



## Programming with Python

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

Thomas Weise (汤卫思)  
[tweise@hfuu.edu.cn](mailto:tweise@hfuu.edu.cn)

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

人工智能与大数据学院  
合肥大学  
中国安徽省合肥市

# Programming with Python



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

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



# Outline

1. Einleitung
2. `__str__` und `__repr__`
3. Strings und Gleichheit
4. Zusammenfassung





# Einleitung



# Einleitung



- In Python, *everything is an object*<sup>16,24</sup>, also „Alles ist ein Objekt“.

# Einleitung



- In Python, *everything is an object*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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*<sup>16,24</sup>, 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 für „double underscore“, also „Doppelter Unterstrich“, also `__`.

# Einleitung

- In Python, *everything is an object*<sup>16,24</sup>, 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 für „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  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<sup>3</sup>.



## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts unterscheiden
  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts liefern<sup>3</sup>. Diese Repräsentation ist hauptsächlich für Endbenutzer.



## `__str__` und `__repr__`

- Wir können zwei Arten von textuellen Repräsentationen eines Objekts unterscheiden
  1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts liefern<sup>3</sup>. 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<sup>3</sup>. 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<sup>3</sup>. 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<sup>2</sup>.



## `__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<sup>3</sup>. 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<sup>2</sup>. Das Zielpublikum sind hier Programmierer, 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 unterscheiden
  - 1. `str(o)` sollte eine kurze Stringrepräsentation des Objekts liefern<sup>3</sup>. 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 erneut zu erzeugen<sup>2</sup>. Das Zielpublikum sind hier Programmierer, 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<sup>3</sup>. 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<sup>2</sup>. Das Zielpublikum sind hier Programmierer, 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 der Name des Datentyps und die ID des Objekts geliefert.

# Beispiel: str und repr

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

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{list = }, but {list = }") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

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

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }, but {list!r}') # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 list = [1, 2, 3], but list = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11     ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

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

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- In Programm `str_vs_repr.py` vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variablen `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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6   str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8   repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11      ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- In Programm `str_vs_repr.py` vergleichen wir die beiden Funktionen.
- Wir erstellen zuerst eine Ganzzahl-Variablen `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".

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{list = }, but {list = }") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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'.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11     ↪ 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?

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = repr(list)}) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = repr(list)}) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6   str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8   repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11      ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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`.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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 Ganzahlen `1`, `2` und `3` beinhaltet.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 l1: list[int] = [1, 2, 3] # A list of integers.
14 l2: list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{l1 = }, but {l2 = }') # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11        ↪ 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 `11`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `12`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added ''' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6   str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8   repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11      ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 l1: list[int] = [1, 2, 3] # A list of integers.
14 l2: list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{l1 = }', but {l2 = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11     ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Nun erstellen wir zwei Kollektionen.
- Zuerst erzeugen wir die Liste 11, die die drei Ganzzahlen 1, 2 und 3 beinhaltet.
- Dann erstellen wir die List 12, die drei Strings, nämlich "1", "2" und "3".
- Dann wenden wir geben wir beide Listen aus wobei intern str(11) und str(12) werden.
- Das Ergebnis von

```
print(f"11 = {11}, but 12 = {12}")  
ist 11 = [1, 2, 3],  
but 12 = ['1', '2', '3'].
```

```
"""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.  
  
list[int] = [1, 2, 3] # A list of integers.  
list[str] = ["1", "2", "3"] # A list of strings.  
print(f"11 = {11}, but 12 = {12}") # 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) = {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) = {repr(right_now)}")  
print(f" right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
123  
123  
123  
'123'  
11 = [1, 2, 3], but 12 = ['1', '2', '3']  
str(right_now) = '2026-02-20 07:31:36.550083+00:00'  
    right_now = 2026-02-20 07:31:36.550083+00:00  
repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,  
    ↪ tzinfo=datetime.timezone.utc)'  
    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,  
    ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Zuerst erzeugen wir die Liste `11`, die die drei Ganzzahlen `1`, `2` und `3` beinhaltet.
- Dann erstellen wir die List `12`, die drei Strings, nämlich `"1"`, `"2"` und `"3"`.
- Dann wenden wir geben wir beide Listen aus wobei intern `str(11)` und `str(12)` werden.
- Das Ergebnis von  
`print(f"{11 = }, but {12 = }")`  
ist `11 = [1, 2, 3],`  
`but 12 = ['1', '2', '3']`.
- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von `12`.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{11 = }, but {12 = }") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11            ↪ 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 = {l1}, but l2 = {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 von Python mit `str` oder `repr` erzeugt wird, dann werden die Elemente der Kollektionen immer `repr` zu Strings konvertiert, nie mit `str`<sup>6</sup>.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"l1 = {l1}, but l2 = {l2}") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now)} = {right_now!s}")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f" {repr(right_now)} = {right_now!r}")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11     ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Das Ergebnis von

```
print(f"l1 = {l1}, but l2 = {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 von Python mit `str` oder `repr` erzeugt wird, dann werden die Elemente der Kollektionen immer `repr` zu Strings konvertiert, nie mit `str`<sup>6</sup>.

- Andernfalls könnten wir nicht zwischen `l1` und `l2` in der Ausgabe unterscheiden.

```
"""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.

list[int] = [1, 2, 3] # A list of integers.
list[str] = ["1", "2", "3"] # A list of strings.
print(f"l1 = {l1}, but l2 = {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) = {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) = {repr(right_now)}")
print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
    ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Beachten Sie wieder die einzelnen Anführungszeichen um die String-Elemente von 12.
- Wenn die textuelle Repräsentation der Standard-Kollektionstypen von Python mit str oder repr erzeugt wird, dann werden die Elemente der Kollektionen immer repr zu Strings konvertiert, nie mit str<sup>6</sup>.
- Andernfalls könnten wir nicht zwischen 11 und 12 in der Ausgabe unterscheiden.
- Ein anderes gutes Beispiel für den Unterschied zwischen str und repr ist Python's Klasse datetime.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = repr}) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
    ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Wenn die textuelle Repräsentation der Standard-Kollektionstypen von Python mit `str` oder `repr` erzeugt wird, dann werden die Elemente der Kollektionen immer `repr` zu Strings konvertiert, nie mit `str`<sup>6</sup>.
- 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 l1: list[int] = [1, 2, 3] # A list of integers.
14 l2: list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{l1 = }', but {l2 = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
    ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Andernfalls könnten wir nicht zwischen `11` und `12` 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list[int]}') # str(list) uses str for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now)} ")
24 print(f"     right_now = {right_now!s} ")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now)} ")
29 print(f"     right_now = {right_now!r} ")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11            ↪ tzinfo=datetime.timezone.utc)
```

# Beispiel: str und repr

- Andernfalls könnten wir nicht zwischen `11` und `12` 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{list = }', but {list = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{list = }", but {list = repr(list)}) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f" {repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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}}"`.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{list = }, but {list = }") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f" {repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{list = }", but {list = repr(list)}) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"     right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"     right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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}}"`.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"{{l1 = }}, but {{l2 = }}") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {{str(right_now) = }}")
24 print(f"     right_now = {{right_now!s}}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{{repr(right_now) = }}")
29 print(f"     right_now = {{right_now!r}}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6     str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7         right_now = 2026-02-20 07:31:36.550083+00:00
8     repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9         ↪ tzinfo=datetime.timezone.utc)'
10        right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11         ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"list = {list}") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f"str(right_now) = {right_now}")
24 print(f"    right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"repr(right_now) = {repr(right_now)}")
29 print(f"    right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
     right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
     ↪ 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!

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added `` around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"list = {list}") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f"str(right_now) = {right_now}")
24 print(f"    right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"repr(right_now) = {repr(right_now)}")
29 print(f"    right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
     right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
     ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 list[int] = [1, 2, 3] # A list of integers.
14 list[str] = ["1", "2", "3"] # A list of strings.
15 print(f"list = {list}") # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f"str(right_now) = {str(right_now)}")
24 print(f"repr(right_now) = {repr(right_now)}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"repr(right_now) = {repr(right_now)}")
29 print(f"str(right_now) = {str(right_now)}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 11 = [1, 2, 3], but 12 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11        ↪ 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.

```
1 """An example comparing `str` and `repr`."""
2
3 from datetime import UTC, datetime
4
5 the_int: int = 123 # An integer with value 123.
6 print(the_int) # This is identical to `print(str(the_int))`.
7 print(repr(the_int)) # Prints the same as above.
8
9 the_string: str = "123" # A string, with value "123".
10 print(the_string) # This is identical to `print(str(the_string))`.
11 print(repr(the_string)) # Notice the added '' around the string.
12
13 l1: list[int] = [1, 2, 3] # A list of integers.
14 l2: list[str] = ["1", "2", "3"] # A list of strings.
15 print(f'{l1 = }', but {l2 = }) # str(list) uses repr for list elements.
16
17 # Get the date and time when this program was run.
18 right_now: datetime = datetime.now(tz=UTC)
19
20 # Print the human-readable, concise string representation for users who
21 # want to know that the object means but do not necessarily need to know
22 # its detailed content.
23 print(f" {str(right_now) = }")
24 print(f"      right_now = {right_now!s}")
25
26 # Print the format for programmers who need to understand the exact
27 # values of all attributes of `right_now`.
28 print(f"{repr(right_now) = }")
29 print(f"      right_now = {right_now!r}")
```

↓ python3 str\_vs\_repr.py ↓

```
1 123
2 123
3 123
4 '123'
5 l1 = [1, 2, 3], but l2 = ['1', '2', '3']
6 str(right_now) = '2026-02-20 07:31:36.550083+00:00'
7     right_now = 2026-02-20 07:31:36.550083+00:00
8 repr(right_now) = 'datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
9     ↪ tzinfo=datetime.timezone.utc)'
10    right_now = datetime.datetime(2026, 2, 20, 7, 31, 36, 550083,
11        ↪ 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.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9         A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26             The constructor: Create a point and set its coordinates.
27
28             :param x: the x-coordinate of the point
29             :param y: the y-coordinate of the point
30             """
31
32             if not (isfinite(x) and isfinite(y)):
33                 raise ValueError(f"x={x} and y={y} must both be finite.")
34             #: the x-coordinate of the point
35             self.x: Final[int | float] = x
36             #: the y-coordinate of the point
37             self.y: Final[int | float] = y
38
39     def distance(self, p: "Point") -> float:
40         """
41             Get the distance to another point.
42
43             :param p: the other point
44             :return: the distance
45
46             >>> Point(1, 1).distance(Point(4, 4))
47             4.242640687119285
48
49             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.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9         A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24     def __init__(self, x: int | float, y: int | float) -> None:
25         """
26             The constructor: Create a point and set its coordinates.
27
28             :param x: the x-coordinate of the point
29             :param y: the y-coordinate of the point
30             """
31
32         if not (isfinite(x) and isfinite(y)):
33             raise ValueError(f"x={x} and y={y} must both be finite.")
34         #: the x-coordinate of the point
35         self.x: Final[int | float] = x
36         #: the y-coordinate of the point
37         self.y: Final[int | float] = y
38
39     def distance(self, p: "Point") -> float:
40         """
41             Get the distance to another point.
42
43             :param p: the other point
44             :return: the distance
45
46             >>> Point(1, 1).distance(Point(4, 4))
47             4.242640687119285
48             """
49
50             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 hatten.

```
1 """A simple class for points."""
2
3 from math import isfinite, sqrt
4 from typing import Final
5
6
7 class Point:
8     """
9         A class for representing a point in the two-dimensional plane.
10
11     >>> p = Point(1, 2.5)
12     >>> p.x
13     1
14     >>> p.y
15     2.5
16
17     >>> try:
18     ...     Point(1, 1e308 * 1e308)
19     ... except ValueError as ve:
20     ...     print(ve)
21     x=1 and y=inf must both be finite.
22     """
23
24
25     def __init__(self, x: int | float, y: int | float) -> None:
26         """
27             The constructor: Create a point and set its coordinates.
28
29             :param x: the x-coordinate of the point
30             :param y: the y-coordinate of the point
31             """
32             if not (isfinite(x) and isfinite(y)):
33                 raise ValueError(f"x={x} and y={y} must both be finite.")
34             #: the x-coordinate of the point
35             self.x: Final[int | float] = x
36             #: the y-coordinate of the point
37             self.y: Final[int | float] = y
38
39     def distance(self, p: "Point") -> float:
40         """
41             Get the distance to another point.
42
43             :param p: the other point
44             :return: the distance
45
46             >>> Point(1, 1).distance(Point(4, 4))
47             4.242640687119285
48
49             """
50             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 hatten.
- Damals haben wir schon die Dunder-Methode `__init__` kennengelernt.

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

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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 hatten.
- 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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is` not
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point und Gleichheit

- Diese Klasse war ziemlich nützlich, als wir Klassen für verschiedene Formen implementiert hatten.
- 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`.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is` not
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is` not
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
22
23
24 ↓ python3 point_user_2.py ↓
25
26 str(p1) = '<point.Point object at 0x7f488ebadc10>'
27     p1 = <point.Point object at 0x7f488ebadc10>
28 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
29     p1 = <point.Point object at 0x7f488ebadc10>
30 (p1 is p2) = False
31 (p1 is p3) = False
32 (p1 == p2) = False
33 (p1 == p3) = False
34 (p1 != p2) = True
35 (p1 != p3) = True
36 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is` not
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
22
23
24 ↓ python3 point_user_2.py ↓
25
26 str(p1) = '<point.Point object at 0x7f488ebadc10>'
27     p1 = <point.Point object at 0x7f488ebadc10>
28 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
29     p1 = <point.Point object at 0x7f488ebadc10>
30 (p1 is p2) = False
31 (p1 is p3) = False
32 (p1 == p2) = False
33 (p1 == p3) = False
34 (p1 != p2) = True
35 (p1 != p3) = True
36 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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 Programm 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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{") # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
22
23
24 ↓ python3 point_user_2.py ↓
25
26 str(p1) = '<point.Point object at 0x7f488ebadc10>'
27     p1 = <point.Point object at 0x7f488ebadc10>
28 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
29     p1 = <point.Point object at 0x7f488ebadc10>
30
31 (p1 is p2) = False
32 (p1 is p3) = False
33 (p1 == p2) = False
34 (p1 == p3) = False
35 (p1 != p2) = True
36 (p1 != p3) = True
37 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{}) # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{}) # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f"{repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f"{{(p1 is p2) = }}") # False, p1 and p2 are different objects
15 print(f"{{(p1 is p3) = }}") # False, p1 and p3 are different objects
16 print(f"{{(p1 == p2) = }}") # False, because without dunder `==` = `is`
17 print(f"{{(p1 == p3) = }}") # False, but should ideally be True
18 print(f"{{(p1 != p2) = }}") # True, because without dunder `==` = `is`
19 print(f"{{(p1 != p3) = }}") # True, but should ideally be False
20
21 print(f"{{(p1 == 5) = }}") # comparison with the integer 5 yields False

↓ python3 point_user_2.py ↓

1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5)    # Create a first point.
6 p2: Point = Point(7, 8)    # Create a second, different point.
7 p3: Point = Point(3, 5)    # Create a third point, which equals the first.
8
9 print(f" {str(p1) = }")   # should be a short string representation of p1
10 print(f"     p1 = {p1!s}")  # (almost) the same as the above
11 print(f"{repr(p1) = }")   # should be a representation for programmers
12 print(f"     p1 = {p1!r}")  # (almost) the same as the above
13
14 print(f" {(p1 is p2) = }") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3) = }") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2) = }") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3) = }") # False, but should ideally be True
18 print(f" {(p1 != p2) = }") # True, because without dunder `==` = `is`
19 print(f" {(p1 != p3) = }") # True, but should ideally be False
20
21 print(f" {(p1 == 5) = }")  # comparison with the integer 5 yields False
```

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1     """Examples for using our class :class:`Point` without dunder."""
2
3     from point import Point
4
5     p1: Point = Point(3, 5)    # Create a first point.
6     p2: Point = Point(7, 8)    # Create a second, different point.
7     p3: Point = Point(3, 5)    # Create a third point, which equals the first.
8
9     print(f" {str(p1)} = {}")  # should be a short string representation of p1
10    print(f"      p1 = {p1!s}")  # (almost) the same as the above
11    print(f" {repr(p1)} = {}")  # should be a representation for programmers
12    print(f"      p1 = {p1!r}")  # (almost) the same as the above
13
14    print(f" {(p1 is p2)} = {}")  # False, p1 and p2 are different objects
15    print(f" {(p1 is p3)} = {}")  # False, p1 and p3 are different objects
16    print(f" {(p1 == p2)} = {}")  # False, because without dunder `==` = `is`
17    print(f" {(p1 == p3)} = {}")  # False, but should ideally be True
18    print(f" {(p1 != p2)} = {}")  # True, because without dunder `!=` = `is`
19    print(f" {(p1 != p3)} = {}")  # True, but should ideally be False
20
21    print(f" {(p1 == 5)} = {}")  # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1     str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3     repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5     (p1 is p2) = False
6     (p1 is p3) = False
7     (p1 == p2) = False
8     (p1 == p3) = False
9     (p1 != p2) = True
10    (p1 != p3) = True
11    (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`.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{}) # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{}) # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1     """Examples for using our class :class:`Point` without dunder."""
2
3     from point import Point
4
5     p1: Point = Point(3, 5)    # Create a first point.
6     p2: Point = Point(7, 8)    # Create a second, different point.
7     p3: Point = Point(3, 5)    # Create a third point, which equals the first.
8
9     print(f" {str(p1)} = ")    # should be a short string representation of p1
10    print(f"     p1 = {p1!s}")    # (almost) the same as the above
11    print(f" {repr(p1)} = ")    # should be a representation for programmers
12    print(f"     p1 = {p1!r}")    # (almost) the same as the above
13
14    print(f" {(p1 is p2)} = ")    # False, p1 and p2 are different objects
15    print(f" {(p1 is p3)} = ")    # False, p1 and p3 are different objects
16    print(f" {(p1 == p2)} = ")    # False, because without dunder `==` = `is`
17    print(f" {(p1 == p3)} = ")    # False, but should ideally be True
18    print(f" {(p1 != p2)} = ")    # True, because without dunder `!=` = `is`
19    print(f" {(p1 != p3)} = ")    # True, but should ideally be False
20
21    print(f" {(p1 == 5)} = ")    # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1     str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3     repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5     (p1 is p2) = False
6     (p1 is p3) = False
7     (p1 == p2) = False
8     (p1 == p3) = False
9     (p1 != p2) = True
10    (p1 != p3) = True
11    (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{}) # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{}) # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is` not equal
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
22
23
24 ↓ python3 point_user_2.py ↓
25
26 str(p1) = '<point.Point object at 0x7f488ebadc10>'
27     p1 = <point.Point object at 0x7f488ebadc10>
28 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
29     p1 = <point.Point object at 0x7f488ebadc10>
30
31 (p1 is p2) = False
32 (p1 is p3) = False
33 (p1 == p2) = False
34 (p1 == p3) = False
35 (p1 != p2) = True
36 (p1 != p3) = True
37 (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<sup>32</sup>.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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<sup>32</sup>.
- Das ist nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht stimmt<sup>36</sup> ... vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte<sup>35</sup>, im Moment sehe ich das aber nicht.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point und Gleichheit

- Das ist nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht stimmt<sup>36</sup> ... vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte<sup>35</sup>, 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.<sup>32</sup>

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point und Gleichheit

- Das ist nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht stimmt<sup>36</sup> ... vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte<sup>35</sup>, 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.<sup>32</sup>
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
22
23
24 ↓ python3 point_user_2.py ↓
```

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point und Gleichheit

- Das ist nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht stimmt<sup>36</sup> ... vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte<sup>35</sup>, 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.<sup>32</sup>
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point und Gleichheit

- Das ist nicht immer der Fall, obwohl ich kein Beispiel kenne, wo es nicht stimmt<sup>36</sup> ... vielleicht war es früher mal so, dass das bei Fließkommazahlen auftreten konnte<sup>35</sup>, 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.<sup>32</sup>
- Zum Schluss vergleichen wir noch, ob `p1` das selbe wie die Ganzzahl `5` ist.
- Das sollte offensichtlich `False` ergeben.
- Und das tut es auch.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{") # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.<sup>32</sup>
- 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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder '==' = 'is'
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder '!=' = 'is'
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = {}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}{}) # (almost) the same as the above
11 print(f" {repr(p1)} = {}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}{}) # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = {}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = {}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = {}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = {}") # False, but should ideally be True
18 print(f" {(p1 != p2)} = {}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = {}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = {}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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...).

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)} = ") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)} = ") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)} = ") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)} = ") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)} = ") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)} = ") # False, but should ideally be True
18 print(f" {(p1 != p2)} = ") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)} = ") # True, but should ideally be False
20
21 print(f" {(p1 == 5)} = ") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (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`.

```
1 """A class for points, with string and equals dunder methods."""
2
3 from math import isfinite
4 from types import NotImplementedType
5 from typing import Final
6
7
8 class Point:
9     """A class for representing a point in the two-dimensional plane."""
10
11     def __init__(self, x: int | float, y: int | float) -> None:
12         """
13             The constructor: Create a point and set its coordinates.
14
15             :param x: the x-coordinate of the point
16             :param y: the y-coordinate of the point
17         """
18         if not (isfinite(x) and isfinite(y)):
19             raise ValueError(f"x={x} and y={y} must both be finite.")
20         #: the x-coordinate of the point
21         self.x: Final[int | float] = x
22         #: the y-coordinate of the point
23         self.y: Final[int | float] = y
24
25     def __repr__(self) -> str:
26         """
27             Get a representation of this object useful for programmers.
28
29             :return: `*Point(x, y)`
30
31         >>> repr(Point(2, 4))
32         'Point(2, 4)'
33         """
34         return f"Point({self.x}, {self.y})"
35
36     def __str__(self) -> str:
37         """
38             Get a concise string representation useful for end users.
39
40             :return: `*(x,y)`
41
42         >>> str(Point(2, 4))
43         '(2,4)'
44         """
45         return f"({self.x},{self.y})"
46
47     def __eq__(self, other) -> bool | NotImplementedType:
48         """
49             Check whether this point is equal to another object.
50
51             :param other: the other object
52             :return: 'True' if and only if 'other' is also a 'Point' and has
53                     the same coordinates; 'NotImplemented' if it is not a point
54
55         >>> Point(1, 2) == Point(2, 3)
56         False
57         >>> Point(1, 2) == Point(1, 2)
58         True
59         """
60         return (other.x == self.x) and (other.y == self.y) \
61             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.

```
"""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.

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
337
338
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
796
797
798
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
846
847
848
848
849
849
850
851
852
853
854
855
856
857
857
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
```



## 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.
  - Deshalb wird `__repr__` einen String der Form `"Point(x, y)"` liefern.

```
"""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

- Die kurze String-Präsentation, 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-Präsentation 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.

```
"""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-Präsentation 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`.

```
"""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-Präsentation 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.

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
717
718
719
719
720
721
722
723
724
725
726
727
727
728
729
729
730
731
732
733
734
735
736
737
737
738
739
739
740
741
742
743
744
745
746
746
747
748
748
749
749
750
751
752
753
754
755
756
757
757
758
759
759
760
761
762
763
764
765
766
766
767
768
768
769
769
770
771
772
773
774
775
776
776
777
778
778
779
779
780
781
782
783
784
785
785
786
787
787
788
788
789
789
790
791
792
793
794
795
795
796
796
797
797
798
798
799
799
800
801
802
803
804
805
805
806
807
807
808
808
809
809
810
811
812
813
814
814
815
815
816
816
817
817
818
818
819
819
820
821
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
831
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
841
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
851
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
861
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
871
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
911
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
921
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
931
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
941
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
951
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
961
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
971
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
981
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
991
992
993
993
994
995
995
996
996
997
997
998
998
999
999
1000
1000
```





# 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.*

## Beispiel: Point mit Dunder Methode

- 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 `==`.

```
"""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.

```
"""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.

```
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
337
338
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
748
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
796
797
798
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
817
818
819
819
820
821
822
823
824
825
826
827
827
828
829
829
830
831
832
833
834
835
836
837
837
838
839
839
840
841
842
843
844
845
846
846
847
848
848
849
849
850
851
852
853
854
855
856
857
857
858
859
859
860
861
862
863
864
865
866
866
867
868
868
869
869
870
871
872
873
874
875
875
876
876
877
877
878
878
879
879
880
881
882
883
884
885
885
886
886
887
887
888
888
889
889
890
891
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
901
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
```



## Beispiel: Point mit Dunder Methode

- 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`.

```
"""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 `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.

```
"""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`.

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

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

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

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

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

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

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

def __eq__(self, other) -> bool | NotImplemented:
    """
    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
    """
    if other.x == self.x and other.y == self.y:
        return True
    else:
        return NotImplemented

    """
    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.

```
1 """Examples for using our class :class:`Point` without dunder."""
2
3 from point import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # should be a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # should be a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, because without dunder `==` = `is`
17 print(f" {(p1 == p3)}") # False, but should ideally be True
18 print(f" {(p1 != p2)}") # True, because without dunder `!=` = `is`
19 print(f" {(p1 != p3)}") # True, but should ideally be False
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_user\_2.py ↓

```
1 str(p1) = '<point.Point object at 0x7f488ebadc10>'
2     p1 = <point.Point object at 0x7f488ebadc10>
3 repr(p1) = '<point.Point object at 0x7f488ebadc10>'
4     p1 = <point.Point object at 0x7f488ebadc10>
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = False
9 (p1 != p2) = True
10 (p1 != p3) = True
11 (p1 == 5) = False
```

# Beispiel: Point mit Dunder Methoden benutzen

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

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # sa representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our `__eq__` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of `__eq__`
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (p1 == 5) = False
```

# Beispiel: Point mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Programm `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.

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # sa representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our `__eq__` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of `__eq__`
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (p1 == 5) = False
```

# Beispiel: Point mit Dunder Methoden benutzen

- Wir benutzen nun unsere neue Klasse `Point` genauso wie wir die alte in Programm `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`.

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1 = Point(3, 5) # Create a first point.
6 p2 = Point(7, 8) # Create a second, different point.
7 p3 = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our ``__eq__`` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of ``__eq__``
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (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.

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our ``__eq__`` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of ``__eq__``
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (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 Gleichheits- und Ungleichheits-Operatoren zeigen ebenfalls viel vernünftigeres Verhalten und sehen, wenn zwei Punkte die selben Koordinaten haben.

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our ``__eq__`` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of ``__eq__``
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (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 Gleichheits- 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.

```
1 """Examples for using our class :class:`Point` with dunder methods."""
2
3 from point_with_dunder import Point
4
5 p1: Point = Point(3, 5) # Create a first point.
6 p2: Point = Point(7, 8) # Create a second, different point.
7 p3: Point = Point(3, 5) # Create a third point, which equals the first.
8
9 print(f" {str(p1)}") # a short string representation of p1
10 print(f"     p1 = {p1!s}") # (almost) the same as the above
11 print(f" {repr(p1)}") # a representation for programmers
12 print(f"     p1 = {p1!r}") # (almost) the same as the above
13
14 print(f" {(p1 is p2)}") # False, p1 and p2 are different objects
15 print(f" {(p1 is p3)}") # False, p1 and p3 are different objects
16 print(f" {(p1 == p2)}") # False, calls our ``__eq__`` method
17 print(f" {(p1 == p3)}") # True, as it should be, because of ``__eq__``
18 print(f" {(p1 != p2)}") # True, returns `not __eq__`
19 print(f" {(p1 != p3)}") # False, as it should be
20
21 print(f" {(p1 == 5)}") # comparison with the integer 5 yields False
```

↓ python3 point\_with\_dunder\_user.py ↓

```
1 str(p1) = '(3,5)'
2     p1 = (3,5)
3 repr(p1) = 'Point(3, 5)'
4     p1 = Point(3, 5)
5 (p1 is p2) = False
6 (p1 is p3) = False
7 (p1 == p2) = False
8 (p1 == p3) = True
9 (p1 != p2) = True
10 (p1 != p3) = False
11 (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?



谢谢您们！  
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\\_\\_](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\\_\\_](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 Beginner's 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), 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 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<sup>5,23,37</sup>. Learn more at <https://www.gnu.org/software/bash>.

**f-string** let you include the results of expressions in strings<sup>7,10–12,22,28</sup>. They can contain expressions (in curly braces) like `f"{}a{}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<sup>27,31</sup>. Learn more at <https://git-scm.com>.

**GitHub** is a website where software projects can be hosted and managed via the Git VCS<sup>25,31</sup>. 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<sup>1,13,26,30,34</sup>. 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<sup>4</sup>. 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>.

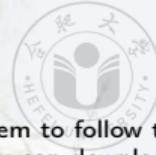
**Mypy** is a static type checking tool for Python<sup>20</sup> that makes use of type hints. Learn more at <https://github.com/python/mypy> and in<sup>36</sup>.

**Python** The Python programming language<sup>15,19,21,36</sup>, i.e., what you will learn about in our book<sup>36</sup>. 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`)<sup>17</sup>. 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`)<sup>17</sup>. 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`)<sup>17</sup>. 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<sup>1,9</sup>. 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 `Windows + R`, dann Schreiben von `cmd`, dann Druck auf `Enter`. 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<sup>18,33</sup>. 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<sup>9,14</sup>. 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<sup>31</sup>. 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.