y-coordinate of the point
30          """
31          if not (isfinite(x) and isfinite(y)):
32              raise ValueError(f"x={x} and y={y} must both be finite.")
33          #: the x-coordinate of the point
34          self.x: Final[int | float] = x
35          #: the y-coordinate of the point
36          self.y: Final[int | float] = y
37
38      def distance(self, p: "Point") -> float:
39          """
40          Get the distance to another point.
41
42          :param p: the other point
43          :return: the distance
44
45          >>> Point(1, 1).distance(Point(4, 4))
46          4.242640687119285
47          """
48          return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Unsere Methode `distance` hat dann zwei Parameter, nämlich `self`, welche das Objekt, dessen Methode wir aufrufen, repräsentiert (`p1` im vorigen Beispiel) und `p`, das andere Punkt-Objekt (oder `p2` oben).
- Sie berechnet dann die Euklidische Distanz als
$$\sqrt{(\text{self.x} - \text{p.x})^2 + (\text{self.y} - \text{p.y})^2}.$$
- In der Methode eines Objekts steht `self` immer für das Objekt selbst.
- Deshalb ist `self.x` die x-Koordinate des aktuellen Objekts und `self.y` ist seine y-Koordinate.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Sie berechnet dann die Euklidische Distanz als
$$\sqrt{(self.x - p.x)^2 + (self.y - p.y)^2}.$$
- In der Methode eines Objekts steht `self` immer für das Objekt selbst.
- Deshalb ist `self.x` die x-Koordinate des aktuellen Objekts und `self.y` ist seine y-Koordinate.
- `p.x` ist die x-Koordinate des Punktes `p`, der als Argument hereingegeben wurde, und `p.y` ist seine y-Koordinate.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- In der Methode eines Objekts steht `self` immer für das Objekt selbst.
- Deshalb ist `self.x` die x-Koordinate des aktuellen Objekts und `self.y` ist seine y-Koordinate.
- `p.x` ist die x-Koordinate des Punktes `p`, der als Argument hereingegeben wurde, und `p.y` ist seine y-Koordinate.
- Beachten Sie, dass der Docstring nicht nur beschreibt, wie die Methode verwendet wird, sondern auch einen einfachen Doctest durchführt.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Deshalb ist `self.x` die x-Koordinate des aktuellen Objekts und `self.y` ist seine y-Koordinate.
- `p.x` ist die x-Koordinate des Punktes `p`, der als Argument hereingegeben wurde, und `p.y` ist seine y-Koordinate.
- Beachten Sie, dass der Docstring nicht nur beschreibt, wie die Methode verwendet wird, sondern auch einen einfachen Doctest durchführt.
- Wenn Sie `Point(1, 1).distance(Point(4, 4))`, dann erwarten Sie als Ergebnis so etwas wie 4.243...

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- `p.x` ist die x-Koordinate des Punktes `p`, der als Argument hereingegeben wurde, und `p.y` ist seine y-Koordinate.
- Beachten Sie, dass der Docstring nicht nur beschreibt, wie die Methode verwendet wird, sondern auch einen einfachen Doctest durchführt.
- Wenn Sie `Point(1, 1).distance(Point(4, 4))`, dann erwarten Sie als Ergebnis so etwas wie 4.243...
- In diesem Doctest – `Point(1, 1).distance(Point(4, 4))` übergeben wir nur einen einzelnen Parameter an die Methode `distance`.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Beachten Sie, dass der Docstring nicht nur beschreibt, wie die Methode verwendet wird, sondern auch einen einfachen Doctest durchführt.
- Wenn Sie `Point(1, 1).distance(Point(4, 4))`, dann erwarten Sie als Ergebnis so etwas wie 4.243...
- In diesem Doctest – `Point(1, 1).distance(Point(4, 4))` übergeben wir nur einen einzelnen Parameter an die Methode `distance`.
- Wenn wir die Methode `distance` aufrufen, dann brauchen wir keinen Wert für Parameter `self` direkt anzugeben.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn Sie `Point(1, 1).distance(Point(4, 4))`, dann erwarten Sie als Ergebnis so etwas wie 4.243...
- In diesem Doctest – `Point(1, 1).distance(Point(4, 4))` übergeben wir nur einen einzelnen Parameter an die Methode `distance`.
- Wenn wir die Methode `distance` aufrufen, dann brauchen wir keinen Wert für Parameter `self` direkt anzugeben.
- Er wird indirekt angegeben.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- In diesem Doctest – `Point(1, 1).distance(Point(4, 4))` übergeben wir nur einen einzelnen Parameter an die Methode `distance`.
- Wenn wir die Methode `distance` aufrufen, dann brauchen wir keinen Wert für Parameter `self` direkt anzugeben.
- Er wird indirekt angegeben:
- Wenn wir zwei Punkte `p1` und `p2` haben und `p1.distance(p2)` aufrufen, dann wird automatisch `self = p1` gesetzt.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn wir die Methode `distance` aufrufen, dann brauchen wir keinen Wert für Parameter `self` direkt anzugeben.
- Er wird indirekt angegeben:
- Wenn wir zwei Punkte `p1` und `p2` haben und `p1.distance(p2)` aufrufen, dann wird automatisch `self = p1` gesetzt.
- Obwohl wir unsere Methode als `def distance(self, p: "Point") -> float` deklariert haben, was so aussieht, als ob wir zwei Parameter angeben müssten (`self` und `p`), brauchen wir nur einen Wert anzugeben, nämlich für `p`.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Er wird indirekt angegeben:
- Wenn wir zwei Punkte `p1` und `p2` haben und `p1.distance(p2)` aufrufen, dann wird automatisch `self = p1` gesetzt.
- Obwohl wir unsere Methode als `def distance(self, p: "Point") -> float` deklariert haben, was so aussieht, als ob wir zwei Parameter angeben müssten (`self` und `p`), brauchen wir nur einen Wert anzugeben, nämlich für `p`.
- Wenn wir das lesen sehen wir, dass der Parameter `p` mit einem sehr eigenartigen Type Hint annotiert ist.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
    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 distance(self, p: "Point") -> float:
```

```
        """
```

```
        Get the distance to another point.
```

```
        :param p: the other point
```

```
        :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn wir zwei Punkte `p1` und `p2` haben und `p1.distance(p2)` aufrufen, dann wird automatisch `self = p1` gesetzt.
- Obwohl wir unsere Methode als `def distance(self, p: "Point") -> float` deklariert haben, was so aussieht, als ob wir zwei Parameter angeben müssten (`self` und `p`), brauchen wir nur einen Wert anzugeben, nämlich für `p`.
- Wenn wir das lesen sehen wir, dass der Parameter `p` mit einem sehr eigenartigen Type Hint annotiert ist:
- Wir würden erwarten, dass er mit `Point` annotiert wird.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

- Obwohl wir unsere Methode als `def distance(self, p: "Point") -> float` deklariert haben, was so aussieht, als ob wir zwei Parameter angeben müssten (`self` und `p`), brauchen wir nur einen Wert anzugeben, nämlich für `p`.
- Wenn wir das lesen sehen wir, dass der Parameter `p` mit einem sehr eigenartigen Type Hint annotiert ist:
- Wir würden erwarten, dass er mit `Point` annotiert wird.
- Stattdessen ist er mit dem String `"Point"` annotiert.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
    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 distance(self, p: "Point") -> float:
```

```
        """
```

```
        Get the distance to another point.
```

```
        :param p: the other point
```

```
        :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wenn wir das lesen sehen wir, dass der Parameter `p` mit einem sehr eigenartigen Type Hint annotiert ist:
- Wir würden erwarten, dass er mit `Point` annotiert wird.
- Stattdessen ist er mit dem String `"Point"` annotiert.
- Das hat den einfachen Grund dass die ganze Klasse `Point` erst *nach* ihrer Definition definiert ist, also nach dem ganzen Klassenkörper.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Wir würden erwarten, dass er mit `Point` annotiert wird.
- Stattdessen ist er mit dem String `"Point"` annotiert.
- Das hat den einfachen Grund dass die ganze Klasse `Point` erst *nach* ihrer Definition definiert ist, also nach dem ganzen Klassenkörper.
- Deshalb ist `Point` noch nicht als Typ *in* der Klassendefinition verfügbar.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Stattdessen ist er mit dem String `"Point"` annotiert.
- Das hat den einfachen Grund dass die ganze Klasse `Point` erst *nach* ihrer Definition definiert ist, also nach dem ganzen Klassenkörper.
- Deshalb ist `Point` noch nicht als Typ *in* der Klassendefinition verfügbar.
- Den String `"Point"` hier zu verwenden ist also nur behelfsmäßig und hat keinen weiteren Effekt.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Den String `"Point"` hier zu verwenden ist also nur behelfsmäßig und hat keinen weiteren Effekt.

Gute Praxis

Alle Methoden einer Klasse müssen mit Docstrings und Type Hints annotiert werden.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

Gute Praxis

Alle Methoden einer Klasse müssen mit Docstrings und Type Hints annotiert werden.

Gute Praxis

Wenn wir eine Klasse `C` als Type Hint *in* ihrer eigenen Definition bzw. ihrem eigenen Körper verwenden wollen, dann müssen wir `"C"` anstatt von `C` schreiben.

```
1 """A simple class for points."""
```

```
2  
3 from math import isfinite, sqrt  
4 from typing import Final
```

```
5  
6  
7 class Point:
```

```
8     """
```

```
9     A class for representing a point in the two-dimensional plane.
```

```
10  
11     >>> p = Point(1, 2.5)
```

```
12     >>> p.x
```

```
13     1
```

```
14     >>> p.y
```

```
15     2.5
```

```
16  
17     >>> try:
```

```
18         ...     Point(1, 1e308 * 1e308)
```

```
19         ... except ValueError as ve:
```

```
20         ...     print(ve)
```

```
21     x=1 and y=inf must both be finite.
```

```
22     """
```

```
23  
24     def __init__(self, x: int | float, y: int | float) -> None:
```

```
25         """
```

```
26         The constructor: Create a point and set its coordinates.
```

```
27  
28         :param x: the x-coordinate of the point
```

```
29         :param y: the y-coordinate of the point
```

```
30         """
```

```
31         if not (isfinite(x) and isfinite(y)):
```

```
32             raise ValueError(f"x={x} and y={y} must both be finite.")
```

```
33         #: the x-coordinate of the point
```

```
34         self.x: Final[int | float] = x
```

```
35         #: the y-coordinate of the point
```

```
36         self.y: Final[int | float] = y
```

```
37  
38     def distance(self, p: "Point") -> float:
```

```
39         """
```

```
40         Get the distance to another point.
```

```
41  
42         :param p: the other point
```

```
43         :return: the distance
```

```
44  
45     >>> Point(1, 1).distance(Point(4, 4))
```

```
46     4.242640687119285
```

```
47     """
```

```
48     return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```


Eine Klasse für Punkte

Gute Praxis

Alle Methoden einer Klasse müssen mit Docstrings und Type Hints annotiert werden.

Gute Praxis

Wenn wir eine Klasse `C` als Type Hint *in* ihrer eigenen Definition bzw. ihrem eigenen Körper verwenden wollen, dann müssen wir `"C"` anstatt von `C` schreiben. Andernfalls werden der Python-Interpreter und Werkzeuge zur statischen Kode-Analyse verwirrt.

```
1  """A simple class for points."""
2
3  from math import isfinite, sqrt
4  from typing import Final
5
6
7  class Point:
8      """
9      A class for representing a point in the two-dimensional plane.
10
11      >>> p = Point(1, 2.5)
12      >>> p.x
13      1
14      >>> p.y
15      2.5
16
17      >>> try:
18      ...     Point(1, 1e308 * 1e308)
19      ... except ValueError as ve:
20      ...     print(ve)
21      x=1 and y=inf must both be finite.
22      """
23
24  def __init__(self, x: int | float, y: int | float) -> None:
25      """
26      The constructor: Create a point and set its coordinates.
27
28      :param x: the x-coordinate of the point
29      :param y: the y-coordinate of the point
30      """
31      if not (isfinite(x) and isfinite(y)):
32          raise ValueError(f"x={x} and y={y} must both be finite.")
33      #: the x-coordinate of the point
34      self.x: Final[int | float] = x
35      #: the y-coordinate of the point
36      self.y: Final[int | float] = y
37
38  def distance(self, p: "Point") -> float:
39      """
40      Get the distance to another point.
41
42      :param p: the other point
43      :return: the distance
44
45      >>> Point(1, 1).distance(Point(4, 4))
46      4.242640687119285
47      """
48      return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Den String `"Point"` hier zu verwenden ist also nur behelfsmäßig und hat keinen weiteren Effekt.
- Wir könnten nun weitere Methoden erstellen, die vernünftige Berechnungen mit `Points` durchführen.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9     A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26         The constructor: Create a point and set its coordinates.
27
28         :param x: the x-coordinate of the point
29         :param y: the y-coordinate of the point
30         """
31         if not (isfinite(x) and isfinite(y)):
32             raise ValueError(f"x={x} and y={y} must both be finite.")
33         #: the x-coordinate of the point
34         self.x: Final[int | float] = x
35         #: the y-coordinate of the point
36         self.y: Final[int | float] = y
37
38     def distance(self, p: "Point") -> float:
39         """
40         Get the distance to another point.
41
42         :param p: the other point
43         :return: the distance
44
45         >>> Point(1, 1).distance(Point(4, 4))
46         4.242640687119285
47         """
48         return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Eine Klasse für Punkte

- Den String `"Point"` hier zu verwenden ist also nur behelfsmäßig und hat keinen weiteren Effekt.
- Wir könnten nun weitere Methoden erstellen, die vernünftige Berechnungen mit `Points` durchführen.
- Aber als erstes Beispiel reicht das eigentlich erstmal.

```
"""A simple class for points."""
```

```
from math import isfinite, sqrt  
from typing import Final
```

```
class Point:
```

```
    """
```

```
    A class for representing a point in the two-dimensional plane.
```

```
    >>> p = Point(1, 2.5)
```

```
    >>> p.x
```

```
    1
```

```
    >>> p.y
```

```
    2.5
```

```
    >>> try:
```

```
        ...     Point(1, 1e308 * 1e308)
```

```
        ... except ValueError as ve:
```

```
        ...     print(ve)
```

```
    x=1 and y=inf must both be finite.
```

```
    """
```

```
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 distance(self, p: "Point") -> float:
```

```
    """
```

```
    Get the distance to another point.
```

```
    :param p: the other point
```

```
    :return: the distance
```

```
    >>> Point(1, 1).distance(Point(4, 4))
```

```
    4.242640687119285
```

```
    """
```

```
    return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

Beispiel: Punkt-Klasse verwenden

- Verwenden wir nun unsere neue Klasse `Point` im Programm `point_user.py`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Verwenden wir nun unsere neue Klasse `Point` im Programm `point_user.py`.
- Zuerst müssen wir unsere Klasse `Point` importieren.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Verwenden wir nun unsere neue Klasse `Point` im Programm `point_user.py`.
- Zuerst müssen wir unsere Klasse `Point` importieren.
- Die Klasse `Point` ist in Datei `point.py` definiert.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Verwenden wir nun unsere neue Klasse `Point` im Programm `point_user.py`.
- Zuerst müssen wir unsere Klasse `Point` importieren.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname ohne das `.py` ist der Modulname, also `point`, von wo wir die Klasse importieren können.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Verwenden wir nun unsere neue Klasse `Point` im Programm `point_user.py`.
- Zuerst müssen wir unsere Klasse `Point` importieren.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname ohne das `.py` ist der Modulname, also `point`, von wo wir die Klasse importieren können.
- Wir schreiben also `from point import Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Zuerst müssen wir unsere Klasse `Point` importieren.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname ohne das `.py` ist der Modulname, also `point`, von wo wir die Klasse importieren können.
- Wir schreiben also `from point import Point`.
- Wir erzeugen nun eine Instanz von `Point` und speichern sie in der Variable `p1`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname ohne das `.py` ist der Modulname, also `point`, von wo wir die Klasse importieren können.
- Wir schreiben also `from point import Point`.
- Wir erzeugen nun eine Instanz von `Point` und speichern sie in der Variable `p1`.
- `p1` soll also eine Instanz von `Point` referenzieren, weshalb wir es mit einem entsprechenden Type Hint annotieren.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")    # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Dateiname ohne das `.py` ist der Modulename, also `point`, von wo wir die Klasse importieren können.
- Wir schreiben also `from point import Point`.
- Wir erzeugen nun eine Instanz von `Point` und speichern sie in der Variable `p1`.
- `p1` soll also eine Instanz von `Point` referenzieren, weshalb wir es mit einem entsprechenden Type Hint annotieren.
- Hier können wir `Point` genau wie jeden anderen Datentyp verwenden.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")  # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")         # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")  # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")  # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir schreiben also

```
from point import Point.
```

- Wir erzeugen nun eine Instanz von `Point` und speichern sie in der Variable `p1`.

- `p1` soll also eine Instanz von `Point` referenzieren, weshalb wir es mit einem entsprechenden Type Hint annotieren.

- Hier können wir `Point` genau wie jeden anderen Datentyp verwenden.

- Wir schreiben

```
p1: Point = Point(3, 5).
```

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")    # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }")  # This is obviously False.
11 print(f"{isinstance(p1, int) = }")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)        # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")     # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")     # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")     # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [        # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Wir erzeugen nun eine Instanz von `Point` und speichern sie in der Variable `p1`.
- `p1` soll also eine Instanz von `Point` referenzieren, weshalb wir es mit einem entsprechenden Type Hint annotieren.
- Hier können wir `Point` genau wie jeden anderen Datentyp verwenden.
- Wir schreiben `p1: Point = Point(3, 5)`.
- Der Initialisierer `__init__` automatisch aufgerufen, wenn wir `Point(3, 5)` ausführen.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `p1` soll also eine Instanz von `Point` referenzieren, weshalb wir es mit einem entsprechenden Type Hint annotieren.
- Hier können wir `Point` genau wie jeden anderen Datentyp verwenden.
- Wir schreiben
`p1: Point = Point(3, 5).`
- Der Initialisierer `__init__` automatisch aufgerufen, wenn wir `Point(3, 5)` ausführen.
- Die beiden Argumente, die wir hereingeben, werden die Werte für dessen Parameters `x` und `y`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")            # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")    # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")  # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)}")      # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)}")      # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Hier können wir `Point` genau wie jeden anderen Datentyp verwenden.
- Wir schreiben `p1: Point = Point(3, 5)`.
- Der Initialisierer `__init__` automatisch aufgerufen, wenn wir `Point(3, 5)` ausführen.
- Die beiden Argumente, die wir hereingeben, werden die Werte für dessen Parameters `x` und `y`.
- Der erste Parameter von `__init__` – `self` – ist dann die neu angelegte und uninitialisierte Instanz von `Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")        # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")             # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir schreiben `p1: Point = Point(3, 5).`
- Der Initialisierer `__init__` automatisch aufgerufen, wenn wir `Point(3, 5)` ausführen.
- Die beiden Argumente, die wir hereingeben, werden die Werte für dessen Parameters `x` und `y`.
- Der erste Parameter von `__init__` – `self` – ist dann die neu angelegte und uninitialisierte Instanz von `Point`.
- Nachdem `__init__` fertig ist, wird die neue Instanz von `Point`, die wir bekommen, ihr Attribut `x` auf 3 und ihr Attribut `y` auf 5 gesetzt haben.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")            # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")    # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")  # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Initialisierer `__init__` automatisch aufgerufen, wenn wir `Point(3, 5)` ausführen.
- Die beiden Argumente, die wir hereingeben, werden die Werte für dessen Parameter `x` und `y`.
- Der erste Parameter von `__init__` – `self` – ist dann die neu angelegte und uninitialisierte Instanz von `Point`.
- Nachdem `__init__` fertig ist, wird die neue Instanz von `Point`, die wir bekommen, ihr Attribut `x` auf 3 und ihr Attribut `y` auf 5 gesetzt haben.
- Wir können auf diese über `p1.x` und `p1.y` zugreifen.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")            # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")    # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")  # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)}")      # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)}")      # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Die beiden Argumente, die wir hereingeben, werden die Werte für dessen Parameters `x` und `y`.
- Der erste Parameter von `__init__` – `self` – ist dann die neu angelegte und uninitialisierte Instanz von `Point`.
- Nachdem `__init__` fertig ist, wird die neue Instanz von `Point`, die wir bekommen, ihr Attribut `x` auf 3 und ihr Attribut `y` auf 5 gesetzt haben.
- Wir können auf diese über `p1.x` und `p1.y` zugreifen.
- Natürlich können wir diese auch in f-Strings verwenden.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")        # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")    # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Der erste Parameter von `__init__` – `self` – ist dann die neu angelegte und uninitialisierte Instanz von `Point`.
- Nachdem `__init__` fertig ist, wird die neue Instanz von `Point`, die wir bekommen, ihr Attribut `x` auf 3 und ihr Attribut `y` auf 5 gesetzt haben.
- Wir können auf diese über `p1.x` und `p1.y` zugreifen.
- Natürlich können wir diese auch in f-Strings verwenden.
- Wir sehen das `f"{p1.x = }, {p1.y = }"` zu `"p1.x = 3, p1.y = 5"` interpoliert wird.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Nachdem `__init__` fertig ist, wird die neue Instanz von `Point`, die wir bekommen, ihr Attribut `x` auf 3 und ihr Attribut `y` auf 5 gesetzt haben.
- Wir können auf diese über `p1.x` und `p1.y` zugreifen.
- Natürlich können wir diese auch in f-Strings verwenden.
- Wir sehen das `f"{p1.x = }, {p1.y = }"` zu `"p1.x = 3, p1.y = 5"` interpoliert wird.
- Der Typ von `p1` ist `Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können auf diese über `p1.x` und `p1.y` zugreifen.
- Natürlich können wir diese auch in f-Strings verwenden.
- Wir sehen das `f"{p1.x = }, {p1.y = }"` zu `"p1.x = 3, p1.y = 5"` interpoliert wird.
- Der Typ von `p1` ist `Point`.
- Die Klasse `Point` ist in Datei `point.py` definiert.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Natürlich können wir diese auch in f-Strings verwenden.
- Wir sehen das `f"{p1.x = }, {p1.y = }"` zu `"p1.x = 3, p1.y = 5"` interpoliert wird.
- Der Typ von `p1` ist `Point`.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname wird als Module `point` interpretiert.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir sehen das

`f"{p1.x = }, {p1.y = }"` zu
`"p1.x = 3, p1.y = 5"` interpoliert
wird.

- Der Typ von `p1` ist `Point`.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname wird als Module `point` interpretiert.
- Daher ist der volle Name des Datentyps `point.Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }")  # This is obviously False.
11 print(f"{isinstance(p1, int) = }")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")     # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")     # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")     # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [        # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Typ von `p1` ist `Point`.
- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname wird als Module `point` interpretiert.
- Daher ist der volle Name des Datentyps `point.Point`.
- Und er ist eine `class`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Die Klasse `Point` ist in Datei `point.py` definiert.
- Der Dateiname wird als Module `point` interpretiert.
- Daher ist der volle Name des Datentyps `point.Point`.
- Und er ist eine `class`.
- Wenn wir `type(p1)` ausgeben, bekommen wir daher `<class 'point.Point'>`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Dateiname wird als Module `point` interpretiert.
- Daher ist der volle Name des Datentyps `point.Point`.
- Und er ist eine `class`.
- Wenn wir `type(p1)` ausgeben, bekommen wir daher `<class 'point.Point'>`.
- Wir können prüfen, ob ein Object `o` eine Instanz unserer Klasse `Point` ist, in dem wir schreiben `isinstance(o, Point)`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")            # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")    # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")  # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Daher ist der volle Name des Datentyps `point.Point`.
- Und er ist eine `class`.
- Wenn wir `type(p1)` ausgeben, bekommen wir daher `<class 'point.Point'>`.
- Wir können prüfen, ob ein Object `o` eine Instanz unserer Klasse `Point` ist, in dem wir schreiben `isinstance(o, Point)`.
- Für `p1` liefert das natürlich `True`, wie man erwarten würde.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Und er ist eine `class`.
- Wenn wir `type(p1)` ausgeben, bekommen wir daher `<class 'point.Point'>`.
- Wir können prüfen, ob ein Object `o` eine Instanz unserer Klasse `Point` ist, in dem wir schreiben `isinstance(o, Point)`.
- Für `p1` liefert das natürlich `True`, wie man erwarten würde.
- Als test prüfen wir `isinstance(5, Point)`, was aus offensichtlichen Gründen `False` ergibt.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wenn wir `type(p1)` ausgeben, bekommen wir daher `<class 'point.Point'>`.
- Wir können prüfen, ob ein Object `o` eine Instanz unserer Klasse `Point` ist, in dem wir schreiben `isinstance(o, Point)`.
- Für `p1` liefert das natürlich `True`, wie man erwarten würde.
- Als test prüfen wir `isinstance(5, Point)`, was aus offensichtlichen Gründen `False` ergibt.
- `isinstance(p1, int)` ist natürlich auch `False`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point                # Import our class from its module.
4
5 p1: Point = Point(3, 5)                # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")        # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")                # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }")   # Hence, this is True.
10 print(f"{isinstance(5, Point) = }")    # This is obviously False.
11 print(f"{isinstance(p1, int) = }")     # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)            # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")        # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")               # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")                # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")                # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")         # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")         # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [            # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können prüfen, ob ein Object `o` eine Instanz unserer Klasse `Point` ist, in dem wir schreiben `isinstance(o, Point)`.
- Für `p1` liefert das natürlich `True`, wie man erwarten würde.
- Als test prüfen wir `isinstance(5, Point)`, was aus offensichtlichen Gründen `False` ergibt.
- `isinstance(p1, int)` ist natürlich auch `False`.
- Jetzt erstellen wir eine zweite Instanz (`p2`) der Klasse `Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Für `p1` liefert das natürlich `True`, wie man erwarten würde.
- Als test prüfen wir `isinstance(5, Point)`, was aus offensichtlichen Gründen `False` ergibt.
- `isinstance(p1, int)` ist natürlich auch `False`.
- Jetzt erstellen wir eine zweite Instanz (`p2`) der Klasse `Point`.
- Diesmal übergeben wir die Argumente via die Parameternamen, schreiben also `x=8` und `y=7`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")               # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Als test prüfen wir `isinstance(5, Point)`, was aus offensichtlichen Gründen `False` ergibt.
- `isinstance(p1, int)` ist natürlich auch `False`.
- Jetzt erstellen wir eine zweite Instanz (`p2`) der Klasse `Point`.
- Diesmal übergeben wir die Argumente via die Parameternamen, schreiben also `x=8` und `y=7`.
- Diese Argumente werden dann wieder an `__init__` weitergereicht.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")    # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")            # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }")  # This is obviously False.
11 print(f"{isinstance(p1, int) = }")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)        # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")     # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")            # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")            # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")            # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")     # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")     # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [        # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `isinstance(p1, int)` ist natürlich auch `False`.
- Jetzt erstellen wir eine zweite Instanz (`p2`) der Klasse `Point`.
- Diesmal übergeben wir die Argumente via die Parameternamen, schreiben also `x=8` und `y=7`.
- Diese Argumente werden dann wieder an `__init__` weitergereicht.
- Das speichert `7` in `p2.x` und `8` in `p2.y`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)} = {p1}")       # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)} = {p2}")       # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {p1}")       # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {p2}")       # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {p1}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {p2}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Jetzt erstellen wir eine zweite Instanz (`p2`) der Klasse `Point`.
- Diesmal übergeben wir die Argumente via die Parameternamen, schreiben also `x=8` und `y=7`.
- Diese Argumente werden dann wieder an `__init__` weitergereicht.
- Das speichert 7 in `p2.x` und 8 in `p2.y`.
- Wir können diese Attributwerte wieder mit einem f-String ausgeben.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")        # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")        # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Diesmal übergeben wir die Argumente via die Parameternamen, schreiben also `x=8` und `y=7`.
- Diese Argumente werden dann wieder an `__init__` weitergereicht.
- Das speichert `7` in `p2.x` und `8` in `p2.y`.
- Wir können diese Attributwerte wieder mit einem f-String ausgeben.
- Der Typ von `p2` ist wieder die `class point.Point`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Diese Argumente werden dann wieder an `__init__` weitergereicht.
- Das speichert 7 in `p2.x` und 8 in `p2.y`.
- Wir können diese Attributwerte wieder mit einem f-String ausgeben.
- Der Typ von `p2` ist wieder die `class point.Point`.
- Unsere Objekte können auch mit dem `is`-Operator verwendet werden, welcher auf Objekt-Identität prüft.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)} = {p1}")      # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)} = {p2}")     # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {p1}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {p2}")     # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {p1}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {p2}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Das speichert 7 in `p2.x` und 8 in `p2.y`.
- Wir können diese Attributwerte wieder mit einem f-String ausgeben.
- Der Typ von `p2` ist wieder die `class point.Point`.
- Unsere Objekte können auch mit dem `is`-Operator verwendet werden, welcher auf Objekt-Identität prüft.
- `p1` ist natürlich das selbe Objekt wie es selbst, also ergibt `p1 is p1` dann `True`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")             # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")            # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")    # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")  # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)}")      # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)}")      # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können diese Attributwerte wieder mit einem f-String ausgeben.
- Der Typ von `p2` ist wieder die `class point.Point`.
- Unsere Objekte können auch mit dem `is`-Operator verwendet werden, welcher auf Objekt-Identität prüft.
- `p1` ist natürlich das selbe Objekt wie es selbst, also ergibt `p1 is p1` dann `True`.
- Obwohl sie Instanzen der gleichen Klasse sind, sind `p1` und `p2` natürlich verschiedene Objekte.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Typ von `p2` ist wieder die `class point.Point`.
- Unsere Objekte können auch mit dem `is`-Operator verwendet werden, welcher auf Objekt-Identität prüft.
- `p1` ist natürlich das selbe Objekt wie es selbst, also ergibt `p1 is p1` dann `True`.
- Obwohl sie Instanzen der gleichen Klasse sind, sind `p1` und `p2` natürlich verschiedene Objekte.
- Deshalb ist `p1 is p2` auch `False`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Unsere Objekte können auch mit dem `is`-Operator verwendet werden, welcher auf Objekt-Identität prüft.
- `p1` ist natürlich das selbe Objekt wie es selbst, also ergibt `p1 is p1` dann `True`.
- Obwohl sie Instanzen der gleichen Klasse sind, sind `p1` und `p2` natürlich verschiedene Objekte.
- Deshalb ist `p1 is p2` auch `False`.
- Wir können nun auch unsere Methode `distance` benutzen.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")       # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)} = {p1}")      # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}") # This is obviously False.
11 print(f"{isinstance(p1, int)}")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")       # p2.x = 7, p2.y = 8
15 print(f"{type(p2)} = {p2}")     # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {p1}")      # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {p2}")     # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {p1}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {p2}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `p1` ist natürlich das selbe Objekt wie es selbst, also ergibt `p1 is p1` dann `True`.
- Obwohl sie Instanzen der gleichen Klasse sind, sind `p1` und `p2` natürlich verschiedene Objekte.
- Deshalb ist `p1 is p2` auch `False`.
- Wir können nun auch unsere Methode `distance` benutzen.
- `p1.distance(p2)`, also der Abstand von `p1` zu `p2`, ist natürlich gleich `p2.distance(p1)`, also dem Abstand von `p2` zu `p1`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Obwohl sie Instanzen der gleichen Klasse sind, sind `p1` und `p2` natürlich verschiedene Objekte.
- Deshalb ist `p1 is p2` auch `False`.
- Wir können nun auch unsere Methode `distance` benutzen.
- `p1.distance(p2)`, also der Abstand von `p1` zu `p2`, ist natürlich gleich `p2.distance(p1)`, also dem Abstand von `p2` zu `p1`.
- Beide sind 5,
weil $\sqrt{(7-3)^2 + (8-5)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Deshalb ist `p1 is p2` auch `False`.
- Wir können nun auch unsere Methode `distance` benutzen.
- `p1.distance(p2)`, also der Abstand von `p1` zu `p2`, ist natürlich gleich `p2.distance(p1)`, also dem Abstand von `p2` zu `p1`.
- Beide sind 5,
weil $\sqrt{(7-3)^2 + (8-5)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$.
- `Point` kann wirklich wie jeder andere Datentyp verwendet werden.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")    # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können nun auch unsere Methode `distance` benutzen.
- `p1.distance(p2)`, also der Abstand von `p1` zu `p2`, ist natürlich gleich `p2.distance(p1)`, also dem Abstand von `p2` zu `p1`.
- Beide sind 5, weil $\sqrt{(7-3)^2 + (8-5)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$.
- `Point` kann wirklich wie jeder andere Datentyp verwendet werden.
- Wir können z. B. Listen von Instanzen von `Points` habe.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `p1.distance(p2)`, also der Abstand von `p1` zu `p2`, ist natürlich gleich `p2.distance(p1)`, also dem Abstand von `p2` zu `p1`.
- Beide sind 5, weil $\sqrt{(7-3)^2 + (8-5)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$.
- `Point` kann wirklich wie jeder andere Datentyp verwendet werden.
- Wir können z. B. Listen von Instanzen von `Points` habe.
- Der richtige Type Hint für so eine Liste ist dann `list[Point]`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")    # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Beide sind 5,
weil $\sqrt{(7-3)^2 + (8-5)^2} = \sqrt{4^2 + 3^2} = \sqrt{25} = 5$.
- `Point` kann wirklich wie jeder andere Datentyp verwendet werden.
- Wir können z. B. Listen von Instanzen von `Points` habe.
- Der richtige Type Hint für so eine Liste ist dann `list[Point]`.
- Wir können so eine Liste auch mit List Comprehension erstellen, die wir in Einheit 35 kennengelernt haben.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")               # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `Point` kann wirklich wie jeder andere Datentyp verwendet werden.
- Wir können z. B. Listen von Instanzen von `Points` habe.
- Der richtige Type Hint für so eine Liste ist dann `list[Point]`.
- Wir können so eine Liste auch mit List Comprehension erstellen, die wir in Einheit 35 kennengelernt haben.
- Wir können die Liste dann mit einem Generator-Ausdruck von Einheit 38, der die Punkte in Strings umwandelt, verarbeiten.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können z. B. Listen von Instanzen von `Points` habe.
- Der richtige Type Hint für so eine Liste ist dann `list[Point]`.
- Wir können so eine Liste auch mit List Comprehension erstellen, die wir in Einheit 35 kennengelernt haben.
- Wir können die Liste dann mit einem Generator-Ausdruck von Einheit 38, der die Punkte in Strings umwandelt, verarbeiten.
- Der Ausdruck interpoliert den f-String `f"({p.x}, {p.y})"` für jeden `Point p` in unserer Liste `point_list`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Der richtige Type Hint für so eine Liste ist dann `list[Point]`.
- Wir können so eine Liste auch mit List Comprehension erstellen, die wir in Einheit 35 kennengelernt haben.
- Wir können die Liste dann mit einem Generator-Ausdruck von Einheit 38, der die Punkte in Strings umwandelt, verarbeiten.
- Der Ausdruck interpoliert den f-String `f"({p.x}, {p.y})"` für jeden `Point p` in unserer Liste `point_list`.
- Eine Sequenz von Strings der Form `"(x, y)"` wird erstellt.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point                # Import our class from its module.
4
5 p1: Point = Point(3, 5)                # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")              # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")                   # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}")      # Hence, this is True.
10 print(f"{isinstance(5, Point)}")       # This is obviously False.
11 print(f"{isinstance(p1, int)}")        # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)            # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")              # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")                   # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")         # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [            # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können so eine Liste auch mit List Comprehension erstellen, die wir in Einheit 35 kennengelernt haben.
- Wir können die Liste dann mit einem Generator-Ausdruck von Einheit 38, der die Punkte in Strings umwandelt, verarbeiten.
- Der Ausdruck interpoliert den f-String `f"({p.x}, {p.y})"` für jeden `Point p` in unserer Liste `point_list`.
- Eine Sequenz von Strings der Form `"(x, y)"` wird erstellt.
- Diese wird dann zusammengefasst von der Methode `join` des Strings `", "` (siehe Einheit 23).

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")               # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Wir können die Liste dann mit einem Generator-Ausdruck von Einheit 38, der die Punkte in Strings umwandelt, verarbeiten.
- Der Ausdruck interpoliert den f-String `f"({p.x}, {p.y})"` für jeden `Point p` in unserer Liste `point_list`.
- Eine Sequenz von Strings der Form `"(x, y)"` wird erstellt.
- Diese wird dann zusammengefasst von der Methode `join` des Strings `", "` (siehe Einheit 23).
- Das Ergebnis sehen wir rechts unten.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")      # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")     # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [        # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Der Ausdruck interpoliert den f-String `f"({p.x}, {p.y})"` für jeden `Point p` in unserer Liste `point_list`.
- Eine Sequenz von Strings der Form `"(x, y)"` wird erstellt.
- Diese wird dann zusammengefasst von der Methode `join` des Strings `", "` (siehe Einheit 23).
- Das Ergebnis sehen wir rechts unten.
- `Point` ist ein Datentyp wie jeder anderer Datentyp.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Eine Sequenz von Strings der Form "(x, y)" wird erstellt.
- Diese wird dann zusammengefasst von der Methode `join` des Strings `", "` (siehe Einheit 23).
- Das Ergebnis sehen wir rechts unten.
- `Point` ist ein Datentyp wie jeder anderer Datentyp.
- Genaugenommen ist es der allererste Datentyp in Python den wir selbst erstellt haben.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1)}")              # <class 'point.Point'>
9 print(f"{isinstance(p1, Point)}") # Hence, this is True.
10 print(f"{isinstance(5, Point)}")  # This is obviously False.
11 print(f"{isinstance(p1, int)}")   # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x} = {p2.y}")         # p2.x = 7, p2.y = 8
15 print(f"{type(p2)}")              # <class 'point.Point'>
16
17 print(f"{p1 is p1} = {True}")     # True, because p1 is the same as p1.
18 print(f"{p1 is p2} = {False}")    # False, as these are two different instances.
19
20 print(f"{p1.distance(p2)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1)} = {5.0}") # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- Diese wird dann zusammengefasst von der Methode `join` des Strings `"", "` (siehe Einheit 23).
- Das Ergebnis sehen wir rechts unten.
- `Point` ist ein Datentyp wie jeder anderer Datentyp.
- Genaugenommen ist es der allererste Datentyp in Python den wir selbst erstellt haben.
- Das ist schon ziemlich cool, wenn man darüber nachdenkt.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")          # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)      # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")          # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")          # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [      # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```


Beispiel: Punkt-Klasse verwenden

- Das Ergebnis sehen wir rechts unten.
- `Point` ist ein Datentyp wie jeder anderer Datentyp.
- Genaugenommen ist es der allererste Datentyp in Python den wir selbst erstellt haben.
- Das ist schon ziemlich cool, wenn man darüber nachdenkt.
- Die Programmiersprache hat Datentypen wie `str` oder `list`.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")    # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")           # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

Beispiel: Punkt-Klasse verwenden

- `Point` ist ein Datentyp wie jeder anderer Datentyp.
- Genaugenommen ist es der allererste Datentyp in Python den wir selbst erstellt haben.
- Das ist schon ziemlich cool, wenn man darüber nachdenkt.
- Die Programmiersprache hate Datentypen wie `str` oder `list`.
- Nun können wir die Programmiersprache um unsere eigenen Datentypen erweitern.

```
1 """Examples of using our class :class:`Point`."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 print(f"{type(p1) = }")           # <class 'point.Point'>
9 print(f"{isinstance(p1, Point) = }") # Hence, this is True.
10 print(f"{isinstance(5, Point) = }") # This is obviously False.
11 print(f"{isinstance(p1, int) = }")  # This is obviously False, too.
12
13 p2: Point = Point(x=7, y=8)       # Create a second instance of Point.
14 print(f"{p2.x = }, {p2.y = }")   # p2.x = 7, p2.y = 8
15 print(f"{type(p2) = }")          # <class 'point.Point'>
16
17 print(f"{p1 is p1 = }")           # True, because p1 is the same as p1.
18 print(f"{p1 is p2 = }")           # False, as these are two different instances.
19
20 print(f"{p1.distance(p2) = }")    # sqrt(4^2 + 3^2) = 5.0
21 print(f"{p2.distance(p1) = }")    # sqrt(4^2 + 3^2) = 5.0
22
23 point_list: list[Point] = [       # Create list of points via comprehension.
24     Point(x, y) for x in range(3) for y in range(2)]
25 print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ python3 point_user.py ↓

```
1 p1.x = 3, p1.y = 5
2 type(p1) = <class 'point.Point'>
3 isinstance(p1, Point) = True
4 isinstance(5, Point) = False
5 isinstance(p1, int) = False
6 p2.x = 7, p2.y = 8
7 type(p2) = <class 'point.Point'>
8 p1 is p1 = True
9 p1 is p2 = False
10 p1.distance(p2) = 5.0
11 p2.distance(p1) = 5.0
12 (0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```



Veränderbar vs. Unveränderbar



Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden, zumindest nicht, ohne gegen die Regeln der Sprache zu verstoßen.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden, zumindest nicht, ohne gegen die Regeln der Sprache zu verstoßen.
- Das, was verhindern soll, dass sie verändert werden, ist der Type Hint `Final`.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden, zumindest nicht, ohne gegen die Regeln der Sprache zu verstoßen.
- Das, was verhindern soll, dass sie verändert werden, ist der Type Hint `Final`.
- Wie schon gesagt, Type Hints sind nur Hinweise und werden vom Interpreter nicht durchgesetzt⁹⁵.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden, zumindest nicht, ohne gegen die Regeln der Sprache zu verstoßen.
- Das, was verhindern soll, dass sie verändert werden, ist der Type Hint `Final`.
- Wie schon gesagt, Type Hints sind nur Hinweise und werden vom Interpreter nicht durchgesetzt⁹⁵.
- Die Attribute `x` und `y` von Instanzen der Klasse `Point` können also doch geändert werden.

Final macht unveränderbar?



- Die Objekte unserer Klasse `Point` sind *unveränderbar*.
- Nachdem sie erstellt wurden, können ihre Attribute nicht mehr verändert werden, zumindest nicht, ohne gegen die Regeln der Sprache zu verstoßen.
- Das, was verhindern soll, dass sie verändert werden, ist der Type Hint `Final`.
- Wie schon gesagt, Type Hints sind nur Hinweise und werden vom Interpreter nicht durchgesetzt⁹⁵.
- Die Attribute `x` und `y` von Instanzen der Klasse `Point` können also doch geändert werden.
- Werkzeuge wie Mypy und sogar PyCharm erkennen solche Fehler aber⁹⁵.

Beispiel für Verstoß gegen Final

- Wir probieren ja immer alles aus. Im Program `point_user_wrong.py` probiert das also auch mal aus.

```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {}, {p1.y} = {}".format(p1.x, p1.y)) # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x} = {}, {p1.y} = {}".format(p1.x, p1.y)) # p1.x = 5, p1.y = 5
```

↓ `python3 point_user_wrong.py` ↓

```
1 p1.x = 3, p1.y = 5
2 p1.x = 5, p1.y = 5
```

Beispiel für Verstoß gegen Final

- Wir probieren ja immer alles aus. Im Programm `point_user_wrong.py` probiert das also auch mal aus.
- Nachdem wir das `Point`-Object `p1` genau wie im vorigen Beispiel erstellt haben, setzen wir `p1.x = 5`.

```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x} = {p1.y}")         # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x} = {p1.y}")         # p1.x = 5, p1.y = 5
```

↓ `python3 point_user_wrong.py` ↓

```
1 p1.x = 3, p1.y = 5
2 p1.x = 5, p1.y = 5
```


Beispiel für Verstoß gegen Final

- Wir probieren ja immer alles aus. Im Program `point_user_wrong.py` probiert das also auch mal aus.
- Nachdem wir das `Point`-Object `p1` genau wie im vorigen Beispiel erstellt haben, setzen wir `p1.x = 5`.
- Der `Final` Type Hint sagt uns explizit, das wir das nicht machen sollen.

```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }")   # p1.x = 5, p1.y = 5
```

↓ `python3 point_user_wrong.py` ↓

```
1 p1.x = 3, p1.y = 5
2 p1.x = 5, p1.y = 5
```

Beispiel für Verstoß gegen Final

- Wir probieren ja immer alles aus. Im Programm `point_user_wrong.py` probiert das also auch mal aus.
- Nachdem wir das `Point`-Object `p1` genau wie im vorigen Beispiel erstellt haben, setzen wir `p1.x = 5`.
- Der `Final` Type Hint sagt uns explizit, das wir das nicht machen sollen.
- Wie die Ausgabe des Programms zeigt, können wir das aber trotzdem machen.

```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }")   # p1.x = 5, p1.y = 5
```

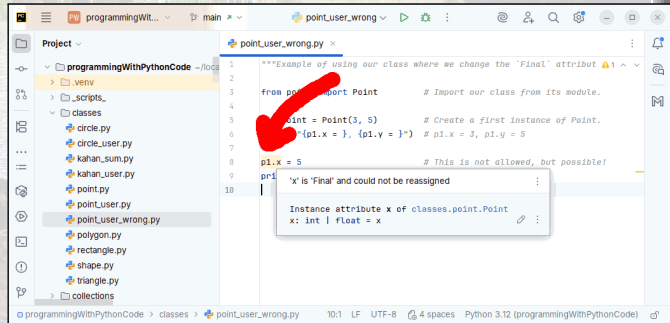
↓ `python3 point_user_wrong.py` ↓

```
1 p1.x = 3, p1.y = 5
2 p1.x = 5, p1.y = 5
```

Beispiel für Verstoß gegen Final

- Wir probieren ja immer alles aus. Im Programm `point_user_wrong.py` probiert das also auch mal aus.
- Nachdem wir das `Point`-Object `p1` genau wie im vorigen Beispiel erstellt haben, setzen wir `p1.x = 5`.
- Der `Final` Type Hint sagt uns explizit, das wir das nicht machen sollen.
- Wie die Ausgabe des Programms zeigt, können wir das aber trotzdem machen.
- Das das keine gute Idee ist, sehen wir schon, wenn wir das Programm in PyCharm öffnen.

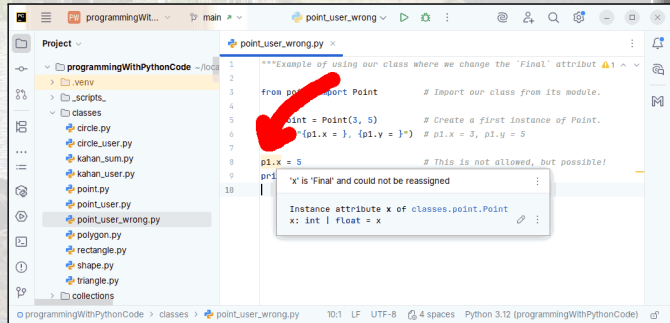
```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)          # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
7
8 p1.x = 5                          # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }")  # p1.x = 5, p1.y = 5
```



Beispiel für Verstoß gegen Final

- Nachdem wir das `Point`-Object `p1` genau wie im vorigen Beispiel erstellt haben, setzen wir `p1.x = 5`.
- Der `Final` Type Hint sagt uns explizit, das wir das nicht machen sollen.
- Wie die Ausgabe des Programms zeigt, können wir das aber trotzdem machen.
- Das das keine gute Idee ist, sehen wir schon, wenn wir das Programm in PyCharm öffnen.
- PyCharm hebt die fehlerhafte Zeile mit einer gelben Markierung hervor.

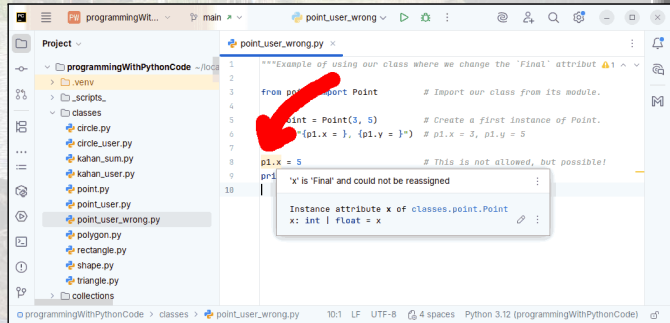
```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point          # Import our class from its module.
4
5 p1: Point = Point(3, 5)         # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }") # p1.x = 3, p1.y = 5
7
8 p1.x = 5                        # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }") # p1.x = 5, p1.y = 5
```



Beispiel für Verstoß gegen Final

- Der `Final` Type Hint sagt uns explizit, das wir das nicht machen sollen.
- Wie die Ausgabe des Programms zeigt, können wir das aber trotzdem machen.
- Das das keine gute Idee ist, sehen wir schon, wenn wir das Programm in PyCharm öffnen.
- PyCharm hebt die fehlerhafte Zeile mit einer gelben Markierung hervor.
- Halten wir die Maus über die Markierung, dann ploppt eine sehr verständliche Fehlermeldung auf.

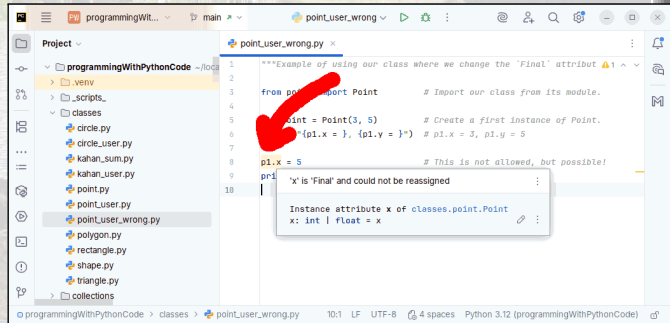
```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }")   # p1.x = 5, p1.y = 5
```



Beispiel für Verstoß gegen Final

- Wie die Ausgabe des Programms zeigt, können wir das aber trotzdem machen.
- Das das keine gute Idee ist, sehen wir schon, wenn wir das Programm in PyCharm öffnen.
- PyCharm hebt die fehlerhafte Zeile mit einer gelben Markierung hervor.
- Halten wir die Maus über die Markierung, dann ploppt eine sehr verständliche Fehlermeldung auf.
- Mypy gibt uns eine ganz ähnliche Warnung.

```
1 """Example of using our class where we change the `Final` attributes."""
2
3 from point import Point           # Import our class from its module.
4
5 p1: Point = Point(3, 5)           # Create a first instance of Point.
6 print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5
7
8 p1.x = 5                           # This is not allowed, but possible!
9 print(f"{p1.x = }, {p1.y = }")   # p1.x = 5, p1.y = 5
```



```
1 $ mypy point_user_wrong.py --no-strict-optional --check-untyped-defs
2 point_user_wrong.py:8: error: Cannot assign to final attribute "x"
   ↪ misc]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 1.18.2 failed with exit code 1.
```


Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.



Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.
- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.



Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.
- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.
- Why did we learn this right now?



Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.
- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.
- Why did we learn this right now?
- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.
- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.
- Why did we learn this right now?
- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Warum eigentlich unveränderbar?



- Die Attribute `x` und `y` werden von `__init__` initialisiert.
- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.
- Why did we learn this right now?
- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

Warum eigentlich unveränderbar?



- Dort werden sie mit dem Type Hint `Final` markiert und sie zu ändern ist daher ein Fehler.
- Why did we learn this right now?
- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

- Klassen zu erstellen, deren Instanzen unveränderlich sind, hat viele Vorteile.

Warum eigentlich unveränderbar?



- Why did we learn this right now?
- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

- Klassen zu erstellen, deren Instanzen unveränderlich sind, hat viele Vorteile, z. B.:
 1. Der Code wird leichter zu verstehen, weil wir nicht darüber nachdenken müssen, ob, wann, und wie ein Objekt verändert wird (weil es nicht verändert werden kann).

Warum eigentlich unveränderbar?



- Weil es in vielen Fällen eine gute Idee ist, Objekte unveränderlich zu machen.

Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

- Klassen zu erstellen, deren Instanzen unveränderlich sind, hat viele Vorteile, z. B.:
 1. Der Code wird leichter zu verstehen, weil wir nicht darüber nachdenken müssen, ob, wann, und wie ein Objekt verändert wird (weil es nicht verändert werden kann).
 2. Die Schlüssel in Mengen und Dictionaries müssen unveränderliche Objekte sein, weil diese Kollektionen die Objekte basierend auf ihren Hash Codes speichern, welche wiederum aus den Attributen berechnet werden.

Warum eigentlich unveränderbar?



Classes should be immutable unless there's a very good reason to make them mutable. . . . If a class cannot be made immutable, limit its mutability as much as possible.

— Joshua Bloch [9], 2008

Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

- Klassen zu erstellen, deren Instanzen unveränderlich sind, hat viele Vorteile, z. B.:
 1. Der Code wird leichter zu verstehen, weil wir nicht darüber nachdenken müssen, ob, wann, und wie ein Objekt verändert wird (weil es nicht verändert werden kann).
 2. Die Schlüssel in Mengen und Dictionaries müssen unveränderliche Objekte sein, weil diese Kollektionen die Objekte basierend auf ihren Hash Codes speichern, welche wiederum aus den Attributen berechnet werden. Wenn die Attribute sich ändern, dann ändern sich die Hash Codes, dann können die Objekte nicht mehr gefunden werden.

Warum eigentlich unveränderbar?



Definition: Unveränderlich (Immutable)

Nach der Initialisierung können die Attribute eines *unveränderlichen* Objekts nicht mehr verändert werden.

- Klassen zu erstellen, deren Instanzen unveränderlich sind, hat viele Vorteile, z. B.:
 1. Der Code wird leichter zu verstehen, weil wir nicht darüber nachdenken müssen, ob, wann, und wie ein Objekt verändert wird (weil es nicht verändert werden kann).
 2. Die Schlüssel in Mengen und Dictionaries müssen unveränderliche Objekte sein, weil diese Kollektionen die Objekte basierend auf ihren Hash Codes speichern, welche wiederum aus den Attributen berechnet werden. Wenn die Attribute sich ändern, dann ändern sich die Hash Codes, dann können die Objekte nicht mehr gefunden werden.
 3. Unveränderliche Objekte sind besonders nützlich bei paralleler Programmierung, wo veränderliche Variablen zu komplexen Bugs und Race Conditions führen kann.



Zusammenfassung



Zusammenfassung

- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.



Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.
- Natürlich verwenden wir Type Hints, Docstrings, und Doctests auch mit Klassen.

Zusammenfassung



- In dem wir Klassen definieren können wir unsere eigenen Datenstrukturen erstellen.
- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.
- Natürlich verwenden wir Type Hints, Docstrings, und Doctests auch mit Klassen.
- Wir können z. B. mit dem Type Hint ein Attribut als unveränderlich markieren.

Zusammenfassung



- Wir können Klassen genau wie alle anderen Datenstrukturen verwenden.
- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.
- Natürlich verwenden wir Type Hints, Docstrings, und Doctests auch mit Klassen.
- Wir können z. B. mit dem Type Hint ein Attribut als unveränderlich markieren.
- Leider setzt Python das dann nicht streng durch, man kann das Attribute also trotzdem ändern.

Zusammenfassung



- Klassen haben ihre eigenen Variablen, die Attribute genannt werden.
- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.
- Natürlich verwenden wir Type Hints, Docstrings, und Doctests auch mit Klassen.
- Wir können z. B. mit dem Type Hint ein Attribut als unveränderlich markieren.
- Leider setzt Python das dann nicht streng durch, man kann das Attribute also trotzdem ändern.
- Das aber ist eine Sünde, die Werkzeuge wie Mypy oder IDEs wie PyCharm melden.

Zusammenfassung



- Klassen können auch dazugehörige Funktionen haben, die Methoden genannt werden.
- Methoden können auf die Attribute zugreifen, um Dinge zu berechnen.
- Jede Klasse kann beliebig viele Methoden und Attribute haben.
- Der Initialisierer `__init__` ist eine spezielle Methode.
- Er wird aufgerufen, wann immer eine neue Instanz der Klasse erzeugt wird.
- Er erstellt alle Attribute einer Instanz und weist ihnen ihren ersten Wert zu.
- Natürlich verwenden wir Type Hints, Docstrings, und Doctests auch mit Klassen.
- Wir können z. B. mit dem Type Hint ein Attribut als unveränderlich markieren.
- Leider setzt Python das dann nicht streng durch, man kann das Attribute also trotzdem ändern.
- Das aber ist eine Sünde, die Werkzeuge wie Mypy oder IDEs wie PyCharm melden.
- Trotzdem ist es eine gute Idee, Attribute mit `Final` zu markieren, da man durch unveränderliche Attribute viele mögliche Probleme vermeiden kann.



谢谢您门！
Thank you!
Vielen Dank!



References I



- [1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also² (siehe S. 290, 305).
- [2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also¹ (siehe S. 290, 305).
- [3] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 305, 307).
- [4] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 305).
- [5] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 308).
- [6] Ben Beitler. *Hands-On Microsoft Access 2019*. Birmingham, England, UK: Packt Publishing Ltd, März 2020. ISBN: 978-1-83898-747-3 (siehe S. 305).
- [7] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...*. Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: <https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt> (besucht am 2025-02-05) (siehe S. 308).
- [8] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 303).
- [9] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: 978-0-321-35668-0 (siehe S. 263–272, 304).
- [10] Bernard Obeng Boateng. *Data Modeling with Microsoft Excel*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-028-2 (siehe S. 305).
- [11] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 306).

References II



- [12] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 305).
- [13] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 303).
- [14] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: <https://fstring.help> (besucht am 2024-07-25) (siehe S. 304).
- [15] Brett Cannon, Jiwon Seo, Yury Selivanov und Larry Hastings. *Function Signature Object*. Python Enhancement Proposal (PEP) 362. Beaverton, OR, USA: Python Software Foundation (PSF), 21. Aug. 2006–4. Juni 2012. URL: <https://peps.python.org/pep-0362> (besucht am 2024-12-12) (siehe S. 307).
- [16] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 304, 307).
- [17] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: <https://cacm.acm.org/research/50-years-of-queries> (besucht am 2025-01-09) (siehe S. 307).
- [18] Christmas, FL, USA: Simon Sez IT. *Microsoft Access 2021 – Beginner to Advanced*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2023. ISBN: 978-1-83546-911-8 (siehe S. 305).
- [19] "Classes". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 9. URL: <https://docs.python.org/3/tutorial/classes.html> (besucht am 2025-09-19) (siehe S. 24–27, 56–60).
- [20] David Clinton und Christopher Negus. *Ubuntu Linux Bible*. 10. Aufl. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 10. Nov. 2020. ISBN: 978-1-119-72233-5 (siehe S. 307, 308).
- [21] Edgar Frank „Ted“ Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (besucht am 2025-01-05) (siehe S. 306).

References III



- [22] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 307).
- [23] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (besucht am 2025-02-27) (siehe S. 307).
- [24] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 307).
- [25] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[?]: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 308).
- [26] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 306).
- [27] 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. 303).
- [28] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/library/doctest.html> (besucht am 2024-11-07) (siehe S. 304).
- [29] Pooyan Doozandeh und Frank E. Ritter. "Some Tips for Academic Writing and Using Microsoft Word". *XRDS: Crossroads, The ACM Magazine for Students* 26(1):10–11, Herbst 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1528-4972. doi:10.1145/3351470 (siehe S. 28–36, 305).
- [30] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 305, 306).

References IV



- [31] Steve Fanning, Vasudev Narayanan, „flywire“, Olivier Hallot, Jean Hollis Weber, Jenna Sargent, Pulkit Krishna, Dan Lewis, Peter Schofield, Jochen Schiffrers, Robert Großkopf, Jost Lange, Martin Fox, Hazel Russman, Steve Schwettman, Alain Romenne, Andrew Pitonyak, Jean-Pierre Ledure, Drew Jensen und Randolph Gam. *Base Guide 7.3. Revision 1. Based on LibreOffice 7.3 Community*. Berlin, Germany: The Document Foundation, Aug. 2022. URL: <https://books.libreoffice.org/en/BG73/BG73-BaseGuide.pdf> (besucht am 2025-01-13) (siehe S. 305).
- [32] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 306).
- [33] *PDF 32000-1:2008 – Document Management – Portable Document Format – Part 1: PDF 1.7*. 1. Aufl. San Jose, CA, USA: Adobe Systems Incorporated, 1. Juli 2008. URL: https://pdf-lib.js.org/assets/with_large_page_count.pdf (besucht am 2024-12-12) (siehe S. 28–36, 306).
- [34] “Formatted String Literals”. In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 7.1.1. URL: <https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals> (besucht am 2024-07-25) (siehe S. 304).
- [35] Jonas Gamalielsson und Björn Lundell. “Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice”. In: *8th IFIP WG 2.13 International Conference on Open Source Systems: Long-Term Sustainability OSS’2012*. 10.–13. Sep. 2012, Hammamet, Tunisia. Hrsg. von Imed Hammouda, Björn Lundell, Tommi Mikkonen und Walt Scacchi. Bd. 378. IFIP Advances in Information and Communication Technology (IFIPACT). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2012, S. 29–47. ISSN: 1868-4238. ISBN: 978-3-642-33441-2. doi:10.1007/978-3-642-33442-9_3 (siehe S. 28–36, 304, 305).
- [36] Bhavesh Gawade. “Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings”. In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, 25. Apr.–3. Juni 2025. URL: <https://code-b.dev/blog/f-strings-in-python> (besucht am 2025-08-04) (siehe S. 304).
- [37] David Goodger und Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Mai–13. Juni 2001. URL: <https://peps.python.org/pep-0257> (besucht am 2024-07-27) (siehe S. 303).

References V



- [38] Michael Goodwin. *reStructuredText Docstring Format*. Techn. Ber. PEP287. 25. März–2. Apr. 2002. URL: <https://peps.python.org/pep-0287> (besucht am 2024-12-12) (siehe S. 154, 155).
- [39] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: <https://www.ibm.com/topics/api> (besucht am 2024-12-12) (siehe S. 303).
- [40] Olaf Górski. "Why f-strings are awesome: Performance of different string concatenation methods in Python". In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., 8. Nov. 2022. URL: <https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97> (besucht am 2025-08-04) (siehe S. 304).
- [41] Dawn Griffiths. *Excel Cookbook – Receipts for Mastering Microsoft Excel*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2024. ISBN: 978-1-0981-4332-9 (siehe S. 305).
- [42] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 306).
- [43] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 306).
- [44] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 305).
- [45] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14. Aufl. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (siehe S. 304, 308).
- [46] Manuel Hoffmann, Frank Nagle und Yanuo Zhou. *The Value of Open Source Software*. Working Paper 24-038. Boston, MA, USA: Harvard Business School, 1. Jan. 2024. URL: https://www.hbs.edu/ris/Publication%20Files/24-038_51f8444f-502c-4139-8bf2-56eb4b65c58a.pdf (besucht am 2025-06-04) (siehe S. 306).

References VI



- [47] 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. 306).
- [48] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. 307).
- [49] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78–1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: <https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf> (besucht am 2024-08-23) (siehe S. 305).
- [50] Holger Krekel und pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Kap. 2.8, S. 65–69. URL: <https://docs.pytest.org/en/stable/how-to/doctest.html> (besucht am 2024-11-07) (siehe S. 304).
- [51] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: <https://readthedocs.org/projects/pytest/downloads/pdf/latest> (besucht am 2024-11-07) (siehe S. 306).
- [52] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 307).
- [53] Joan Lambert und Curtis Frye. *Microsoft Office Step by Step (Office 2021 and Microsoft 365)*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Juni 2022. ISBN: 978-0-13-754493-6 (siehe S. 305).
- [54] 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. 307).
- [55] 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. 306).

References VII



- [56] Jukka Lehtosalo, Ivan Levkivskiy, 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. 306).
- [57] Jukka Lehtosalo und Mypy Contributors. *Welcome to Mypy Documentation! (Mypy 1.13.0 documentation)*. Portland, OR, USA: Read the Docs, Inc., 22. Okt. 2024. URL: <https://mypy.readthedocs.io> (besucht am 2024-12-12) (siehe S. 148, 149).
- [58] *LibreOffice – The Document Foundation*. Berlin, Germany: The Document Foundation, 2024. URL: <https://www.libreoffice.org> (besucht am 2024-12-12) (siehe S. 28–36, 304, 305).
- [59] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. “Client-Server Architecture”. In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: <https://www.britannica.com/technology/client-server-architecture> (besucht am 2025-01-20) (siehe S. 303).
- [60] 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. 304).
- [61] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 306).
- [62] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: <https://mariadb.com/kb/en/documentation> (besucht am 2025-04-24) (siehe S. 305).
- [63] Charlie Marsh. “Ruff”. In: URL: <https://pypi.org/project/ruff> (besucht am 2025-08-29) (siehe S. 306).
- [64] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: <https://docs.astral.sh/ruff> (besucht am 2024-08-23) (siehe S. 306).
- [65] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, Juni 2017. ISBN: 978-1-4919-9486-3 (siehe S. 304).
- [66] Ron McFadyen und Cindy Miller. *Relational Databases and Microsoft Access*. 3. Aufl. Palatine, IL, USA: Harper College, 2014–2019. URL: <https://harpercollege.pressbooks.pub/relationaldatabases> (besucht am 2025-04-11) (siehe S. 305).

References VIII



- [67] MDN Contributors. *Signature (Functions)*. San Francisco, CA, USA: Mozilla Corporation, 8. Juni 2023. URL: <https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function> (besucht am 2024-12-12) (siehe S. 307).
- [68] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 307).
- [69] *Microsoft Word*. Redmond, WA, USA: Microsoft Corporation, 2024. URL: <https://www.microsoft.com/en-us/microsoft-365/word> (besucht am 2024-12-12) (siehe S. 28–36, 305).
- [70] Cameron Newham und Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (siehe S. 303).
- [71] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 306).
- [72] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 308).
- [73] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 306).
- [74] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: <https://www.researchgate.net/publication/255673967> (besucht am 2025-09-05) (siehe S. 308).
- [75] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 303).
- [76] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 306, 308).

References IX



- [77] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Egle, Daniel S. Katz, Tom J. Pollard, Alexander Kononov, 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. 304).
- [78] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 306).
- [79] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/3IEC9899:2017 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. 303).
- [80] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 303).
- [81] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 305).
- [82] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021)*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 306).
- [83] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 303).
- [84] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (siehe S. 308).

References X



- [85] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: **978-1-4503-9475-8**. doi:[10.1145/3551349.3559497](https://doi.org/10.1145/3551349.3559497) (siehe S. 305).
- [86] Winfried Seimert. *LibreOffice 7.3 – Praxiswissen für Ein- und Umsteiger*. Blaufelden, Schwäbisch Hall, Baden-Württemberg, Germany: mitp Verlags GmbH & Co. KG, Apr. 2022. ISBN: **978-3-7475-0504-5** (siehe S. 304, 305).
- [87] Ellen Siever, Stephen Figgins, Robert Love und Arnold Robbins. *Linux in a Nutshell*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2009. ISBN: **978-0-596-15448-6** (siehe S. 305).
- [88] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: **978-1-0981-3391-7** (siehe S. 304).
- [89] Eric V. „[ericvsmith](https://ericvsmith.com/)“ Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), 6. Nov. 2016–9. Sep. 2023. URL: <https://peps.python.org/pep-0498> (besucht am 2024-07-25) (siehe S. 304).
- [90] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: **0001-0782**. doi:[10.1145/361020.361025](https://doi.org/10.1145/361020.361025) (siehe S. 306).
- [91] Sphinx Developers. "Doc Comments and Docstrings". In: [sphinx.ext.autodoc](https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#doc-comments-and-docstrings) – *Include Documentation from Docstrings*. 13. Okt. 2024. URL: <https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#doc-comments-and-docstrings> (besucht am 2024-12-12) (siehe S. 56–79, 154, 155).
- [92] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-commands.html> (besucht am 2025-02-25) (siehe S. 307).
- [93] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: **978-0-672-32451-2** (siehe S. 300, 307).

References XI



- [94] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of ⁹³ (siehe S. 307).
- [95] Michael J. Sullivan und Ivan Levkivskiy. *Adding a Final Qualifier to typing*. Python Enhancement Proposal (PEP) 591. Beaverton, OR, USA: Python Software Foundation (PSF), 15. März 2019. URL: <https://peps.python.org/pep-0591> (besucht am 2024-11-19) (siehe S. 138–147, 150–153, 156–158, 248–254).
- [96] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 306, 307).
- [97] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 306).
- [98] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/tutorial> (besucht am 2025-04-26).
- [99] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: <https://www.researchgate.net/publication/220094077> (besucht am 2024-10-01) (siehe S. 308).
- [100] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (siehe S. 305).
- [101] 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. 304, 308).
- [102] Adam Turner, Bénédict Tran, Chris Sewell, François Freitag, Jakob Lykke Andersen, Jean-François B., Stephen Finucane, Takayuki Shimizukawa, Takeshi Komiya und Sphinx Developers. *Sphinx – Create Intelligent and Beautiful Documentation with Ease*. 13. Okt. 2024. URL: <https://www.sphinx-doc.org> (besucht am 2024-12-12) (siehe S. 307).

References XII



- [103] Laurie A. Ulrich und Ken Cook. *Access For Dummies*. Hoboken, NJ, USA: For Dummies (Wiley), Dez. 2021. ISBN: 978-1-119-82908-9 (siehe S. 305).
- [104] Bruce M. Van Horn II und Quan Nguyen. *Hands-On Application Development with PyCharm*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-235-0 (siehe S. 306).
- [105] 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. 307).
- [106] Guido van Rossum, Barry Warsaw und Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), 5. Juli 2001. URL: <https://peps.python.org/pep-0008> (besucht am 2024-07-27) (siehe S. 56–66, 303).
- [107] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 305).
- [108] Thomas Weise (汤卫思). *Databases*. 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), 2025. URL: <https://thomasweise.github.io/databases> (besucht am 2025-01-05) (siehe S. 303, 304, 306).
- [109] 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. 306).
- [110] *What does PDF mean?* San Jose, CA, USA: Adobe Systems Incorporated, 2024. URL: <https://www.adobe.com/acrobat/about-adobe-pdf.html> (besucht am 2024-12-12) (siehe S. 28–36, 306).
- [111] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (besucht am 2025-01-05) (siehe S. 306).
- [112] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 306).

References XIII



- [113] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 306).
- [114] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 306).
- [115] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: <https://www.techtarget.com/searchdatamanagement/definition/database-management-system> (besucht am 2025-01-11) (siehe S. 303).
- [116] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 303).

Glossary (in English) I



API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another³⁹.

Bash is a the shell used under Ubuntu Linux, i.e., the program that „runs“ in the terminal and interprets your commands, allowing you to start and interact with other programs^{13,70,116}. Learn more at <https://www.gnu.org/software/bash>.

C is a programming language, which is very successful in system programming situations^{27,79}.

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples for clients are web browsers in the internet as well as clients for database management systems (DBMSes), such as `psql`.

client-server architecture is a system design where a central server receives requests from one or multiple clients^{8,59,75,80,83}. These requests and responses are usually sent over network connections. A typical example for such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as `psql`, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*¹⁰⁸.

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB¹¹⁵.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions³⁷. They must be delimited by `"""..."""`^{37,106}.

Glossary (in English) II



doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by These snippets can be executed by modules like `doctest`²⁸ or tools such as `pytest`⁵⁰. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

f-string let you include the results of expressions in strings^{14,34,36,40,65,89}. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

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^{88,101}. Learn more at <https://git-scm.com>.

GitHub is a website where software projects can be hosted and managed via the Git VCS^{77,101}. Learn more at <https://github.com>.

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

IT information technology

Java is another very successful programming language, with roots in the C family of languages^{9,60}.

LAMP Stack A system setup for web applications: Linux, Apache (a webserver), MySQL, and the server-side scripting language PHP^{16,45}.

LibreOffice is on open source office suite^{35,58,86} which is a good and free alternative to Microsoft Office. It offers software such as LibreOffice Calc and LibreOffice Base. See¹⁰⁸ for more information and installation instructions.

Glossary (in English) III



- LibreOffice Base** is a DBMSes that can work on stand-alone files but also connect to other popular relational databases^{31,86}. It is part of LibreOffice^{35,58,86} and has functionality that is comparable to Microsoft Access^{6,18,103}.
- LibreOffice Calc** is a spreadsheet software that allows you to arrange and perform calculations with data in a tabular grid. It is a free and open source spread sheet software^{58,86}, i.e., an alternative to Microsoft Excel. It is part of LibreOffice^{35,58,86}.
- linter** A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles^{49,85}. Ruff is an example for a linter used in the Python world.
- Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows^{3,44,87,100,107}. We recommend using it for this course, for software development, and for research. Learn more at <https://www.linux.org>. Its variant Ubuntu is particularly easy to use and install.
- MariaDB** An open source relational database management system that has forked off from MySQL^{1,2,4,30,62,81}. See <https://mariadb.org> for more information.
- Microsoft Access** is a DBMSes that can work on DBs stored in single, stand-alone files but also connect to other popular relational databases^{6,18,66,103}. It is part of Microsoft Office. A free and open source alternative to this commercial software is LibreOffice Base.
- Microsoft Excel** is a spreadsheet program that allows users to store, organize, manipulate, and calculate data in tabular structures^{10,41,53}. It is part of Microsoft Office. A free alternative to this commercial software is LibreOffice Calc^{58,86}.
- Microsoft Office** is a commercial suite of office software, including Microsoft Excel, Microsoft Word, and Microsoft Access⁵³. LibreOffice is a free and open source alternative.
- Microsoft Windows** is a commercial proprietary operating system¹². It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at <https://www.microsoft.com/windows>.
- Microsoft Word** is one of the leading text writing programs^{29,69} and part of Microsoft Office. A free alternative to this commercial software is the LibreOffice Writer.

Glossary (in English) IV



Mypy is a static type checking tool for Python⁵⁶ that makes use of type hints. Learn more at <https://github.com/python/mypy> and in¹⁰⁹.

MySQL An open source relational database management system^{11,30,82,97,112}. MySQL is famous for its use in the LAMP Stack. See <https://www.mysql.com> for more information.

OSS Open source software, i.e., software that can freely be used, whose source code is made available in the internet, and which is usually developed cooperatively over the internet as well⁴⁶. Typical examples are Python, Linux, Git, and PostgreSQL.

PDF The *Portable Document Format*^{33,110} is the format in which provide this book. It is the standard format for the exchange of documents in the internet.

PostgreSQL An open source object-relational DBMS^{32,71,78,97}. See <https://postgresql.org> for more information.
psql is the client program used to access the PostgreSQL DBMS server.

PyCharm is the convenient Python Integrated Development Environment (IDE) that we recommend for this course^{104,113,114}. It comes in a free community edition, so it can be downloaded and used at no cost. Learn more at <https://www.jetbrains.com/pycharm>.

pytest is a framework for writing and executing unit tests in Python^{26,51,73,76,113}. Learn more at <https://pytest.org>.

Python The Python programming language^{47,55,61,109}, i.e., what you will learn about in our book¹⁰⁹. Learn more at <https://python.org>.

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other^{21,42,43,90,96,108,111}.

Ruff is a linter and code formatting tool for Python^{63,64}. Learn more at <https://docs.astral.sh/ruff> or in¹⁰⁹.

Glossary (in English) V



server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carrying the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples for servers are web servers¹⁶ in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“⁵².

signature The signature of a function refers to the parameters and their types, the return type, and the exceptions that the function can raise⁶⁷. In Python, the function `signature` of the module `inspect` provides some information about the signature of a function¹⁵.

Sphinx Sphinx is a tool for generating software documentation¹⁰². It supports Python can use both docstrings and type hints to generate beautiful documents. Learn more at <https://www.sphinx-doc.org>.

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases^{17,22–24,48,68,92–94,96}. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference⁹².

(string) interpolation In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`.

terminal A terminal is a text-based window where you can enter commands and execute them^{3,20}. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf `Win` + `R`, dann Schreiben von `cmd`, dann Druck auf `↵`. Under Ubuntu Linux, `Ctrl` + `Alt` + `T` opens a terminal, which then runs a Bash shell inside.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be^{54,105}. 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.

Glossary (in English) VI



Ubuntu is a variant of the open source operating system Linux^{20,45}. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at <https://ubuntu.com>. If you are in China, you can download it from <https://mirrors.ustc.edu.cn/ubuntu-releases>.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification^{5,72,74,76,84,99}. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code¹⁰¹. 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.

WWW World Wide Web^{7,25}

x -axis The x -axis is the horizontal axis of a two-dimensional coordinate system, often referred to abscissa.