合肥大學
HEFEI UNIVERSITY

# Programming with Python

# 43. Klassen/Dunder: `__str__`, `__repr__`, und `__eq__`

Thomas Weise (汤卫思)
tweise@hfuu.edu.cn

Institute of Applied Optimization (IAO) | 应用优化研究所
School of Artificial Intelligence and Big Data | 人工智能与大数据学院
Hefei University | 合肥大学
Hefei, Anhui, China | 中国安徽省合肥市

Version: 2025-09-29

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

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

# Outline

# Einleitung

# Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt“.

## Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt“.
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.

## Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt".
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen

# Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt".
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.

# Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt".
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.

## Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt".
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.
- Wir können über die Elemente von Objekten iterieren, die Sequenzen darstellen.

# Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt“.
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.
- Wir können über die Elemente von Objekten iterieren, die Sequenzen darstellen.
- Wir können Objekte ausführen, die Funktionen repräsentieren.

# Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt“.
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.
- Wir können über die Elemente von Objekten iterieren, die Sequenzen darstellen.
- Wir können Objekte ausführen, die Funktionen repräsentieren.
- Diese besonderen Funktionalitäten sind als so genannte *dunder*-Methoden implementiert.

## Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt“.
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.
- Wir können über die Elemente von Objekten iterieren, die Sequenzen darstellen.
- Wir können Objekte ausführen, die Funktionen repräsentieren.
- Diese besonderen Funktionalitäten sind als so genannte *dunder*-Methoden implementiert.
- *Dunder* steht fpr „double underscore“, also „Doppelter Unterstrich“, also `__`.

## Einleitung

- In Python, *everything is an object*[16,24], also „Alles ist ein Objekt".
- Funktionen, Module, Klassen, Datentypen, Werte einfacher Datentypen, und so weiter – alles sind Objekte.
- Viele dieser Objekte haben besondere Funktionen
- Zum Beispiel können wir numerische Objekte addieren, subtrahieren, multiplizieren und dividieren.
- Wir können die String-Repräsentation von allen Objekten bekommen und auf der Konsole ausgeben.
- Wir können über die Elemente von Objekten iterieren, die Sequenzen darstellen.
- Wir können Objekte ausführen, die Funktionen repräsentieren.
- Diese besonderen Funktionalitäten sind als so genannte *dunder*-Methoden implementiert.
- *Dunder* steht fpr „double underscore", also „Doppelter Unterstrich", also `__`.
- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.

# Dunder Methoden

- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.

# Dunder Methoden

- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.
- Ein typisches Beispiel haben wir schon kennengelernt, nämlich den Initialisierer `__init__`, der die Attribute von Objekten erstellt.

# Dunder Methoden

- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.
- Ein typisches Beispiel haben wir schon kennengelernt, nämlich den Initialisierer `__init__`, der die Attribute von Objekten erstellt.
- Wir haben auch gelernt das, wenn wir eine Subklasse von einer Klasse ableiten, sowohl neue Methoden definieren als auch existierende Methoden überschreiben können.

# Dunder Methoden

- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.
- Ein typisches Beispiel haben wir schon kennengelernt, nämlich den Initialisierer `__init__`, der die Attribute von Objekten erstellt.
- Wir haben auch gelernt das, wenn wir eine Subklasse von einer Klasse ableiten, sowohl neue Methoden definieren als auch existierende Methoden überschreiben können.
- Das können wir auch mit den Dunder-Methoden machen.

# Dunder Methoden

- Alle Namen solcher Methoden beginnen und enden nämlich mit `__`.
- Ein typisches Beispiel haben wir schon kennengelernt, nämlich den Initialisierer `__init__`, der die Attribute von Objekten erstellt.
- Wir haben auch gelernt das, wenn wir eine Subklasse von einer Klasse ableiten, sowohl neue Methoden definieren als auch existierende Methoden überschreiben können.
- Das können wir auch mit den Dunder-Methoden machen.
- Das heist, das wir im Grunde alle der vorher genannten Funktionalitäten erzeugen, verändern, und anpassen können!

# __str__ und __repr__

## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden.

## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden
  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3].

# `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer.

## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden
  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde.

## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde. Sonst ruft es `o.__repr__()` auf.

# `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde. Sonst ruft es `o.__repr__()` auf.
  2. `repr(o)` sollte idealerweise eine String-Repräsentation liefern, die alle Informationen beinhaltet, die notwendig sind, um das Objekt `o` erneut zu erzeugen[2].

# `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde. Sonst ruft es `o.__repr__()` auf.
  2. `repr(o)` sollte idealerweise eine String-Repräsentation liefern, die alle Informationen beinhaltet, die notwendig sind, um das Objekt `o` erneut zu erzeugen[2]. Das Zielpublikum sind hier Programmier, die am Kode arbeiten, und die z. B. genau Informationen in Log-Dateien schreiben oder nach Fehlern suchen müssen.

# `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

    1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde. Sonst ruft es `o.__repr__()` auf.
    2. `repr(o)` sollte idealerweise eine String-Repräsentation liefern, die alle Informationen beinhaltet, die notwendig sind, um das Objekt `o` erneut zu erzeugen[2]. Das Zielpublikum sind hier Programmier, die am Kode arbeiten, und die z. B. genau Informationen in Log-Dateien schreiben oder nach Fehlern suchen müssen. `repr(o)` ruft die Dunder-Methode `__repr__` auf, wenn diese implementiert wurde.

## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts `o` unterscheiden

  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts `o` liefern[3]. Diese Repräsentation ist hauptsächlich für Endbenutzer. `str(o)` ruft die Dunder-Methode `__str__` von `o` auf, wenn diese implementiert wurde. Sonst ruft es `o.__repr__()` auf.
  2. `repr(o)` sollte idealerweise eine String-Repräsentation liefern, die alle Informationen beinhaltet, die notwendig sind, um das Objekt `o` erneut zu erzeugen[2]. Das Zielpublikum sind hier Programmier, die am Kode arbeiten, und die z. B. genau Informationen in Log-Dateien schreiben oder nach Fehlern suchen müssen. `repr(o)` ruft die Dunder-Methode `__repr__` auf, wenn diese implementiert wurde. Sonst wird er name des Datentyps und die ID des Objekts geliefert.

# Beispiel: str und repr

- In Programm str_vs_repr.py vergleichen wir die beiden Funktionen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- In Programm `str_vs_repr.py` vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variable `the_int` mit Wert `123`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- In Programm str_vs_repr.py vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variable `the_int` mit Wert `123`.
- Sowohl `str(the_int)` als auch `repr(the_int)` liefern `"123"`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- In Programm `str_vs_repr.py` vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variable `the_int` mit Wert `123`.
- Sowohl `str(the_int)` als auch `repr(the_int)` liefern `"123"`.
- Das erwarten wir so auch, denn das ist sowohl die kompakteste Variante, den Wert darzustellen, als auch alle Information, die man braucht, um ihn neu zu erzeugen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- In Programm `str_vs_repr.py` vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variable `the_int` mit Wert `123`.
- Sowohl `str(the_int)` als auch `repr(the_int)` liefern `"123"`.
- Das erwarten wir so auch, denn das ist sowohl die kompakteste Variante, den Wert darzustellen, als auch alle Information, die man braucht, um ihn neu zu erzeugen.
- Wir erstellen eine Variable `the_str` mit Wert `"123"`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Sowohl `str(the_int)` als auch `repr(the_int)` liefern `"123"`.
- Das erwarten wir so auch, denn das ist sowohl die kompakteste Variante, den Wert darzustellen, als auch alle Information, die man braucht, um ihn neu zu erzeugen.
- Wir erstellen eine Variable `the_str` mit Wert `"123"`.
- Wenn wir `the_str` auf dem standard output stream (`stdout`) ausgeben, wenn wir also `print(str(the_str))` machen, dann taucht der Text `123` auf der Konsole auf.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Das erwarten wir so auch, denn das ist sowohl die kompakteste Variante, den Wert darzustellen, als auch alle Information, die man braucht, um ihn neu zu erzeugen.
- Wir erstellen eine Variable `the_str` mit Wert `"123"`.
- Wenn wir `the_str` auf dem standard output stream (`stdout`) ausgeben, wenn wir also `print(str(the_str))` machen, dann taucht der Text `123` auf der Konsole auf.
- Drucken wir dagegen `repr(the_str)` aus, dann erscheint `'123'`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
    ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
          ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wir erstellen eine Variable `the_str` mit Wert `"123"`.
- Wenn wir `the_str` auf dem standard output stream (`stdout`) ausgeben, wenn wir also `print(str(the_str))` machen, dann taucht der Text `123` auf der Konsole auf.
- Drucken wir dagegen `repr(the_str)` aus, dann erscheint `'123'`.
- Beachten Sie die einzelnen Anführungszeichen an beiden Enden?

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Wir erstellen eine Variable `the_str` mit Wert `"123"`.
- Wenn wir `the_str` auf dem standard output stream (`stdout`) ausgeben, wenn wir also `print(str(the_str))` machen, dann taucht der Text `123` auf der Konsole auf.
- Drucken wir dagegen `repr(the_str)` aus, dann erscheint `'123'`.
- Beachten Sie die einzelnen Anführungszeichen an beiden Enden?
- Diese sind notwendig.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Wenn wir `the_str` auf dem standard output stream (stdout) ausgeben, wenn wir also `print(str(the_str))` machen, dann taucht der Text `123` auf der Konsole auf.
- Drucken wir dagegen `repr(the_str)` aus, dann erscheint `'123'`.
- Beachten Sie die einzelnen Anführungszeichen an beiden Enden?
- Diese sind notwendig.
- Ohne sie wären `repr(the_str)` und `repr(the_int)` gleich.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Drucken wir dagegen `repr(the_str)` aus, dann erscheint `'123'`.
- Beachten Sie die einzelnen Anführungszeichen an beiden Enden?
- Diese sind notwendig.
- Ohne sie wären `repr(the_str)` und `repr(the_int)` gleich.
- Wir könnten nicht unterscheiden, ob ein String oder eine Ganzzahl ausgegeben wurde.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Beachten Sie die einzelnen Anführungszeichen an beiden Enden?
- Diese sind notwendig.
- Ohne sie wären `repr(the_str)` und `repr(the_int)` gleich.
- Wir könnten nicht unterscheiden, ob ein String oder eine Ganzzahl ausgegeben wurde.
- Das spielt natürlich nur eine Rolle, wenn wir uns für die innere Arbeitsweise unseres Programms interessieren.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Diese sind notwendig.
- Ohne sie wären `repr(the_str)` und `repr(the_int)` gleich.
- Wir könnten nicht unterscheiden, ob ein String oder eine Ganzzahl ausgegeben wurde.
- Das spielt natürlich nur eine Rolle, wenn wir uns für die innere Arbeitsweise unseres Programms interessieren.
- Und genau dafür existiert `repr`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Ohne sie wären `repr(the_str)` und `repr(the_int)` gleich.
- Wir könnten nicht unterscheiden, ob ein String oder eine Ganzzahl ausgegeben wurde.
- Das spielt natürlich nur eine Rolle, wenn wir uns für die innere Arbeitsweise unseres Programms interessieren.
- Und genau dafür existiert `repr`.
- Nun erstellen wir zwei Kollektionen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
       ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wir könnten nicht unterscheiden, ob ein String oder eine Ganzzahl ausgegeben wurde.
- Das spielt natürlich nur eine Rolle, wenn wir uns für die innere Arbeitsweise unseres Programms interessieren.
- Und genau dafür existiert `repr`.
- Nun erstellen wir zwei Kollektionen.
- Zuerst erzeugen wir die Liste `l1`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now   =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now   =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Das spielt natürlich nur eine Rolle, wenn wir uns für die innere Arbeitsweise unseres Programms interessieren.
- Und genau dafür existiert `repr`.
- Nun erstellen wir zwei Kollektionen.
- Zuerst erzeugen wir die Liste `l1`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `l2`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Und genau dafür existiert `repr`.
- Nun erstellen wir zwei Kollektionen.
- Zuerst erzeugen wir die Liste `l1`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `l2`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.
- Dann wenden wir geben wir beide Listen aus wobei intern `str(l1)` und `str(l2)` werden.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"      right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"      right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Nun erstellen wir zwei Kollektionen.
- Zuerst erzeugen wir die Liste `l1`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `l2`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.
- Dann wenden wir geben wir beide Listen aus wobei intern `str(l1)` und `str(l2)` werden.
- Das Ergebnis von `print(f"{l1 = }, but {l2 = }")` ist `l1 = [1, 2, 3],` `but l2 = ['1', '2', '3']`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now   =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now   =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Zuerst erzeugen wir die Liste `l1`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `l2`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.
- Dann wenden wir geben wir beide Listen aus wobei intern `str(l1)` und `str(l2)` werden.
- Das Ergebnis von `print(f"{l1 = }, but {l2 = }")` ist `l1 = [1, 2, 3],` `but l2 = ['1', '2', '3']`.
- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von `l2`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Dann wenden wir geben wir beide Listen aus wobei intern `str(l1)` und `str(l2)` werden.
- Das Ergebnis von `print(f"{l1 = }, but {l2 = }")` ist `l1 = [1, 2, 3],` `but l2 = ['1', '2', '3']`.
- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von `l2`.
- Wenn die textuelle Repräsentation der Standard-Kollektionstypen bon Python mit `str` oder `repr` erzeugt wird, dann weden die Elemente der Kollektionen immer `repr` zu Stringskonvertiert, nie mit `str`[6].

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"      right_now   =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"      right_now   =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Das Ergebnis von `print(f"{l1 = }, but {l2 = }")` ist `l1 = [1, 2, 3],` `but l2 = ['1', '2', '3']`.
- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von `l2`.
- Wenn die textuelle Repräsentation der Standard-Kollektionstypen bon Python mit `str` oder `repr` erzeugt wird, dann weden die Elemente der Kollektionen immer `repr` zu Stringskonvertiert, nie mit `str`[6].
- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
    ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von `l2`.
- Wenn die textuelle Repräsentation der Standard-Kollektionstypen bon Python mit `str` oder `repr` erzeugt wird, dann weden die Elemente der Kollektionen immer `repr` zu Stringskonvertiert, nie mit `str`[6].
- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.
- Ein anderes gutes Beispiel für den Unterschied zwischen `str` und `repr` ist Python's Klasse `datetime`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ python3 str_vs_repr.py ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
       ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wenn die textuelle Repräsentation der Standard-Kollektionstypen bon Python mit `str` oder `repr` erzeugt wird, dann weden die Elemente der Kollektionen immer `repr` zu Stringskonvertiert, nie mit `str`[6].
- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.
- Ein anderes gutes Beispiel für den Unterschied zwischen `str` und `repr` ist Python's Klasse `datetime`.
- Wir diskutieren diese Klasse hier nicht im Detail.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now   =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now   =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.
- Ein anderes gutes Beispiel für den Unterschied zwischen `str` und `repr` ist Python's Klasse `datetime`.
- Wir diskutieren diese Klasse hier nicht im Detail.
- Es reicht zu wissen, dass Instanzen dieser Klasse eine Kombination aus Datum und Uhrzeit darstellen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.
- Ein anderes gutes Beispiel für den Unterschied zwischen `str` und `repr` ist Python's Klasse `datetime`.
- Wir diskutieren diese Klasse hier nicht im Detail.
- Es reicht zu wissen, dass Instanzen dieser Klasse eine Kombination aus Datum und Uhrzeit darstellen.
- Im Programm importieren wir erst die Klasse `datetime` aus dem Modul mit dem selben Namen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now   =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now   =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
       ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wir diskutieren diese Klasse hier nicht im Detail.
- Es reicht zu wissen, dass Instanzen dieser Klasse eine Kombination aus Datum und Uhrzeit darstellen.
- Im Programm importieren wir erst die Klasse `datetime` aus dem Modul mit dem selben Namen.
- Wir erstellen eine Variable `right_now`, die das Ergebnis der Funktion `datetime.now` zugewiesen bekommt, die ein Objekt zurückliefert, in dem das aktuelle Datum und die aktuelle Uhrzeit gespeichert sind.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Im Programm importieren wir erst die Klasse `datetime` aus dem Modul mit dem selben Namen.
- Wir erstellen eine Variable `right_now`, die das Ergebnis der Funktion `datetime.now` zugewiesen bekommt, die ein Objekt zurückliefert, in dem das aktuelle Datum und die aktuelle Uhrzeit gespeichert sind.
- Wenn wir das Ergebnis der Funktion `str` auf ein Objekt `o` in einem f-String ausgeben wollen, dann benutzen wir die Format-Spezifikation `!s`, schreiben also `f"{o!s}"`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"      right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"      right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Wir erstellen eine Variable `right_now`, die das Ergebnis der Funktion `datetime.now` zugewiesen bekommt, die ein Objekt zurückliefert, in dem das aktuelle Datum und die aktuelle Uhrzeit gespeichert sind.
- Wenn wir das Ergebnis der Funktion `str` auf ein Objekt `o` in einem f-String ausgeben wollen, dann benutzen wir die Format-Spezifikation `!s`, schreiben also `f"{o!s}"`.
- Wir sehen, dass die String-Repräsentation von einem `datetime`-Objekt ein einfacher, leicht lesbarer Datums- und Uhrzeit-String ist.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wenn wir das Ergebnis der Funktion `str` auf ein Objekt `o` in einem f-String ausgeben wollen, dann benutzen wir die Format-Spezifikation `!s`, schreiben also `f"{o!s}"`.
- Wir sehen, dass die String-Repräsentation von einem `datetime`-Objekt ein einfacher, leicht lesbarer Datums- und Uhrzeit-String ist.
- Das Ergebnis der Funktion `repr` für ein Objekt `o` in einem f-String wird mit der Format-Spezifikation `!r` abgefragt, also durch `f"{o!r}"`.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
    ↪ tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
          ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wir sehen, dass die String-Repräsentation von einem `datetime`-Objekt ein einfacher, leicht lesbarer Datums- und Uhrzeit-String ist.
- Das Ergebnis der Funktion `repr` für ein Objekt `o` in einem f-String wird mit der Format-Spezifikation `!r` abgefragt, also durch `f"{o!r}"`.
- Machen wir das mit einem `datetime`-Objekt, dann bekommen wir tatsächlich die Information, die wir brauchen, um das Objekt wieder zu erzeugen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Wir sehen, dass die String-Repräsentation von einem `datetime`-Objekt ein einfacher, leicht lesbarer Datums- und Uhrzeit-String ist.
- Das Ergebnis der Funktion `repr` für ein Objekt `o` in einem f-String wird mit der Format-Spezifikation `!r` abgefragt, also durch `f"{o!r}"`.
- Machen wir das mit einem `datetime`-Objekt, dann bekommen wir tatsächlich die Information, die wir brauchen, um das Objekt wieder zu erzeugen.
- Wir könnten die Ausgabe von `repr` direkt in die Python-Konsole kopieren!

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
       ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Das Ergebnis der Funktion `repr` für ein Objekt `o` in einem f-String wird mit der Format-Spezifikation `!r` abgefragt, also durch `f"{o!r}"`.
- Machen wir das mit einem `datetime`-Objekt, dann bekommen wir tatsächlich die Information, die wir brauchen, um das Objekt wieder zu erzeugen.
- Wir könnten die Ausgabe von `repr` direkt in die Python-Konsole kopieren!
- Das würde ein `datetime`-Objekt mit genau den selben Daten wie `right_now` erzeugen.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"      right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"      right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
      right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)'
      right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616, tzinfo=datetime.timezone.utc)
```

# Beispiel: `str` und `repr`

- Machen wir das mit einem `datetime`-Objekt, dann bekommen wir tatsächlich die Information, die wir brauchen, um das Objekt wieder zu erzeugen.
- Wir könnten die Ausgabe von `repr` direkt in die Python-Konsole kopieren!
- Das würde ein `datetime`-Objekt mit genau den selben Daten wie `right_now` erzeugen.
- Das würde auch mit den String-Repräsentationen der beiden Listen `l1` und `l2` oben funktionieren.

```python
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2025-09-29 22:08:58.423616+00:00'
     right_now  =  2025-09-29 22:08:58.423616+00:00
repr(right_now) = 'datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
   ↪ tzinfo=datetime.timezone.utc)'
     right_now  =  datetime.datetime(2025, 9, 29, 22, 8, 58, 423616,
        ↪ tzinfo=datetime.timezone.utc)
```

# Strings und Gleichheit

# Beispiel: `Point` und Gleichheit

- Gehen wir noch ein paar Schritte zurück zu einem Beispiel, das wir selbst erstellt haben.

```python
"""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: `Point` und Gleichheit

- Gehen wir noch ein paar Schritte zurück zu einem Beispiel, das wir selbst erstellt haben.
- Wir erinnern uns an die Klasse `Point`, mit der wir Punkte in der zweidimensionalen Euklidischen Ebene dargestellt haben.

```python
"""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: `Point` und Gleichheit

- Gehen wir noch ein paar Schritte zurück zu einem Beispiel, das wir selbst erstellt haben.
- Wir erinnern uns an die Klasse `Point`, mit der wir Punkte in der zweidimensionalen Euklidischen Ebene dargestellt haben.
- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert haben.

```python
"""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: `Point` und Gleichheit

- Gehen wir noch ein paar Schritte zurück zu einem Beispiel, das wir selbst erstellt haben.
- Wir erinnern uns an die Klasse `Point`, mit der wir Punkte in der zweidimensionalen Euklidischen Ebene dargestellt haben.
- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert haben.
- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.

```python
"""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: `Point` und Gleichheit

- Gehen wir noch ein paar Schritte zurück zu einem Beispiel, das wir selbst erstellt haben.
- Wir erinnern uns an die Klasse `Point`, mit der wir Punkte in der zweidimensionalen Euklidischen Ebene dargestellt haben.
- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert haben.
- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.
- Spielen wir mit der Klasse etwas mehr.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Wir erinnern uns an die Klasse `Point`, mit der wir Punkte in der zweidimensionalen Euklidischen Ebene dargestellt haben.
- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert haben.
- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.
- Spielen wir mit der Klasse etwas mehr.
- Im Programm `point_user_2.py` erstellen wir drei Instanzen dieser Klasse.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert haben.
- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.
- Spielen wir mit der Klasse etwas mehr.
- Im Programm `point_user_2.py` erstellen wir drei Instanzen dieser Klasse.
- `p1` steht für die Koordinaten $(3, 5)$, `p2` speichert $(7, 8)$ und `p3` hat die selben Koordinaten wie `p1`.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.
- Spielen wir mit der Klasse etwas mehr.
- Im Programm `point_user_2.py` erstellen wir drei Instanzen dieser Klasse.
- `p1` steht für die Koordinaten $(3, 5)$, `p2` speichert $(7, 8)$ und `p3` hat die selben Koordinaten wie `p1`.
- In diesem Program geben wir erstmal die Ergebnisse von `str` und `repr` für `p1` aus.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Spielen wir mit der Klasse etwas mehr.
- Im Programm `point_user_2.py` erstellen wir drei Instanzen dieser Klasse.
- `p1` steht für die Koordinaten $(3, 5)$, `p2` speichert $(7, 8)$ und `p3` hat die selben Koordinaten wie `p1`.
- In diesem Program geben wir erstmal die Ergebnisse von `str` und `repr` für `p1` aus.
- Wir erkennen sofort, dass sie eher nutzlos sind.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Im Programm `point_user_2.py` erstellen wir drei Instanzen dieser Klasse.
- `p1` steht für die Koordinaten $(3, 5)$, `p2` speichert $(7, 8)$ und `p3` hat die selben Koordinaten wie `p1`.
- In diesem Program geben wir erstmal die Ergebnisse von `str` und `repr` für `p1` aus.
- Wir erkennen sofort, dass sie eher nutzlos sind.
- Da wir weder `__str__` noch `__repr__` implementiert haben, greift `str` auf `__repr__` zurück, welches dann einfach den Typename und die Objekt-ID liefert.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- `p1` steht für die Koordinaten $(3, 5)$, `p2` speichert $(7, 8)$ und `p3` hat die selben Koordinaten wie `p1`.
- In diesem Program geben wir erstmal die Ergebnisse von `str` und `repr` für `p1` aus.
- Wir erkennen sofort, dass sie eher nutzlos sind.
- Da wir weder `__str__` noch `__repr__` implementiert haben, greift `str` auf `__repr__` zurück, welches dann einfach den Typename und die Objekt-ID liefert.
- Für uns sind das keine nützlichen Informationen.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- In diesem Program geben wir erstmal die Ergebnisse von `str` und `repr` für `p1` aus.
- Wir erkennen sofort, dass sie eher nutzlos sind.
- Da wir weder `__str__` noch `__repr__` implementiert haben, greift `str` auf `__repr__` zurück, welches dann einfach den Typename und die Objekt-ID liefert.
- Für uns sind das keine nützlichen Informationen.
- Wenn wir gerade bei „nicht nützlich“ sind, da ist noch ein anderer Aspekt unserer Klasse `Point`, der sich nicht nützlich verhält.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Da wir weder `__str__` noch `__repr__` implementiert haben, greift `str` auf `__repr__` zurück, welches dann einfach den Typename und die Objekt-ID liefert.
- Für uns sind das keine nützlichen Informationen.
- Wenn wir gerade bei „nicht nützlich“ sind, da ist noch ein anderer Aspekt unserer Klasse `Point`, der sich nicht nützlich verhält.
- In Einheit 16 hatten wir den Unterschied zwischen der Gleichheit und der Identität von Objekten diskutiert.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Für uns sind das keine nützlichen Informationen.
- Wenn wir gerade bei „nicht nützlich“ sind, da ist noch ein anderer Aspekt unserer Klasse `Point`, der sich nicht nützlich verhält.
- In Einheit 16 hatten wir den Unterschied zwischen der Gleichheit und der Identität von Objekten diskutiert.
- Alle drei Variablen `p1`, `p2` und `p3` zeigen auf verschiedene Objekte.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Für uns sind das keine nützlichen Informationen.
- Wenn wir gerade bei „nicht nützlich“ sind, da ist noch ein anderer Aspekt unserer Klasse `Point`, der sich nicht nützlich verhält.
- In Einheit 16 hatten wir den Unterschied zwischen der Gleichheit und der Identität von Objekten diskutiert.
- Alle drei Variablen `p1`, `p2` und `p3` zeigen auf verschiedene Objekte.
- Während `p1 is p1` offensichtlich `True` ist, müssen `p1 is p2` und `p1 is p3` offensichtlich `False` sein.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Wenn wir gerade bei „nicht nützlich“ sind, da ist noch ein anderer Aspekt unserer Klasse `Point`, der sich nicht nützlich verhält.
- In Einheit 16 hatten wir den Unterschied zwischen der Gleichheit und der Identität von Objekten diskutiert.
- Alle drei Variablen `p1`, `p2` und `p3` zeigen auf verschiedene Objekte.
- Während `p1 is p1` offensichtlich `True` ist, müssen `p1 is p2` und `p1 is p3` offensichtlich `False` sein.
- Die drei Objekte sind alle verschiedene Instanzen von `Point`, also erwarten wir das genau so.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- In Einheit 16 hatten wir den Unterschied zwischen der Gleichheit und der Identität von Objekten diskutiert.
- Alle drei Variablen `p1`, `p2` und `p3` zeigen auf verschiedene Objekte.
- Während `p1 is p1` offensichtlich `True` ist, müssen `p1 is p2` und `p1 is p3` offensichtlich `False` sein.
- Die drei Objekte sind alle verschiedene Instanzen von `Point`, also erwarten wir das genau so.
- Es ist aber ärgerlich, dass `p1 == p3` auch `False` ergibt.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Alle drei Variablen `p1`, `p2` und `p3` zeigen auf verschiedene Objekte.
- Während `p1 is p1` offensichtlich `True` ist, müssen `p1 is p2` und `p1 is p3` offensichtlich `False` sein.
- Die drei Objekte sind alle verschiedene Instanzen von `Point`, also erwarten wir das genau so.
- Es ist aber ärgerlich, dass `p1 == p3` auch `False` ergibt.
- `p1 == p2` soll `False` sein, weil diese beiden Punkte unterschiedlich sind.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Während `p1 is p1` offensichtlich `True` ist, müssen `p1 is p2` und `p1 is p3` offensichtlich `False` sein.
- Die drei Objekte sind alle verschiedene Instanzen von `Point`, also erwarten wir das genau so.
- Es ist aber ärgerlich, dass `p1 == p3` auch `False` ergibt.
- `p1 == p2` soll `False` sein, weil diese beiden Punkte unterschiedlich sind.
- Die beiden Punkte `p1` und `p3` haben aber die selben Koordinaten.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Die drei Objekte sind alle verschiedene Instanzen von `Point`, also erwarten wir das genau so.
- Es ist aber ärgerlich, dass `p1 == p3` auch `False` ergibt.
- `p1 == p2` soll `False` sein, weil diese beiden Punkte unterschiedlich sind.
- Die beiden Punkte `p1` und `p3` haben aber die selben Koordinaten.
- Sie sollten als gleich betrachtet werden.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Es ist aber ärgerlich, dass `p1 == p3` auch `False` ergibt.
- `p1 == p2` soll `False` sein, weil diese beiden Punkte unterschiedlich sind.
- Die beiden Punkte `p1` und `p3` haben aber die selben Koordinaten.
- Sie sollten als gleich betrachtet werden.
- Andersherum ist `p1 != p2` `True`, so wie es seien soll, aber `p1 != p3` sollte eigentlich `False` seien, ist aber `True`.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- `p1 == p2` soll `False` sein, weil diese beiden Punkte unterschiedlich sind.
- Die beiden Punkte `p1` und `p3` haben aber die selben Koordinaten.
- Sie sollten als gleich betrachtet werden.
- Andersherum ist `p1 != p2` `True`, so wie es seien soll, aber `p1 != p3` sollte eigentlich `False` seien, ist aber `True`.
- Der Grund dafür ist das Python nicht wissen kann, wann und warum Instanzen unserer eigenen Klasse als gleich betrachtet werden sollen.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Die beiden Punkte `p1` und `p3` haben aber die selben Koordinaten.
- Sie sollten als gleich betrachtet werden.
- Andersherum ist `p1 != p2` `True`, so wie es seien soll, aber `p1 != p3` sollte eigentlich `False` seien, ist aber `True`.
- Der Grund dafür ist das Python nicht wissen kann, wann und warum Instanzen unserer eigenen Klasse als gleich betrachtet werden sollen.
- Es nimmt einfach an, dass Gleichheit = Identity, also ein Objekt nur gleich zu sich selbst ist.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Sie sollten als gleich betrachtet werden.
- Andersherum ist `p1 != p2` `True`, so wie es seien soll, aber `p1 != p3` sollte eigentlich `False` seien, ist aber `True`.
- Der Grund dafür ist das Python nicht wissen kann, wann und warum Instanzen unserer eigenen Klasse als gleich betrachtet werden sollen.
- Es nimmt einfach an, dass Gleichheit = Identity, also ein Objekt nur gleich zu sich selbst ist.
- Wir könnten das reparieren, in dem wir die Dunder-Methode `__eq__` selbst implementieren.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Der Grund dafür ist das Python nicht wissen kann, wann und warum Instanzen unserer eigenen Klasse als gleich betrachtet werden sollen.
- Es nimmt einfach an, dass Gleichheit = Identity, also ein Objekt nur gleich zu sich selbst ist.
- Wir könnten das reparieren, in dem wir die Dunder-Methode `__eq__` selbst implementieren.
- Diese Methode bekommt ein beliebiges Objekt `other` als Parameter und soll `True` bei Gleichheit zurückliefern und `False` sonst.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Es nimmt einfach an, dass Gleichheit = Identity, also ein Objekt nur gleich zu sich selbst ist.
- Wir könnten das reparieren, in dem wir die Dunder-Methode `__eq__` selbst implementieren.
- Diese Methode bekommt ein beliebiges Objekt `other` als Parameter und soll `True` bei Gleichheit zurückliefern und `False` sonst.
- Genau wie `str(o)` `o.__str__()` aufruft, so ruft der `a == b`-Operator `a.__eq__(b)` auf, wenn `a` `__eq__` implementiert.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Wir könnten das reparieren, in dem wir die Dunder-Methode `__eq__` selbst implementieren.
- Diese Methode bekommt ein beliebiges Objekt `other` als Parameter und soll `True` bei Gleichheit zurückliefern und `False` sonst.
- Genau wie `str(o)` `o.__str__()` aufruft, so ruft der `a == b`-Operator `a.__eq__(b)` auf, wenn `a` `__eq__` implementiert.
- Wenn Sie `__eq__` implementieren, dann wird Python annehmen das `(a != b)==` `not (a == b)`, also annehmen, dass zwei Objekte ungleich sind, wenn sie nicht gleich sind[32].

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Diese Methode bekommt ein beliebiges Objekt `other` als Parameter und soll `True` bei Gleichheit zurückliefern und `False` sonst.
- Genau wie `str(o)` `o.__str__()` aufruft, so ruft der `a == b`-Operator `a.__eq__(b)` auf, wenn `a` `__eq__` implementiert.
- Wenn Sie `__eq__` implementieren, dann wird Python annehmen das `(a != b)==` `not (a == b)`, also annehmen, dass zwei Objekte ungleich sind, wenn sie nicht gleich sind[32].
- Das ist muss nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht simmt[36] … vielleicht war es

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Genau wie `str(o)` `o.__str__()` aufruft, so ruft der `a == b`-Operator `a.__eq__(b)` auf, wenn `a` `__eq__` implementiert.
- Wenn Sie `__eq__` implementieren, dann wird Python annehmen das `(a != b)==` `not (a == b)`, also annehmen, dass zwei Objekte ungleich sind, wenn sie nicht gleich sind[32].
- Das ist muss nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht simmt[36] … vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte[35], im Moment sehe ich das aber nicht.
- Python erlaubt uns auch, die Dunder-Methode `__ne__` zu

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Wenn Sie `__eq__` implementieren, dann wird Python annehmen das `(a != b)==` `not (a == b)`, also annehmen, dass zwei Objekte ungleich sind, wenn sie nicht gleich sind[32].
- Das ist muss nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht simmt[36] . . . vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte[35], im Moment sehe ich das aber nicht.
- Python erlaubt uns auch, die Dunder-Methode `__ne__` zu implementieren, die von `a != b` als `a.__ne__(a)` aufgerufen wird.[32]
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Das ist muss nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht simmt[36] . . . vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte[35], im Moment sehe ich das aber nicht.
- Python erlaubt uns auch, die Dunder-Methode `__ne__` zu implementieren, die von `a != b` als `a.__ne__(a)` aufgerufen wird.[32]
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Das ist muss nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht simmt[36] . . . vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte[35], im Moment sehe ich das aber nicht.
- Python erlaubt uns auch, die Dunder-Methode `__ne__` zu implementieren, die von `a != b` als `a.__ne__(a)` aufgerufen wird.[32]
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.
- Und das tut es auch.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
      p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Python erlaubt uns auch, die Dunder-Methode `__ne__` zu implementieren, die von `a != b` als `a.__ne__(a)` aufgerufen wird.[32]
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.
- Und das tut es auch.
- Weil die beiden Objekte `p1` und `5` nicht identisch sind.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f"{str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_user_2.py ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.
- Und das tut es auch.
- Weil die beiden Objekte `p1` und `5` nicht identisch sind.
- Wie gesagt prüft der Standard-Gleichheits-Operator nur auf Identität.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` und Gleichheit

- Das sollte offensichtlich `False` ergeben.
- Und das tut es auch.
- Weil die beiden Objekte `p1` und `5` nicht identisch sind.
- Wie gesagt prüft der Standard-Gleichheits-Operator nur auf Identität.
- Falls wir `__eq__` selbst implementieren, dann muss die Methode `False` zurückliefern, wenn sie `5` als Argument bekommt (anstatt abzustürzen oder eine Ausnahme auszulösen. . . ).

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"      p1   =   {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"      p1   =   {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1  =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden

- Um die vorhin genannten Probleme alle zu lösen, implementieren wir die drei Dunder-Methoden `__str__`, `__repr__` und `__eq__` in unserer `Point`-Klasse in Datei `point_with_dunder.py`.

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

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


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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Um die vorhin genannten Probleme alle zu lösen, implementieren wir die drei Dunder-Methoden `__str__`, `__repr__` und `__eq__` in unserer `Point`-Klasse in Datei `point_with_dunder.py`.
- Die kurze String-Repräsentation, die `__str__` liefert, beinhaltet einfach die Punkt-Koordinaten, mit Komma getrennt, in Klammern.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Um die vorhin genannten Probleme alle zu lösen, implementieren wir die drei Dunder-Methoden `__str__`, `__repr__` und `__eq__` in unserer `Point`-Klasse in Datei `point_with_dunder.py`.
- Die kurze String-Repräsentation, die `__str__` liefert, beinhaltet einfach die Punkt-Koordinaten, mit Komma getrennt, in Klammern.
- Das liefert alle Informationen auf einen Blick, aber es könnte mit der String-Repräsentation eines Tupels verwechselt werden.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- Um die vorhin genannten Probleme alle zu lösen, implementieren wir die drei Dunder-Methoden `__str__`, `__repr__` und `__eq__` in unserer `Point`-Klasse in Datei `point_with_dunder.py`.
- Die kurze String-Repräsentation, die `__str__` liefert, beinhaltet einfach die Punkt-Koordinaten, mit Komma getrennt, in Klammern.
- Das liefert alle Informationen auf einen Blick, aber es könnte mit der String-Repräsentation eines Tupels verwechselt werden.
- Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- Die kurze String-Repräsentation, die `__str__` liefert, beinhaltet einfach die Punkt-Koordinaten, mit Komma getrennt, in Klammern.
- Das liefert alle Informationen auf einen Blick, aber es könnte mit der String-Repräsentation eines Tupels verwechselt werden.
- Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.
- Die Dunder-Methode `__eq__` prüft erst, ob das andere Objekt eine Instanz von `Point` ist.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Das liefert alle Informationen auf einen Blick, aber es könnte mit der String-Repräsentation eines Tupels verwechselt werden.
- Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.
- Die Dunder-Methode `__eq__` prüft erst, ob das andere Objekt eine Instanz von `Point` ist.
- Wenn ja, dann liefert sie `True` genau dann wenn die `x`- und `y`-Koordinate des Punkts `other` die gleichen sind wie die des Punkts `self`.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- Das liefert alle Informationen auf einen Blick, aber es könnte mit der String-Repräsentation eines Tupels verwechselt werden.
- Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.
- Die Dunder-Methode `__eq__` prüft erst, ob das andere Objekt eine Instanz von `Point` ist.
- Wenn ja, dann liefert sie `True` genau dann wenn die `x`- und `y`-Koordinate des Punkts `other` die gleichen sind wie die des Punkts `self`.
- Andernfalls liefert sie die Konstante `NotImplemented` zurück.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.
- Die Dunder-Methode `__eq__` prüft erst, ob das andere Objekt eine Instanz von `Point` ist.
- Wenn ja, dann liefert sie `True` genau dann wenn die `x`- und `y`-Koordinate des Punkts `other` die gleichen sind wie die des Punkts `self`.
- Andernfalls liefert sie die Konstante `NotImplemented` zurück.

> *A special value which should be returned by the binary special methods [...] to indicate that the operation is not implemented with respect to the other type...*
>
> *Note: When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.*
>
> — [8], 2001

# Beispiel: `Point` mit Dunder Methoden

- Die Dunder-Methode `__eq__` prüft erst, ob das andere Objekt eine Instanz von `Point` ist.
- Wenn ja, dann liefert sie `True` genau dann wenn die `x`- und `y`-Koordinate des Punkts `other` die gleichen sind wie die des Punkts `self`.
- Andernfalls liefert sie die Konstante `NotImplemented` zurück.
- In dem wir `NotImplemented` bei `other`-Objekten, die keine Instanzen von `Point` sind, zurückliefern, dann verweisen wir einfach auf das Standardverhalten des Operators `==`.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Wenn ja, dann liefert sie `True` genau dann wenn die `x`- und `y`-Koordinate des Punkts `other` die gleichen sind wie die des Punkts `self`.
- Andernfalls liefert sie die Konstante `NotImplemented` zurück.
- In dem wir `NotImplemented` bei `other`-Objekten, die keine Instanzen von `Point` sind, zurückliefern, dann verweisen wir einfach auf das Standardverhalten des Operators `==`.
- Wenn `other` keine Instanz von `Point`, dann gibt es keine Möglichkeit, auf Gleichheit zu vergleichen.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Andernfalls liefert sie die Konstante `NotImplemented` zurück.
- In dem wir `NotImplemented` bei `other`-Objekten, die keine Instanzen von `Point` sind, zurückliefern, dann verweisen wir einfach auf das Standardverhalten des Operators `==`.
- Wenn `other` keine Instanz von `Point`, dann gibt es keine Möglichkeit, auf Gleichheit zu vergleichen.
- Wir könnten in diesem Fall einfach `False` zurückliefern, was auch OK wäre.

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

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

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

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

    :return: `"Point(x, y)"`

    >>> repr(Point(2, 4))
    'Point(2, 4)'
    """
    return f"Point({self.x}, {self.y})"

def __str__(self) -> str:
    """
    Get a concise string representation useful for end users.

    :return: `"(x,y)"`

    >>> str(Point(2, 4))
    '(2,4)'
    """
    return f"({self.x},{self.y})"

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

    :param other: the other object
    :return: `True` if and only if `other` is also a `Point` and has
        the same coordinates; `NotImplemented` if it is not a point

    >>> Point(1, 2) == Point(2, 3)
    False
    >>> Point(1, 2) == Point(1, 2)
    True
    """
    return (other.x == self.x) and (other.y == self.y) \
        if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- In dem wir `NotImplemented` bei `other`-Objekten, die keine Instanzen von `Point` sind, zurückliefern, dann verweisen wir einfach auf das Standardverhalten des Operators `==`.
- Wenn `other` keine Instanz von `Point`, dann gibt es keine Möglichkeit, auf Gleichheit zu vergleichen.
- Wir könnten in diesem Fall einfach `False` zurückliefern, was auch OK wäre.
- `NotImplemented` zu liefern gibt uns das gleiche Ergebnis wenn wir mit Objekten eines anderen Typs vergleichen wie z. B. `5`.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methoden

- Wenn `other` keine Instanz von `Point`, dann gibt es keine Möglichkeit, auf Gleichheit zu vergleichen.
- Wir könnten in diesem Fall einfach `False` zurückliefern, was auch OK wäre.
- `NotImplemented` zu liefern gibt uns das gleiche Ergebnis wenn wir mit Objekten eines anderen Typs vergleichen wie z. B. `5`.
- Es ermöglicht aber anderen Programmieren, eine neue Klasse zu schreiben, die einen Gleichheitsvergleich mit unseren `Point`-Instanzen implementiert.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: `Point` mit Dunder Methode

- Wir könnten in diesem Fall einfach `False` zurückliefern, was auch OK wäre.
- `NotImplemented` zu liefern gibt uns das gleiche Ergebnis wenn wir mit Objekten eines anderen Typs vergleichen wie z. B. `5`.
- Es ermöglicht aber anderen Programmieren, eine neue Klasse zu schreiben, die einen Gleichheitsvergleich mit unseren `Point`-Instanzen implementiert.
- Wenn wir `__eq__` implementieren, dann ist der richtige Type Hint für den Rückgabewert `bool | NotImplementedType`.

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

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

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

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

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

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

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

# Beispiel: Point mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Program `point_user_2.py` verwendet haben.

```python
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

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

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

```
 str(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
repr(p1) = '<point.Point object at 0x7f5fd9705b80>'
     p1   =  <point.Point object at 0x7f5fd9705b80>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Program `point_user_2.py` verwendet haben.
- Das tun wir in dem neuen Programm `point_with_dunder_user.py`.

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

from point_with_dunder import Point

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

print(f"{str(p1) = }")  # a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Beispiel: Point mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Program `point_user_2.py` verwendet haben.
- Das tun wir in dem neuen Programm `point_with_dunder_user.py`.
- Die Ausgabe dieses Programms passt viel besser zu dem, was wir uns vorstellen.

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

from point_with_dunder import Point

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

print(f" {str(p1) = }")  # a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Program `point_user_2.py` verwendet haben.
- Das tun wir in dem neuen Programm `point_with_dunder_user.py`.
- Die Ausgabe dieses Programms passt viel besser zu dem, was wir uns vorstellen.
- Die Funktion `str` liefert uns nun kurzen aber informativen Output für Instanzen der Klasse `Point`.

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

from point_with_dunder import Point

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

print(f" {str(p1) = }")  # a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden benutzen

- Das tun wir in dem neuen Programm `point_with_dunder_user.py`.
- Die Ausgabe dieses Programms passt viel besser zu dem, was wir uns vorstellen.
- Die Funktion `str` liefert uns nun kurzen aber informativen Output für Instanzen der Klasse `Point`.
- Der `repr`-Operator gibt uns Text, den wir im Prinzip in den Python-Interpreter kopieren könnten und mit dem wir so das gleiche Objekt nochmal erstellen könnten.

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

from point_with_dunder import Point

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

print(f" {str(p1) = }")  # a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden benutzen

- Die Ausgabe dieses Programms passt viel besser zu dem, was wir uns vorstellen.
- Die Funktion `str` liefert uns nun kurzen aber informativen Output für Instanzen der Klasse `Point`.
- Der `repr`-Operator gibt uns Text, den wir im Prinzip in den Python-Interpreter kopieren könnten und mit dem wir so das gleiche Objekt nochmal erstellen könnten.
- Die Gleicheits- und Ungleichheits-Operatoren zeigen ebenfalls viel vernünftigeres Verhalten und sehen, wenn zwei Punkte die selben Koordinaten haben.

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

from point_with_dunder import Point

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

print(f" {str(p1) = }")  # a short string representation of p1
print(f"     p1   =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1   =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ python3 point_with_dunder_user.py ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Beispiel: `Point` mit Dunder Methoden benutzen

- Die Funktion `str` liefert uns nun kurzen aber informativen Output für Instanzen der Klasse `Point`.
- Der `repr`-Operator gibt uns Text, den wir im Prinzip in den Python-Interpreter kopieren könnten und mit dem wir so das gleiche Objekt nochmal erstellen könnten.
- Die Gleicheits- und Ungleichheits-Operatoren zeigen ebenfalls viel vernünftigeres Verhalten und sehen, wenn zwei Punkte die selben Koordinaten haben.
- Sie funktionieren auch richtig wenn das andere Objekt kein Punkt ist.

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

from point_with_dunder import Point

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

print(f" {str(p1) = }")  # a short string representation of p1
print(f"      p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"      p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

# Zusammenfassung

## Zusammenfassung

- Wir haben nun unsere Reise in das Reich der so genannten *Dunder*-Methoden begonnen.

# Zusammenfassung

- Wir haben nun unsere Reise in das Reich der so genannten *Dunder*-Methoden begonnen.
- Diese Methoden kontrollieren viel von dem Verhalten der Operatoren und Konstrukte der Sprache Python.

## Zusammenfassung

- Wir haben nun unsere Reise in das Reich der so genannten *Dunder*-Methoden begonnen.
- Diese Methoden kontrollieren viel von dem Verhalten der Operatoren und Konstrukte der Sprache Python.
- Mit `__str__`, `__repr__`, `__eq__` und `__ne__` haben wir bereits vier Methoden berührt, die wir bereits sehr oft verwendet haben – wenn auch indirekt.

## Zusammenfassung

- Wir haben nun unsere Reise in das Reich der so genannten *Dunder*-Methoden begonnen.
- Diese Methoden kontrollieren viel von dem Verhalten der Operatoren und Konstrukte der Sprache Python.
- Mit `__str__`, `__repr__`, `__eq__` und `__ne__` haben wir bereits vier Methoden berührt, die wir bereits sehr oft verwendet haben – wenn auch indirekt.
- Welche anderen Abenteuer erwarten uns so tief im Getriebe der Python-Machine?

# 谢谢您门！

# Thank you!

# Vielen Dank!

# References I


[1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 130, 131).

[2] "Basic Customizations: `object.__repr__`". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__repr__ (besucht am 2024-09-25) (siehe S. 21–29).

[3] "Basic Customizations: `object.__str__`". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__str__ (besucht am 2024-09-25) (siehe S. 21–29).

[4] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 130).

[5] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 130).

[6] Oleg Broytman und Jim J. Jewett. *`str(container)` should call `str(item)`, not `repr(item)` [Rejected]*. Python Enhancement Proposal (PEP) 3140. Beaverton, OR, USA: Python Software Foundation (PSF), 27.–28. Mai 2008. URL: https://peps.python.org/pep-3140 (besucht am 2024-12-08) (siehe S. 30–52).

[7] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 130).

[8] "Built-in Constants". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/constants.html (besucht am 2024-12-09) (siehe S. 106).

[9] 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. 131, 132).

[10] "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. 130).

# References II


[11] 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. 130).

[12] 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. 130).

[13] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 130).

[14] 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. 132).

[15] 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. 130).

[16] Mike James. *Programmer's Python: Everything is an Object – Something Completely Different*. 2. Aufl. I/O Press, 25. Juni 2022 (siehe S. 5–14).

[17] ."stderr, stdin, stdout – Standard I/O Streams". In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html (besucht am 2024-10-30) (siehe S. 131).

[18] 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. 131).

[19] 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. 130).

# References III


[20] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 130).

[21] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 130).

[22] 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. 130).

[23] 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. 130).

[24] "Objects, Values and Types". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.1. URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types (besucht am 2024-12-07) (siehe S. 5–14).

[25] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 130).

[26] 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. 130).

[27] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 130).

[28] Eric V. „ericvsmith“ 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. 130).

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

# References IV


[30] 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. 130).

[31] 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. 130, 132).

[32] Guido van Rossum und David Ascher. *Rich Comparisons*. Python Enhancement Proposal (PEP) 207. Beaverton, OR, USA: Python Software Foundation (PSF), 25. Juli 2000. URL: https://peps.python.org/pep-0207 (besucht am 2024-12-08) (siehe S. 64–96).

[33] 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. 131).

[34] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 130).

[35] Gregory R. Warnes. *IEEE 754 Floating Point Special Values [Rejected]*. Python Enhancement Proposal (PEP) 754. Beaverton, OR, USA: Python Software Foundation (PSF), 28. März 2003. URL: https://peps.python.org/pep-0754 (besucht am 2024-12-08) (siehe S. 64–95).

[36] 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. 64–95, 130).

[37] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 130).

# Glossary (in English) I

**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[5,23,37]. Learn more at https://www.gnu.org/software/bash.

**f-string** let you include the results of expressions in strings[7,10–12,22,28]. 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[27,31]. Learn more at https://git-scm.com.

**GitHub** is a website where software projects can be hosted and managed via the Git VCS[25,31]. Learn more at https://github.com.

**IT** information technology

**Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows[1,13,26,30,34]. 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.

**Microsoft Windows** is a commercial proprietary operating system[4]. 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.

**Mypy** is a static type checking tool for Python[20] that makes use of type hints. Learn more at https://github.com/python/mypy and in[36].

**Python** The Python programming language[15,19,21,36], i.e., what you will learn about in our book[36]. Learn more at https://python.org.

# Glossary (in English) II

**stderr** The *standard error stream* is one of the three pre-defined streams of a console process (together with the standard input stream (stdin) and the stdout)[17]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to standard error stream (stderr). If you run a program in a terminal, then the text that a process writes to its stderr appears in the console.

**stdin** The *standard input stream* is one of the three pre-defined streams of a console process (together with the stdout and the stderr)[17]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

**stdout** The *standard output stream* is one of the three pre-defined streams of a console process (together with the stdin and the stderr)[17]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its stdout appears in the console.

**(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[1,9]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↲. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

**type hint** are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[18,33]. 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) III

**Ubuntu** is a variant of the open source operating system Linux[9,14]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at `https://ubuntu.com`. If you are in China, you can download it from `https://mirrors.ustc.edu.cn/ubuntu-releases`.

**VCS** A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[31]. 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.