



# Programming with Python

## 15. Variablen: Typen und Type Hints

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

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

应用优化研究所  
人工智能与大数据学院  
合肥大学  
中国安徽省合肥市

# Programming with Python



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

Die Webseite mit dem Lehrmaterial dieses Kurses ist <https://thomasweise.github.io/programmingWithPython> (siehe auch den QR-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. Verwirrung mit Typen
3. Static Type Checker: Mypy
4. Type Hints
5. Zusammenfassung





# Einleitung



# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...

## Variablen und Datentypen

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 float_var = 3.0 # 3.0 is a float and it is stored in float_var.  
5 print(type(float_var)) # This prints "<class 'float'>".  
6  
7 str_var = f"f{float_var = }" # Render an f-string into str_var.  
8 print(type(str_var)) # This prints "<class 'str'>".  
9  
10 bool_var = (1 == 0) # 1 == 0 is False, so a bool is stored in bool_var.  
11 print(type(bool_var)) # This prints "<class 'bool'>".  
12  
13 none_var = None # We create none_var which, well, holds None.  
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```

↓ python3 variable\_types.py ↓

```
1 <class 'int'>  
2 <class 'float'>  
3 <class 'str'>  
4 <class 'bool'>  
5 <class 'NoneType'>
```

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...
- In dem kleinen Programm `variable_types.py` können wir den Datentyp sehen, wenn wir ein Objekt in einer Variablen `var` speichern und dann `type(var)` aufrufen.

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...
- In dem kleinen Programm `variable_types.py` können wir den Datentyp sehen, wenn wir ein Objekt in einer Variablen `var` speichern und dann `type(var)` aufrufen.
- Es ist klar, dass eine Variable, die einen Ganzahlwert speichert, `int` als Datentyp hat.

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...
- In dem kleinen Programm `variable_types.py` können wir den Datentyp sehen, wenn wir ein Objekt in einer Variablen `var` speichern und dann `type(var)` aufrufen.
- Es ist klar, dass eine Variable, die einen Ganzahlwert speichert, `int` als Datentyp hat.
- Eine Variable, die eine Fließkommazahl speichert, hat den Datentyp `float`.

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...
- In dem kleinen Programm `variable_types.py` können wir den Datentyp sehen, wenn wir ein Objekt in einer Variablen `var` speichern und dann `type(var)` aufrufen.
- Es ist klar, dass eine Variable, die einen Ganzahlwert speichert, `int` als Datentyp hat.
- Eine Variable, die eine Fließkommazahl speichert, hat den Datentyp `float`.
- Und so weiter.

# Variablen und Datentypen



- Eine Variable ist im Grunde ein Name, der auf ein Objekt zeigt.
- Jedes Objekt hat einen Datentyp, z. B. `int`, `bool`, `str`, ...
- In dem kleinen Programm `variable_types.py` können wir den Datentyp sehen, wenn wir ein Objekt in einer Variablen `var` speichern und dann `type(var)` aufrufen.
- Es ist klar, dass eine Variable, die einen Ganzahlwert speichert, `int` als Datentyp hat.
- Eine Variable, die eine Fließkommazahl speichert, hat den Datentyp `float`.
- Und so weiter.
- Da gibt es nicht viel dazu zu sagen.

## Variablen und Datentypen

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 float_var = 3.0 # 3.0 is a float and it is stored in float_var.  
5 print(type(float_var)) # This prints "<class 'float'>".  
6  
7 str_var = f"f{float_var = }" # Render an f-string into str_var.  
8 print(type(str_var)) # This prints "<class 'str'>".  
9  
10 bool_var = (1 == 0) # 1 == 0 is False, so a bool is stored in bool_var.  
11 print(type(bool_var)) # This prints "<class 'bool'>".  
12  
13 none_var = None # We create none_var which, well, holds None.  
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```

↓ python3 variable\_types.py ↓

```
1 <class 'int'>  
2 <class 'float'>  
3 <class 'str'>  
4 <class 'bool'>  
5 <class 'NoneType'>
```

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung

- Vielleicht gibt es doch etwas dazu zu sagen.



# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.
- Sie dürfen dann nur Werte von genau diesem Datentyp in der Variable speichern.

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.
- Sie dürfen dann nur Werte von genau diesem Datentyp in der Variable speichern.
- In Python müssen Sie keinen Datentyp für eine Variable zu spezifizieren.

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.
- Sie dürfen dann nur Werte von genau diesem Datentyp in der Variable speichern.
- In Python müssen Sie keinen Datentyp für eine Variable zu spezifizieren.
- Sie können z. B. zuerst einen Text-String in einer Variable speichern und später dann einen Booleschen Wert.

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.
- Sie dürfen dann nur Werte von genau diesem Datentyp in der Variable speichern.
- In Python müssen Sie keinen Datentyp für eine Variable zu spezifizieren.
- Sie können z. B. zuerst einen Text-String in einer Variable speichern und später dann einen Booleschen Wert.
- Das bedeutet offensichtlich, dass verschiedene Programmiersprachen verschiedene Ansätze haben, wie Datentypen gehandhabt werden.

# Arten von Programmiersprachen hinsichtlich Datentyp-Behandlung



- Vielleicht gibt es doch etwas dazu zu sagen.
- Wenn Sie eine Variable in einer Programmiersprache wie C deklarieren, dann müssen Sie ihren Datentyp mit angeben.
- Sie dürfen dann nur Werte von genau diesem Datentyp in der Variable speichern.
- In Python müssen Sie keinen Datentyp für eine Variable zu spezifizieren.
- Sie können z. B. zuerst einen Text-String in einer Variable speichern und später dann einen Booleschen Wert.
- Das bedeutet offensichtlich, dass verschiedene Programmiersprachen verschiedene Ansätze haben, wie Datentypen gehandhabt werden.
- Schauen wir uns also einmal die grundlegenden, Datentyp-bezogenen Eigenschaften<sup>6</sup> von Programmiersprachen an.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

- Solche Sprachen verbieten die Interaktion zwischen inkompatiblen Typen.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

- Solche Sprachen verbieten die Interaktion zwischen inkompatiblen Typen.
- Man kann also nicht so etwas wie `"str" + 5` machen.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

- Solche Sprachen verbieten die Interaktion zwischen inkompatiblen Typen.
- Man kann also nicht so etwas wie `"str" + 5` machen.
- Beispiele für solche Sprachen sind Python, Java, und C.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

- Solche Sprachen verbieten die Interaktion zwischen inkompatiblen Typen.
- Man kann also nicht so etwas wie `"str" + 5` machen.
- Beispiele für solche Sprachen sind Python, Java, und C.

## Definition: Schwach Typisierte Sprache

In einer schwach typisierten (EN: *weakly-typed*) Programmiersprache kann der Typ eines Wertes automatisch nach den Erfordernissen der aktuellen Berechnung konvertiert werden.

# Starke und Schwache Typisierung



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder Wert einen unveränderlichen *Typ*.

- Solche Sprachen verbieten die Interaktion zwischen inkompatiblen Typen.
- Man kann also nicht so etwas wie `"str" + 5` machen.
- Beispiele für solche Sprachen sind Python, Java, und C.

## Definition: Schwach Typisierte Sprache

In einer schwach typisierten (EN: *weakly-typed*) Programmiersprache kann der Typ eines Wertes automatisch nach den Erfordernissen der aktuellen Berechnung konvertiert werden.

- In schwach typisierten Sprachen, Ausdrücke wie `"str" + 5` können funktionieren und könnten z. B. `"str5"` ergeben.



## Definition: Stark Typisierte Sprache

In einer stark typisierten (EN: *strongly-typed*) Programmiersprache hat jeder *Wert* einen unveränderlichen *Typ*.

- Man kann also nicht so etwas wie `"str" + 5` machen.
- Beispiele für solche Sprachen sind Python, Java, und C.

## Definition: Schwach Typisierte Sprache

In einer schwach typisierten (EN: *weakly-typed*) Programmiersprache kann der Typ eines Wertes automatisch nach den Erfordernissen der aktuellen Berechnung konvertiert werden.

- In schwach typisierten Sprachen, Ausdrücke wie `"str" + 5` können funktionieren und könnten z. B. `"str5"` ergeben.
- Ein bekanntes Beispiel ist JavaScript.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen festen *Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen festen *Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen festen *Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- C und Java sind statisch typisiert.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- C und Java sind statisch typisiert.
- Die ursprüngliche Kombination von starker und dynamischer Typisierung hat viele Vorteile für Python.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- C und Java sind statisch typisiert.
- Die ursprüngliche Kombination von starker und dynamischer Typisierung hat viele Vorteile für Python.
- Der Kode ist kurz, weil wir keine Datentypen für Variablen angeben müssen.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen festen *Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- Die ursprüngliche Kombination von starker und dynamischer Typisierung hat viele Vorteile für Python.
- Der Kode ist kurz, weil wir keine Datentypen für Variablen angeben müssen.
- Der Kode sieht eleganter aus und Programmieren wird einfacher.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- Der Kode ist kurz, weil wir keine Datentypen für Variablen angeben müssen.
- Der Kode sieht eleganter aus und Programmieren wird einfacher.
- Nun ja.



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- Der Kode ist kurz, weil wir keine Datentypen für Variablen angeben müssen.
- Der Kode sieht eleganter aus und Programmieren wird einfacher.
- Nun ja. Auf den ersten Blick...



## Definition: Statisch Typisierte Sprache

In einer statisch typisierten (EN: *statically-typed*) Programmiersprache hat jede *Variable* einen *festen Datentyp* und kann nur Werte dieses Typs aufnehmen.

## Definition: Dynamisch Typisierte Sprache

In einer dynamisch typisierten (EN: *dynamically-typed*) Programmiersprache hat eine *Variable* keinen expliziten Typ. Wir können Werte verschiedenen Typs in einer Variable speichern.

- Python ist eine dynamisch typisierte Sprache.
- Der Kode sieht eleganter aus und Programmieren wird einfacher.
- Nun ja. Auf den ersten Blick...
- Es gibt auch Probleme.



# Verwirrung mit Typen



# Verwirrung mit Typen

- Schauen wir uns mal ein Beispiel an.



# Verwirrung mit Typen



- Schauen wir uns mal ein Beispiel an.
- Wir deklarieren eine Variable `int_var` und speichern die Ganzzahl 8 darin.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Schauen wir uns mal ein Beispiel an.
- Wir deklarieren eine Variable `int_var` und speichern die Ganzzahl `8` darin.
- Wir updaten dann `int_var` in dem wir `int_var = int_var / 3` ausrechnen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Schauen wir uns mal ein Beispiel an.
- Wir deklarieren eine Variable `int_var` und speichern die Ganzzahl `8` darin.
- Wir updaten dann `int_var` in dem wir `int_var = int_var / 3` ausrechnen.
- Sie haben gelernt, dass `//` eine Ganzzahldivision mit einem `int`-Ergebnis durchführt, wohingegen die Division mit `/` immer einen `float` ergibt<sup>109</sup>.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Wir deklarieren eine Variable `int_var` und speichern die Ganzzahl `8` darin.
- Wir updaten dann `int_var` in dem wir `int_var = int_var / 3` ausrechnen.
- Sie haben gelernt, dass `//` eine Ganzzahldivision mit einem `int`-Ergebnis durchführt, wohingegen die Division mit `/` immer einen `float` ergibt<sup>109</sup>.
- Unsere Variable `int_var` beinhaltet jetzt einen `float`.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Wir updaten dann `int_var` in dem wir `int_var = int_var / 3` ausrechnen.
- Sie haben gelernt, dass `//` eine Ganzzahldivision mit einem `int`-Ergebnis durchführt, wohingegen die Division mit `/` immer einen `float` ergibt<sup>109</sup>.
- Unsere Variable `int_var` beinhaltet jetzt einen `float`.
- Aus Sicht von Python ist das total OK.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Sie haben gelernt, dass `//` eine Ganzzahldivision mit einem `int`-Ergebnis durchführt, wohingegen die Division mit `/` immer einen `float` ergibt<sup>109</sup>.
- Unsere Variable `int_var` beinhaltet jetzt einen `float`.
- Aus Sicht von Python ist das total OK.
- Der *Wert* der Variable hat immer einen festen Datentyp, denn die Sprache ist stark typisiert.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Unsere Variable `int_var` beinhaltet jetzt einen `float`.
- Aus Sicht von Python ist das total OK.
- Der Wert der Variable hat immer einen festen Datentyp, denn die Sprache ist stark typisiert.
- Der erste Wert der Variable hat den Datentyp `int`.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Aus Sicht von Python ist das total OK.
- Der Wert der Variable hat immer einen festen Datentyp, denn die Sprache ist stark typisiert.
- Der erste Wert der Variable hat den Datentyp `int`.
- Der nächste Wert der Variable ist ein `float`.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Der Wert der Variable hat immer einen festen Datentyp, denn die Sprache ist stark typisiert.
- Der erste Wert der Variable hat den Datentyp `int`.
- Der nächste Wert der Variable ist ein `float`.
- Die Variable selbst hat keinen festen Typ, denn die Sprache ist dynamisch typisiert.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Der erste Wert der Variable hat den Datentyp `int`.
- Der nächste Wert der Variable ist ein `float`.
- Die Variable selbst hat keinen festen Typ, denn die Sprache ist dynamisch typisiert.
- Es ist OK, einen `float` in einer Variable zu speichern die aktuell einen `int` beherbergt.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Der nächste Wert der Variable ist ein `float`.
- Die Variable selbst hat keinen festen Typ, denn die Sprache ist dynamisch typisiert.
- Es ist OK, einen `float` in einer Variable zu speichern die aktuell einen `int` beherbergt.
- Das Programm läuft ohne Fehler durch.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Die Variable selbst hat keinen festen Typ, denn die Sprache ist dynamisch typisiert.
- Es ist OK, einen `float` in einer Variable zu speichern die aktuell einen `int` beherbergt.
- Das Programm läuft ohne Fehler durch.
- Aus der Sicht eines Programmierers ist das Programm trotzdem einfach **falsch**.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Es ist OK, einen `float` in einer Variable zu speichern die aktuell einen `int` beherbergt.
- Das Programm läuft ohne Fehler durch.
- Aus der Sicht eines Programmierers ist das Programm trotzdem einfach **falsch**.
- Stellen Sie sich vor, dass das nicht einfach irgendein bedeutungsloses Beispiel wäre.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Das Programm läuft ohne Fehler durch.
- Aus der Sicht eines Programmierers ist das Program trotzdem einfach **falsch**.
- Stellen Sie sich vor, dass das nicht einfach irgendein bedeutungsloses Beispiel wäre.
- Stellen Sie sich vor, das wäre Teil eines echten nützlichen Programms.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Aus der Sicht eines Programmierers ist das Program trotzdem einfach **falsch**.
- Stellen Sie sich vor, dass das nicht einfach irgendein bedeutungsloses Beispiel wäre.
- Stellen Sie sich vor, das wäre Teil eines echten nützlichen Programms.
- Stellen Sie sich vor, Sie bekommen diesen Kode und sollen ihn verstehen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Stellen Sie sich vor, dass das nicht einfach irgendein bedeutungsloses Beispiel wäre.
- Stellen Sie sich vor, das wäre Teil eines echten nützlichen Programms.
- Stellen Sie sich vor, Sie bekommen diesen Kode und sollen ihn verstehen.
- Es ist gut möglich, dass Sie dabei übersehen, dass eine Variable names `int_var` nun einen `float` beinhaltet.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Stellen Sie sich vor, das wäre Teil eines echten nützlichen Programms.
- Stellen Sie sich vor, Sie bekommen diesen Kode und sollen ihn verstehen.
- Es ist gut möglich, dass Sie dabei übersehen, dass eine Variable names `int_var` nun einen `float` beinhaltet.
- Und wenn Sie es sehen, dann empfinden Sie es als komisch und verwirrend.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Stellen Sie sich vor, Sie bekommen diesen Kode und sollen ihn verstehen.
- Es ist gut möglich, dass Sie dabei übersehen, dass eine Variable names `int_var` nun einen `float` beinhaltet.
- Und wenn Sie es sehen, dann empfinden Sie es als komisch und verwirrend.
- Nehmen wir, es fällt Ihnen auf. Dann denken Sie: „Hm. Warum ist da ein `float` in der Variablen `int_var`?“

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Und wenn Sie es sehen, dann empfinden Sie es als komisch und verwirrend.
- Nehmen wir, es fällt Ihnen auf. Dann denken Sie: „Hm. Warum ist da ein `float` in der Variablen `int_var`?“
- Normalerweise erwarten wir, das Kode Sinn ergibt und die Namen für Dinge deren Natur vernünftig widerspiegeln.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Und wenn Sie es sehen, dann empfinden Sie es als komisch und verwirrend.
- Nehmen wir, es fällt Ihnen auf. Dann denken Sie: „Hm. Warum ist da ein `float` in der Variablen `int_var`?“
- Normalerweise erwarten wir, das Kode Sinn ergibt und die Namen für Dinge deren Natur vernünftig widerspiegeln.
- Es gibt also zwei mögliche Erklärungen für diese Situation.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Normalerweise erwarten wir, das Kode Sinn ergibt und die Namen für Dinge deren Natur vernünftig widerspiegeln.
- Es gibt also zwei mögliche Erklärungen für diese Situation.
- Erstens: Vielleicht hat der Autor des Kodes ja den `/`-Operator mit dem `//`-Operator verwechselt?

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Es gibt also zwei mögliche Erklärungen für diese Situation.
- Erstens: Vielleicht hat der Autor des Kodes ja den `/`-Operator mit dem `//`-Operator verwechselt?
- Vielleicht wollte er ja eine Ganzzahldivision machen und hat aus Versehen eine Fließkommadivision durchgeführt.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Erstens: Vielleicht hat der Autor des Kodes ja den `/`-Operator mit dem `//`-Operator verwechselt?
- Vielleicht wollte er ja eine Ganzzahldivision machen und hat aus Versehen eine Fließkommadivision durchgeführt.
- So ein Fehler könnte übrigens in einem großen Programm sehr sehr schwer zu finden sein...

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Vielleicht wollte er ja eine Ganzzahldivision machen und hat aus Versehen eine Fließkommadivision durchgeführt.
- So ein Fehler könnte übrigens in einem großen Programm sehr schwer zu finden sein...
- Zweitens: Vielleicht wollte der Autor ja eine Fließkommadivision machen und wollte einen `float` in `int_var` speichern und hat nur einen verwirrenden Namen gewählt.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Zweitens: Vielleicht wollte der Autor ja eine Fließkommadivision machen und wollte einen `float` in `int_var` speichern und hat nur einen verwirrenden Namen gewählt.
- Vielleicht habt er ja anfänglich eine Ganzzahldivision gemacht und einen `ints` in der Variable gespeichert.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Zweitens: Vielleicht wollte der Autor ja eine Fließkommadivision machen und wollte einen `float` in `int_var` speichern und hat nur einen verwirrenden Namen gewählt.
- Vielleicht habt er ja anfänglich eine Ganzzahldivision gemacht und einen `ints` in der Variable gespeichert.
- Vielleicht hat er später gemerkt, dass eine Fließkommadivision doch besser ist, den Operator getauscht, aber einfach vergessen, die Variable umzubenennen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Vielleicht habt er ja anfänglich eine Ganzzahldivision gemacht und einen `int`s in der Variable gespeichert.
- Vielleicht hat er später gemerkt, dass eine Fließkommadivision doch besser ist, den Operator getauscht, aber einfach vergessen, die Variable umzubenennen.
- Oder es war ihm einfach egal.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Vielleicht habt er ja anfänglich eine Ganzzahldivision gemacht und einen `ints` in der Variable gespeichert.
- Vielleicht hat er später gemerkt, dass eine Fließkommadivision doch besser ist, den Operator getauscht, aber einfach vergessen, die Variable umzubenennen.
- Oder es war ihm einfach egal.
- Durch das Beibehalten des Namens entstehen aber Gefahren!

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Vielleicht hat er später gemerkt, dass eine Fließkommadivision doch besser ist, den Operator getauscht, aber einfach vergessen, die Variable umzubenennen.
- Oder es war ihm einfach egal.
- Durch das Beibehalten des Namens entstehen aber Gefahren!
- Wenn jetzt ein anderer Programmierer an dem Kode arbeitet, dann könnte der aus dem Variablenamen schlussfolgern, dass dort ein `int` drin gespeichert ist.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Durch das Beibehalten des Namens entstehen aber Gefahren!
- Wenn jetzt ein anderer Programmierer an dem Kode arbeitet, dann könnte der aus dem Variablenamen schlussfolgern, dass dort ein `int` drin gespeichert ist.
- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>."  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>."
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Wenn jetzt ein anderer Programmierer an dem Kode arbeitet, dann könnte der aus dem Variablenamen schlussfolgern, dass dort ein `int` drin gespeichert ist.
- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.
- Das könnte gehen, wenn `int_var` ein `int` wäre, crashed aber, weil es ein `float` ist.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.
- Das könnte gehen, wenn `int_var` ein `int` wäre, crashed aber, weil es ein `float` ist.

## Gute Praxis

Die Namen, die wir im Programmcode verwenden, sollten klar unsere Intentionen widerspiegeln.



# Verwirrung mit Typen

- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.
- Das könnte gehen, wenn `int_var` ein `int` wäre, crashed aber, weil es ein `float` ist.
- Hätte der Programmierer diese Best-Practice beachtet, dann könnte es nur einen Grund geben, weshalb ein `float` in der Variable `int_var` gelandet ist.

## Gute Praxis

Die Namen, die wir im Programmcode verwenden, sollten klar unsere Intentionen widerspiegeln.



# Verwirrung mit Typen

- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.
- Das könnte gehen, wenn `int_var` ein `int` wäre, crashed aber, weil es ein `float` ist.
- Hätte der Programmierer diese Best-Practice beachtet, dann könnte es nur einen Grund geben, weshalb ein `float` in der Variable `int_var` gelandet ist: Es war ein Fehler!

## Gute Praxis

Die Namen, die wir im Programmcode verwenden, sollten klar unsere Intentionen widerspiegeln.



# Verwirrung mit Typen

- Da aber stattdessen ein `float` drin ist, können viele Fehler entstehen.
- Vielleicht versucht der nächste Programmierer einen String mit `int_var` zu indizieren, z. B. `"abcdefg"[int_var]`.
- Das könnte gehen, wenn `int_var` ein `int` wäre, crashed aber, weil es ein `float` ist.
- Hätte der Programmierer diese Best-Practice beachtet, dann könnte es nur einen Grund geben, weshalb ein `float` in der Variable `int_var` gelandet ist: Es war ein Fehler!
- Der wahrscheinlichste Grund wäre dann eine Verwechslung der `/`- und `//`-Operatoren.

## Gute Praxis

Die Namen, die wir im Programmcode verwenden, sollten klar unsere Intentionen widerspiegeln.

# Verwirrung mit Typen



- Hätte der Programmierer diese Best-Practice beachtet, dann könnte es nur einen Grund geben, weshalb ein `float` in der Variable `int_var` gelandet ist: Es war ein Fehler!
- Der wahrscheinlichste Grund wäre dann eine Verwechslung der `/`- und `//`-Operatoren.
- So oder so, Sie stimmen mir sicherlich zu, dass etwas mit dem Programm nicht stimmt.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Der wahrscheinlichste Grund wäre dann eine Verwechslung der `/`- und `//`-Operatoren.
- So oder so, Sie stimmen mir sicherlich zu, dass etwas mit dem Programm nicht stimmt.
- Leider sind wir nicht die Autoren des Programms, also wissen wir nicht, was falsch ist.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- So oder so, Sie stimmen mir sicherlich zu, dass etwas mit dem Programm nicht stimmt.
- Leider sind wir nicht die Autoren des Programms, also wissen wir nicht, was falsch ist.
- Dieser Kode wird also später Fehler verursachen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```



# Verwirrung mit Typen

- Leider sind wir nicht die Autoren des Programms, also wissen wir nicht, was falsch ist.
- Dieser Kode wird also später Fehler verursachen.
- Viele solche Probleme existieren in vielen Softwareprojekten – und sie sind schwer zu finden<sup>41</sup>.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Dieser Kode wird also später Fehler verursachen.
- Viele solche Probleme existieren in vielen Softwareprojekten – und sie sind schwer zu finden<sup>41</sup>.
- Hier wird es zum Problem, dass wir in Python die Datentypen nicht explizit spezifizieren müssen.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Viele solche Probleme existieren in vielen Softwareprojekten – und sie sind schwer zu finden<sup>41</sup>.
- Hier wird es zum Problem, dass wir in Python die Datentypen nicht explizit spezifizieren müssen.
- Diese ganze Gruppe von Problem resultiert daraus, das Python dynamisch typisiert ist.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Hier wird es zum Problem, dass wir in Python die Datentypen nicht explizit spezifizieren müssen.
- Diese ganze Gruppe von Problem resultiert daraus, das Python dynamisch typisiert ist.
- Jeder von uns macht solche Fehler.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Diese ganze Gruppe von Problem resultiert daraus, das Python dynamisch typisiert ist.
- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.
- Es gibt aber zwei Dinge, die wir tun können, um zu verhindern, dass diese Fehler durch unsere Qualitätskontrolle durchsickern.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.
- Es gibt aber zwei Dinge, die wir tun können, um zu verhindern, dass diese Fehler durch unsere Qualitätskontrolle durchsickern:
  1. Wir können statische Typchecker (EN: *static type checkers*) verwenden, also Werkzeuge, die existieren um solche Fehler zu finden.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ python3 variable\_types\_wrong.py ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Verwirrung mit Typen



- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.
- Es gibt aber zwei Dinge, die wir tun können, um zu verhindern, dass diese Fehler durch unsere Qualitätskontrolle durchsickern:
  1. Wir können statische Typchecker (EN: *static type checkers*) verwenden, also Werkzeuge, die existieren um solche Fehler zu finden.
  2. Wir können Type Hints verwenden, um unsere Variablen mit Datentypen zu annotieren und um *a)* unsere Intentionen klarer zu machen und *b)* die Typchecker zu unterstützen.

# Verwirrung mit Typen



- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.
- Es gibt aber zwei Dinge, die wir tun können, um zu verhindern, dass diese Fehler durch unsere Qualitätskontrolle durchsickern:
  1. Wir können statische Typchecker (EN: *static type checkers*) verwenden, also Werkzeuge, die existieren um solche Fehler zu finden.
  2. Wir können Type Hints verwenden, um unsere Variablen mit Datentypen zu annotieren und um *a)* unsere Intentionen klarer zu machen und *b)* die Typchecker zu unterstützen.
- Und in meinem Kurs machen Sie besser beides.

# Verwirrung mit Typen



- Jeder von uns macht solche Fehler.
- Wir können das gar nicht verhindern.
- Es gibt aber zwei Dinge, die wir tun können, um zu verhindern, dass diese Fehler durch unsere Qualitätskontrolle durchsickern:
  1. Wir können statische Typchecker (EN: *static type checkers*) verwenden, also Werkzeuge, die existieren um solche Fehler zu finden.
  2. Wir können Type Hints verwenden, um unsere Variablen mit Datentypen zu annotieren und um *a)* unsere Intentionen klarer zu machen und *b)* die Typchecker zu unterstützen.
- Und in meinem Kurs machen Sie besser beides.
- Und das gucken wir uns jetzt an.



# Static Type Checker: Mypy





## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.



## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmkode Mehrdeutigkeiten oder Fehler beinhaltet.



## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmkode Mehrdeutigkeiten oder Fehler beinhaltet.
- In Programmiersprachen wie C und Java, kann das der Kompiler schon zu einem Teil erledigen, denn diese Sprachen sind statisch und stark typisiert.



## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmcode Mehrdeutigkeiten oder Fehler beinhaltet.
- In Programmiersprachen wie C und Java, kann das der Kompiler schon zu einem Teil erledigen, denn diese Sprachen sind statisch und stark typisiert.
- In Python, das dynamisch typisiert ist, benutzen wir Werkzeuge wie Mypy<sup>46</sup>.



## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmcode Mehrdeutigkeiten oder Fehler beinhaltet.
- In Programmiersprachen wie C und Java, kann das der Kompiler schon zu einem Teil erledigen, denn diese Sprachen sind statisch und stark typisiert.
- In Python, das dynamisch typisiert ist, benutzen wir Werkzeuge wie Mypy<sup>46</sup>.
- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu **Ctrl**+**Alt**+**T** drücken und unter Microsoft Windows mit Druck auf **Windows**+**R**, dann Schreiben von `cmd`, dann Druck auf **↓**.



## Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmcode Mehrdeutigkeiten oder Fehler beinhaltet.
- In Programmiersprachen wie C und Java, kann das der Kompiler schon zu einem Teil erledigen, denn diese Sprachen sind statisch und stark typisiert.
- In Python, das dynamisch typisiert ist, benutzen wir Werkzeuge wie Mypy<sup>46</sup>.
- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu **Ctrl**+**Alt**+**T** drücken und unter Microsoft Windows mit Druck auf **Windows**+**R**, dann Schreiben von `cmd`, dann Druck auf **Enter**.
- Sie würden dann `pip install mypy` eintippen und **Enter** drücken.



# Mypy installieren

- Der erste Schritt, weniger Typ-bezogene Fehler zu machen, ist natürlich vorsichtiges Programmieren.
- Der zweite Schritt ist es, Werkzeuge einzusetzen, die prüfen, ob Programmcode Mehrdeutigkeiten oder Fehler beinhaltet.
- In Programmiersprachen wie C und Java, kann das der Kompiler schon zu einem Teil erledigen, denn diese Sprachen sind statisch und stark typisiert.
- In Python, das dynamisch typisiert ist, benutzen wir Werkzeuge wie Mypy<sup>46</sup>.
- Um dieses Programm zu installieren, öffnen Sie ein Terminal, in dem Sie unter Ubuntu **Ctrl**+**Alt**+**T** drücken und unter Microsoft Windows mit Druck auf **Windows**+**R**, dann Schreiben von `cmd`, dann Druck auf **Enter**.
- Sie würden dann `pip install mypy` eintippen und **Enter** drücken.
- Normalerweise machen Sie dass unter einem virtuellen Environment, was wir später diskutieren.



# Mypy installieren

- Um dieses Programm zu installieren, öffnen Sie ein Terminal.
- Sie würden dann `pip install mypy` eintippen und drücken.
- Normalerweise machen Sie dass unter einem virtuellen Environment.
- So oder so, Mypy wird installiert.

```
tweise@weise-laptop:~/tmp$ pip install mypy
Defaulting to user installation because normal site-packages is not writeable
Collecting mypy
  Using cached mypy-1.11.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_6
4.manylinux_2_28_x86_64.whl.metadata (1.9 kB)
Requirement already satisfied: typing-extensions>=4.6.0 in /home/tweise/.local/l
ib/python3.10/site-packages (from mypy) (4.7.1)
Requirement already satisfied: mypy-extensions>=1.0.0 in /home/tweise/.local/lib
/python3.10/site-packages (from mypy) (1.0.0)
Requirement already satisfied: tomli>=1.1.0 in /home/tweise/.local/lib/python3.1
0/site-packages (from mypy) (2.0.1)
Downloading mypy-1.11.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_6
4.manylinux_2_28_x86_64.whl (12.5 MB)
12.5/12.5 MB 139.7 kB/s eta 0:00:00
WARNING: Error parsing dependencies of pdfminer-six: Invalid version: '-VERSION-'
Installing collected packages: mypy
Successfully installed mypy-1.11.1
tweise@weise-laptop:~/tmp$
```

# Code mit Mypy Prüfen

- Jetzt können wir Mypy auf das Programm `variable_types_wrong.py` anwenden.



## Code mit Mypy Prüfen

- Jetzt können wir Mypy auf das Programm `variable_types_wrong.py` anwenden.
- Wir rufen Mypy dazu im Terminal auf und geben den Name des zu prüfenden Programms als Parameter an, sowie ein paar zusätzliche Parameter.

## Code mit Mypy Prüfen

- Jetzt können wir Mypy auf das Programm `variable_types_wrong.py` anwenden.
- Wir rufen Mypy dazu im Terminal auf und geben den Name des zu prüfenden Programms als Parameter an, sowie ein paar zusätzliche Parameter.
- Wir rufen

```
mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
```

 auf.

# Code mit Mypy Prüfen



- Jetzt können wir Mypy auf das Programm `variable_types_wrong.py` anwenden.
- Wir rufen Mypy dazu im Terminal auf und geben den Name des zu prüfenden Programms als Parameter an, sowie ein paar zusätzliche Parameter.
- Wir rufen

```
mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
```

- Wenn das nicht geht, dann versuchen Sie es mit

```
python3 -m mypy variable_types_wrong.py  
--no-strict-optional --check-untyped-defs
```

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs  
2 variable_types_wrong.py:4: error: Incompatible types in assignment (  
    ↪ expression has type "float", variable has type "int") [assignment  
    ↪ ]  
3 Found 1 error in 1 file (checked 1 source file)  
4 # mypy 1.19.1 failed with exit code 1.
```

# Code mit Mypy Prüfen



- Wir rufen Mypy dazu im Terminal auf und geben den Name des zu prüfenden Programms als Parameter an, sowie ein paar zusätzliche Parameter.

- Wir rufen

```
mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
```

 auf.

- Wenn das nicht geht, dann versuchen Sie es mit

```
python3 -m mypy variable_types_wrong.py  
--no-strict-optional --check-untyped-defs
```

- Tatsächlich: Mypy sagt uns, dass etwas komisches in der vierten Zeile passiert, also bei  
`int_var = int_var / 3.`

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs  
2 variable_types_wrong.py:4: error: Incompatible types in assignment (  
    ↪ expression has type "float", variable has type "int") [assignment  
    ↪ ]  
3 Found 1 error in 1 file (checked 1 source file)  
4 # mypy 1.19.1 failed with exit code 1.
```



# Code mit Mypy Prüfen

- Wir rufen

```
mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
```

- Wenn das nicht geht, dann versuchen Sie es mit

```
python3 -m mypy variable_types_wrong.py  
--no-strict-optional --check-untyped-defs
```

- Tatsächlich: Mypy sagt uns, dass etwas komisches in der vierten Zeile passiert, also bei

```
int_var = int_var / 3.
```

- Mypy schlägt mit dem Exit-Kode 1 fehl.

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs  
2 variable_types_wrong.py:4: error: Incompatible types in assignment (  
    ↪ expression has type "float", variable has type "int") [assignment  
    ↪ ]  
3 Found 1 error in 1 file (checked 1 source file)  
4 # mypy 1.19.1 failed with exit code 1.
```



# Code mit Mypy Prüfen

- Wenn das nicht geht, dann versuchen Sie es mit  
`python3 -m mypy variable_types_wrong.py  
--no-strict-optional --check-untyped-defs`
- Tatsächlich: Mypy sagt uns, dass etwas komisches in der vierten Zeile passiert, also bei  
`int_var = int_var / 3.`
- Mypy schlägt mit dem Exit-Kode 1 fehl.
- Programme beenden sich normalerweise mit dem Exit-Kode 0 wenn sie fehlerlos abgelaufen sind.

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
2 variable_types_wrong.py:4: error: Incompatible types in assignment (
3     ↪ expression has type "float", variable has type "int") [assignment
4     ↪ ]
5 Found 1 error in 1 file (checked 1 source file)
6 # mypy 1.19.1 failed with exit code 1.
```

# Code mit Mypy Prüfen

- Tatsächlich: Mypy sagt uns, dass etwas komisches in der vierten Zeile passiert, also bei `int_var = int_var / 3.`
- Mypy schlägt mit dem Exit-Kode `1` fehl.
- Programme beenden sich normalerweise mit dem Exit-Kode `0` wenn sie fehlerlos abgelaufen sind.
- Ein Rückgabewert, der nicht `0` ist, zeugt von einem Fehler.

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
2 variable_types_wrong.py:4: error: Incompatible types in assignment (
3     ↪ expression has type "float", variable has type "int") [assignment
4     ↪ ]
5 Found 1 error in 1 file (checked 1 source file)
6 # mypy 1.19.1 failed with exit code 1.
```



# Code mit Mypy Prüfen

- Mypy schlägt mit dem Exit-Kode 1 fehl.
- Programme beenden sich normalerweise mit dem Exit-Kode 0 wenn sie fehlerlos abgelaufen sind.
- Ein Rückgabewert, der nicht 0 ist, zeugt von einem Fehler.
- Und Mypy hat tatsächlich einen Fehler gefunden.

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
2 variable_types_wrong.py:4: error: Incompatible types in assignment (
3     ↪ expression has type "float", variable has type "int")  [assignment
4     ↪ ]
5 Found 1 error in 1 file (checked 1 source file)
6 # mypy 1.19.1 failed with exit code 1.
```

## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python.



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist.



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren.



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren. Sie können es dann mit dem Kommando `mypy fileToScan.py` anwenden, wobei `fileToScan.py` der Name der zu prüfenden Datei ist (es kann auch ein Verzeichnis angegeben werden).



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren. Sie können es dann mit dem Kommando `mypy fileToScan.py` anwenden, wobei `fileToScan.py` der Name der zu prüfenden Datei ist (es kann auch ein Verzeichnis angegeben werden).

- Wenn wir Mypy auf das korrekte (jedoch nutzlose) Programm `variable_types.py` anwenden, dann sagt es uns, dass kein Fehler vorliegt.

```
1 $ mypy variable_types.py --no-strict-optional --check-untyped-defs
2 Success: no issues found in 1 source file
3 # mypy 1.19.1 succeeded with exit code 0.
```



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren. Sie können es dann mit dem Kommando `mypy fileToScan.py` anwenden, wobei `fileToScan.py` der Name der zu prüfenden Datei ist (es kann auch ein Verzeichnis angegeben werden).

- Wenn wir Mypy auf das korrekte (jedoch nutzlose) Programm `variable_types.py` anwenden, dann sagt es uns, dass kein Fehler vorliegt.
- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren. Sie können es dann mit dem Kommando `mypy fileToScan.py` anwenden, wobei `fileToScan.py` der Name der zu prüfenden Datei ist (es kann auch ein Verzeichnis angegeben werden).

- Wenn wir Mypy auf das korrekte (jedoch nutzlose) Programm `variable_types.py` anwenden, dann sagt es uns, dass kein Fehler vorliegt.
- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.



## Nützliches Werkzeug

Mypy<sup>46</sup> ist ein statischer Type-Checker für Python. Dieses Werkzeug kann Sie warnen wenn Sie, z. B. einen Wert einer Variable zuweisen, der einen anderen Datentyp hat als der vorher in der gespeicherte Wert – was ein potentieller Programmierfehler ist. Sie können Mypy via `pip install mypy` installieren. Sie können es dann mit dem Kommando `mypy fileToScan.py` anwenden, wobei `fileToScan.py` der Name der zu prüfenden Datei ist (es kann auch ein Verzeichnis angegeben werden).

- Wenn wir Mypy auf das korrekte (jedoch nutzlose) Programm `variable_types.py` anwenden, dann sagt es uns, dass kein Fehler vorliegt.
- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.
- Mypy führt unseren Kode auch nicht aus.

# Mypy

- Wenn wir Mypy auf das korrekte (jedoch nutzlose) Programm `variable_types.py` anwenden, dann sagt es uns, dass kein Fehler vorliegt.
- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.
- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.



## Mypy

- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.
- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.



## Mypy

- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.
- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.



## Mypy

- Wir haben jetzt also ein Werkzeug in der Hand, mit dem wir unseren Quelltext auf Type-bezogene Fehler prüfen können.
- Mypy verändert unseren Kode nicht.
- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen. Jeder mögliche Fehler, der mit so einem Tool gefunden wird, muss beseitigt werden.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen. Jeder mögliche Fehler, der mit so einem Tool gefunden wird, muss beseitigt werden. In anderen Worten: Type-checken Sie jedes Programm.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen. Jeder mögliche Fehler, der mit so einem Tool gefunden wird, muss beseitigt werden. In anderen Worten: Type-checken Sie jedes Programm. Beseitigen Sie alle Fehler.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen. Jeder mögliche Fehler, der mit so einem Tool gefunden wird, muss beseitigt werden. In anderen Worten: Type-checken Sie jedes Programm. Beseitigen Sie alle Fehler. Und Type-checken Sie es nochmal.



# Mypy

- Mypy führt unseren Kode auch nicht aus.
- Mypy liest nur den Kode ein und sucht nach möglichen Fehlern.
- Es hat also keinen Einfluss auf die Performanz oder Geschwindigkeit unseres Programms.
- Es kann keine Fehler korrigieren, denn es weiß ja nicht, was der Programmierer eigentlich vor hatte.
- Aber zu wissen, dass Zeile 4 in `variable_types_wrong.py` wahrscheinlich falsch ist, kann uns schon sehr viel helfen, mögliche Fehler zu finden und **selbst** zu korrigieren.

## Gute Praxis

Jedes Programm muss statische Typ-Prüfung mit Werkzeugen wie Mypy überstehen. Jeder mögliche Fehler, der mit so einem Tool gefunden wird, muss beseitigt werden. In anderen Worten: Type-checken Sie jedes Programm. Beseitigen Sie alle Fehler. Und Type-checken Sie es nochmal. Bis nichts mehr gefunden wird.

# Weiteres Beispiel

- Wenden wir nun Mypy auf das Beispielprogramm `assignment_wrong.py` aus der vorigen Einheit an.

```
1 # We define a variable named "int_var" and assign the int value 1 to it.
2 int_var = 1
3
4 # We can use the variable int_var in computations like any other value.
5 print(2 + int_var) # This should print 2 + int_var = 2 + 1 = 3.
6
7 # We can also use the variable in f-strings.
8 print(f"int_var has value {int_var}.") # prints 'int_var has value 1.'
9
10 # We can also change the value of the variable.
11 int_var = (3 * int_var) + 1 # int_var = (3 * 1) + 1 = 4
12 print(f"int_var is now {intvar}.") # prints 'int_var is now 4.'
13
14 float_var = 3.5 # Ofcourse we can also use floating point numbers.
15 print(f"float_var has value {float_var}.") # 'float_var has value 3.5.'
16
17 new_var = float_var * int_var # new_var = 3.5 * 4 = 14.0 <- a float!
18 print(f"{new_var = })."
```

## Weiteres Beispiel

- Wenden wir nun Mypy auf das Beispielprogramm `assignment_wrong.py` aus der vorigen Einheit an.
- Es sagt uns „*Name 'intvar' is not defined.*“

```
1 $ mypy assignment_wrong.py --no-strict-optional --check-untyped-defs
2 assignment_wrong.py:12: error: Name "intvar" is not defined [name-
   ↪ defined]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 1.19.1 failed with exit code 1.
```

## Weiteres Beispiel



- Wenden wir nun Mypy auf das Beispielprogramm `assignment_wrong.py` aus der vorigen Einheit an.
- Es sagt uns „*Name 'intvar' is not defined.*“
- Mit dem IDE und Mypy haben wir nun zwei unabhängige Werzeuge die uns bei der Fehlersuche im Kode helfen können.

```
1 $ mypy assignment_wrong.py --no-strict-optional --check-untyped-defs
2 assignment_wrong.py:12: error: Name "intvar" is not defined [name-
   ↪ defined]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 1.19.1 failed with exit code 1.
```

## Weiteres Beispiel



- Wenden wir nun Mypy auf das Beispielprogramm `assignment_wrong.py` aus der vorigen Einheit an.
- Es sagt uns „*Name 'intvar' is not defined.*“
- Mit dem IDE und Mypy haben wir nun zwei unabhängige Werkzeuge die uns bei der Fehlersuche im Kode helfen können.
- Je mehr solche Werkzeuge wir nutzen, desto wahrscheinlicher ist es, dass wir fehlerfreien Kode produzieren.

```
1 $ mypy assignment_wrong.py --no-strict-optional --check-untyped-defs
2 assignment_wrong.py:12: error: Name "intvar" is not defined [name-
   ↪ defined]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 1.19.1 failed with exit code 1.
```

# Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software.*“



# Mypy und andere Werkzeuge



- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren.*“

# Mypy und andere Werkzeuge



- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen.*“

# Mypy und andere Werkzeuge



- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen.*“



## Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit.“*



## Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit. Warum muss das sein?*“



## Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit. Warum muss das sein?*“
- Es gibt zwei Antworten darauf.



# Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit. Warum muss das sein?*“
- Es gibt zwei Antworten darauf.
- Erstens: „*This will improve the quality of your code.*“



# Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit. Warum muss das sein?*“
- Es gibt zwei Antworten darauf.
- Erstens: „*This will improve the quality of your code.*“
- Und zweitens... .



# Mypy und andere Werkzeuge

- Sie denken jetzt vielleicht: „*Mypy ist zusätzliche Software. Wenn ich es benutzen will, dann muss ich es installieren. Dann muss ich es und seine Kommandozeilenparameter lernen. Jedes Mal, wenn ich es benutzen will, dann muss ich das Terminal öffnen, in das richtige Verzeichnis gehen, Mypy ausführen, und die Ausgabe lesen. Das ist ... ziemlich viel Arbeit. Warum muss das sein?*“
- Es gibt zwei Antworten darauf.
- Erstens: „*This will improve the quality of your code.*“
- Und zweitens... .

## Gute Praxis

Ein professioneller Softwareingenieur oder Programmierer kennt viele Werkzeuge und freut sich immer, ein neues Werkzeug zu erlernen.

# Mypy und andere Werkzeuge



- Es gibt zwei Antworten darauf.
- Erstens: „*This will improve the quality of your code.*“
- Und zweitens...

## Gute Praxis

Ein professioneller Softwareingenieur oder Programmierer kennt viele Werkzeuge und freut sich immer, ein neues Werkzeug zu erlernen.

- In Ihrem professionellen Leben werden Sie Dutzende wenn nicht Hunderte von Werkzeugen auf verschiedenen Betriebssystemen erlernen.

# Mypy und andere Werkzeuge



- Erstens: „*This will improve the quality of your code.*“
- Und zweitens... .

## Gute Praxis

Ein professioneller Softwareingenieur oder Programmierer kennt viele Werkzeuge und freut sich immer, ein neues Werkzeug zu erlernen.

- In Ihrem professionellen Leben werden Sie Dutzende wenn nicht Hunderte von Werkzeugen auf verschiedenen Betriebssystemen erlernen.
- Je mehr Werkzeuge Sie bereits kennen, desto einfacher wird es, neue Werkzeuge zu lernen.

# Mypy und andere Werkzeuge



- Und zweitens...

## Gute Praxis

Ein professioneller Softwareingenieur oder Programmierer kennt viele Werkzeuge und freut sich immer, ein neues Werkzeug zu erlernen.

- In Ihrem professionellen Leben werden Sie Dutzende wenn nicht Hunderte von Werkzeugen auf verschiedenen Betriebssystemen erlernen.
- Je mehr Werkzeuge Sie bereits kennen, desto einfacher wird es, neue Werkzeuge zu lernen.
- Heutige automatisierte Builds in Continuous Integration-Umgebungen involvieren oftmals schon Dutzende von Programmen.



## Gute Praxis

Ein professioneller Softwareingenieur oder Programmierer kennt viele Werkzeuge und freut sich immer, ein neues Werkzeug zu erlernen.

- In Ihrem professionellen Leben werden Sie Dutzende wenn nicht Hunderte von Werkzeugen auf verschiedenen Betriebssystemen erlernen.
- Je mehr Werkzeuge Sie bereits kennen, desto einfacher wird es, neue Werkzeuge zu lernen.
- Heutige automatisierte Builds in Continuous Integration-Umgebungen involvieren oftmals schon Dutzende von Programmen.
- In der Lage zu seien, neue Werkzeuge zu erlernen und benutzen zu können ist eine essentielle Fähigkeit.



# Type Hints



## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Aber eben nicht *warum*.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Aber eben nicht *warum*.
- Komischerweise existiert dieses Problem nur zu einem viel kleineren Grad in einer statisch typisierten Sprache wie C.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Aber eben nicht *warum*.
- Komischerweise existiert dieses Problem nur zu einem viel kleineren Grad in einer statisch typisierten Sprache wie C.
- Hier müssen wir nämlich den Datentyp jeder Variable definieren, bevor wir ihr einen Wert zuweisen können.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Aber eben nicht *warum*.
- Komischerweise existiert dieses Problem nur zu einem viel kleineren Grad in einer statisch typisierten Sprache wie C.
- Hier müssen wir nämlich den Datentyp jeder Variable definieren, bevor wir ihr einen Wert zuweisen können.
- Wenn der Programmierer gewollt hätte, das `int_var` nur Ganzzahlen beinhalten kann, dann hätte der die Variable entsprechend deklariert.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Aber eben nicht *warum*.
- Komischerweise existiert dieses Problem nur zu einem viel kleineren Grad in einer statisch typisierten Sprache wie C.
- Hier müssen wir nämlich den Datentyp jeder Variable definieren, bevor wir ihr einen Wert zuweisen können.
- Wenn der Programmierer gewollt hätte, das `int_var` nur Ganzzahlen beinhalten kann, dann hätte der die Variable entsprechend deklariert.
- Wenn er einen `float` hätte darin speichern wollen, dann hätte er die Variable eben als „vom Typ `float`“ deklariert.

## Nicht Genug

- Wir hatten gesagt, dass es zwei Gründe für den Fehler in `variable_types_wrong.py` geben kann:
  1. Der Programmierer hat aus Versehen die Operatoren `/` und `//` verwechselt,
  2. Oder er hat einen schlechten Namen für die Variable `int_var` gewählt.
- Ein Type-Checking Werkzeug kann nicht wissen, was die Intention des Programmierers war.
- Es kann nur herausfinden, dass Zeile 4 wahrscheinlich falsch ist, weil wir in `int_var` erst einen `int` und später einen `float` gespeichert haben.
- Komischerweise existiert dieses Problem nur zu einem viel kleineren Grad in einer statisch typisierten Sprache wie C.
- Hier müssen wir nämlich den Datentyp jeder Variable definieren, bevor wir ihr einen Wert zuweisen können.
- Wenn der Programmierer gewollt hätte, das `int_var` nur Ganzzahlen beinhalten kann, dann hätte der die Variable entsprechend deklariert.
- Wenn er einen `float` hätte darin speichern wollen, dann hätte er die Variable eben als „vom Typ `float`“ deklariert.
- Der Kompiler hätte den Fehler dann sofort gesehen und gewusst, ob Zeile 10 falsch ist oder ob die Ganzzahl in Zeile 1 einfach als Fließkommazahl zu interpretieren ist.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...



# Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.



## Type Hints

- Wie gesagt, hier erwischen uns die Nachteile dynamisch typisierter Sprachen wie Python.
- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.



## Type Hints

- Für kleine Projekte sind sie angenehm.
- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.



## Type Hints

- Aber wenn die Projekte größer werden, rutschen wir in ein Durcheinander.
- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.



## Type Hints

- Und natürlich sind die meisten Programme viel viel komplexer als unsere kleinen Beispiele hier.
- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, das dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.
- Wenn wir eine Variable deklarieren, z. B. `my_var = 1` und explizit festlegen wollen, dass diese nur Ganzzahlen aufnimmt, dann können wir schreiben `my_var: int = 1`.



# Type Hints

- Stellen Sie sich vor, dass sie Tausende Zeilen Kode durchsuchen, um herauszufinden, welchen Typ eine Variable haben soll und warum.
- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, das dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.
- Wenn wir eine Variable deklarieren, z. B. `my_var = 1` und explizit festlegen wollen, dass diese nur Ganzzahlen aufnimmt, dann können wir schreiben `my_var: int = 1`.
- Ein Type Checker würde natürlich schon sehen, dass `my_var` Ganzzahlen aufnimmt, weil wir ja den Wert `1` darin speichern.



# Type Hints

- Und während Sie das tun, bedenken Sie, dass wir in Python Variablen jederzeit mit Werten anderer Datentypen überschreiben dürfen...
- Die Erkenntnis, dass dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.
- Wenn wir eine Variable deklarieren, z. B. `my_var = 1` und explizit festlegen wollen, dass diese nur Ganzzahlen aufnimmt, dann können wir schreiben `my_var: int = 1`.
- Ein Type Checker würde natürlich schon sehen, dass `my_var` Ganzzahlen aufnimmt, weil wir ja den Wert `1` darin speichern.
- Aber in dem wir `: int` nach dem Name schreiben, definieren wir auch, dass dies unsere Intention ist, dass `my_var` eine Variable ist, in der nur `ints` gespeicher werden sollen.



# Type Hints

- Die Erkenntnis, das dynamische Typisierung sowohl ein Segen als auch ein Problem seien kann, führte zu der Entwicklung von **optionalen** so genannten Type Hints<sup>72,97</sup>.
- Python **kann** statisches Typisieren unterstützen<sup>92</sup>.
- Wir können den Datentyp einer Variable deklarieren, *wenn wir das wollen*.
- Somit wird das obige Problem im Grunde gelöst: Wir können unsere Intentionen im Kode definieren.
- Wenn wir eine Variable deklarieren, z. B. `my_var = 1` und explizit festlegen wollen, dass diese nur Ganzzahlen aufnimmt, dann können wir schreiben `my_var: int = 1`.
- Ein Type Checker würde natürlich schon sehen, dass `my_var` Ganzzahlen aufnimmt, weil wir ja den Wert `1` darin speichern.
- Aber in dem wir `: int` nach dem Name schreiben, definieren wir auch, dass dies unsere Intention ist, dass `my_var` eine Variable ist, in der nur `ints` gespeicher werden sollen.
- Und das macht einen Unterschied.

# Probieren wir das mal aus

- Wie gesagt, es gibt zwei mögliche Intentionen, die der Programmierer von `variable_types_wrong.py` hätte gehabt haben können.



# Probieren wir das mal aus



- Wie gesagt, es gibt zwei mögliche Intentionen, die der Programmierer von `variable_types_wrong.py` hätte gehabt haben können.
- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.

```
1 int_var: int = 1 + 7 # Create a variable hinted as integer.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_1.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Probieren wir das mal aus



- Wie gesagt, es gibt zwei mögliche Intentionen, die der Programmierer von `variable_types_wrong.py` hätte haben können.
- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- Er würde sie mit `: int` annotieren.

```
1 int_var: int = 1 + 7 # Create a variable hinted as integer.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_1.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- Er würde sie mit `: int` annotieren.
- Er kann das Programm trotzdem ausführen, denn dem Python-Interpreter sind Type Hints egal.

```
1 int_var: int = 1 + 7 # Create a variable hinted as integer.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 int_var = int_var / 3 # / returns a float, but we may expect an int?  
5 print(type(int_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_1.py` ↓

```
1 <class 'int'>  
2 <class 'float'>
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- Er würde sie mit `: int` annotieren.
- Er kann das Programm trotzdem ausführen, denn dem Python-Interpreter sind Type Hints egal.
- Mypy aber nicht: Diesmal ist klar, dass ein Fehler gemacht wurde und warum.

```
1 $ mypy variable_types_wrong_hints_1.py --no-strict-optional --check-
  ↪ untyped-defs
2 variable_types_wrong_hints_1.py:4: error: Incompatible types in
  ↪ assignment (expression has type "float", variable has type "int")
  ↪ [assignment]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 1.19.1 failed with exit code 1.
```



# Probieren wir das mal aus

- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.

```
1 my_var: int | float = 1 + 7 # A variable hinted as either int or float.
2 print(type(my_var)) # This prints "<class 'int'>".
3
4 my_var = my_var / 3 # / returns a float, which is OK.
5 print(type(my_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_2.py` ↓

```
1 <class 'int'>
2 <class 'float'>
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Hier annotiert er die Variable mit `: int | float`, was bedeutet, dass entweder ein `int` oder ein `float` in ihr gespeichert werden darf.

```
1 my_var: int | float = 1 + 7 # A variable hinted as either int or float.
2 print(type(my_var)) # This prints "<class 'int'>".
3
4 my_var = my_var / 3 # / returns a float, which is OK.
5 print(type(my_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_2.py` ↓

```
1 <class 'int'>
2 <class 'float'>
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Hier annotiert er die Variable mit `: int | float`, was bedeutet, dass entweder ein `int` oder ein `float` in ihr gespeichert werden darf.
- Dabei merkt er, dass `int_var` dann ein schlechter Name ist und ändert ihn in `my_var`.

```
1 my_var: int | float = 1 + 7 # A variable hinted as either int or float.
2 print(type(my_var)) # This prints "<class 'int'>".
3
4 my_var = my_var / 3 # / returns a float, which is OK.
5 print(type(my_var)) # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_2.py` ↓

```
1 <class 'int'>
2 <class 'float'>
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Hier annotiert er die Variable mit `: int | float`, was bedeutet, dass entweder ein `int` oder ein `float` in ihr gespeichert werden darf.
- Dabei merkt er, dass `int_var` dann ein schlechter Name ist und ändert ihn in `my_var`.
- Damit ist das Programm nun tatsächlich fehlerlos, was auch Mypy anerkennt.

```
1 $ mypy variable_types_wrong_hints_2.py --no-strict-optional --check-
   ↪ untyped-defs
2 Success: no issues found in 1 source file
3 # mypy 1.19.1 succeeded with exit code 0.
```

# Probieren wir das mal aus



- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Durch das Nutzen eines Type Checkers zusammen mit Type Hints können wir also eine Klasse von Programmierfehlern von Anfang an verhindern.

# Probieren wir das mal aus



- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Durch das Nutzen eines Type Checkers zusammen mit Type Hints können wir also eine Klasse von Programmierfehlern von Anfang an verhindern.
- Entweder, der Programmierer hätte gesehen, dass er aus Versehen versucht, einen `float` in einer `int`-Variable zu speichern. Dann hätte er den anderen Divisionsoperator verwendet.

# Probieren wir das mal aus



- In `variable_types_wrong_hints_1.py` wollte er tatsächlich, dass `int_var` eine Ganzzahlvariable ist.
- In `variable_types_wrong_hints_2.py` hatte er vor, entweder Ganzzahlen oder Fließkommazahlen in der Variable zu speichern.
- Durch das Nutzen eines Type Checkers zusammen mit Type Hints können wir also eine Klasse von Programmierfehlern von Anfang an verhindern.
- Entweder, der Programmierer hätte gesehen, dass er aus Versehen versucht, einen `float` in einer `int`-Variable zu speichern. Dann hätte er den anderen Divisionsoperator verwendet.
- Oder er hätte gemerkt, dass der Name der Variable schlecht ist und ihn geändert.



## Probieren wir das mal aus

- Durch das Nutzen eines Type Checkers zusammen mit Type Hints können wir also eine Klasse von Programmierfehlern von Anfang an verhindern.
- Entweder, der Programmierer hätte gesehen, dass er aus Versehen versucht, einen `float` in einer `int`-Variable zu speichern. Dann hätte er den anderen Divisionsoperator verwendet.
- Oder er hätte gemerkt, dass der Name der Variable schlecht ist und ihn geändert.
- All das geht ohne Performance-Kosten, denn der Python-Interpreter ignoriert Type Hints.



## Probieren wir das mal aus

- Durch das Nutzen eines Type Checkers zusammen mit Type Hints können wir also eine Klasse von Programmierfehlern von Anfang an verhindern.
- Entweder, der Programmierer hätte gesehen, dass er aus Versehen versucht, einen `float` in einer `int`-Variable zu speichern. Dann hätte er den anderen Divisionsoperator verwendet.
- Oder er hätte gemerkt, dass der Name der Variable schlecht ist und ihn geändert.
- All das geht ohne Performance-Kosten, denn der Python-Interpreter ignoriert Type Hints.

### Gute Praxis

Benutzen Sie **immer** Type Hints.



# Type Checking und Type Hints

- Es gibt gute Gründe, immer Type Hints zu verwenden.

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

- Typ-bezogene Fehler sind eine Kategorie von Bugs, die erst während der Ausführung eines Programmes sichtbar werden.

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

- Typ-bezogene Fehler sind eine Kategorie von Bugs, die erst während der Ausführung eines Programmes sichtbar werden.
- Gleichzeitig sind das Fehler, die wir ziemlich einfach durch statische Kodeanalyse entdecken können.

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

- Typ-bezogene Fehler sind eine Kategorie von Bugs, die erst während der Ausführung eines Programmes sichtbar werden.
- Gleichzeitig sind das Fehler, die wir ziemlich einfach durch statische Kodeanalyse entdecken können.
- Zumindest viel einfacher als logische Fehler.

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

- Typ-bezogene Fehler sind eine Kategorie von Bugs, die erst während der Ausführung eines Programmes sichtbar werden.
- Gleichzeitig sind das Fehler, die wir ziemlich einfach durch statische Kodeanalyse entdecken können.
- Zumindest viel einfacher als logische Fehler.
- In C oder Java sind sie viel viel seltener.

# Type Checking und Type Hints



- Es gibt gute Gründe, immer Type Hints zu verwenden.

*Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and that most errors occur run-time.*

— Paul Jansen [37], 2025

- Typ-bezogene Fehler sind eine Kategorie von Bugs, die erst während der Ausführung eines Programmes sichtbar werden.
- Gleichzeitig sind das Fehler, die wir ziemlich einfach durch statische Kodeanalyse entdecken können.
- Zumindest viel einfacher als logische Fehler.
- In C oder Java sind sie viel viel seltener.
- Mit Type Hints und statischen Type-Checkern wie Mypy bringen wir diese Funktionalität nach Python.

# Weiteres Beispiel



- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.

```
1 int_var = 1 + 7 # Create an integer variable holding an integer number.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 float_var = 3.0 # 3.0 is a float and it is stored in float_var.  
5 print(type(float_var)) # This prints "<class 'float'>".  
6  
7 str_var = f"f{float_var = }" # Render an f-string into str_var.  
8 print(type(str_var)) # This prints "<class 'str'>".  
9  
10 bool_var = (1 == 0) # 1 == 0 is False, so a bool is stored in bool_var.  
11 print(type(bool_var)) # This prints "<class 'bool'>".  
12  
13 none_var = None # We create none_var which, well, holds None.  
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```



## Weiteres Beispiel

- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.
- Die Variable `int_var`, in der wir die Ganzzahl `8` speichern, wird mit `: int` annotiert.

```
1 int_var: int = 1 + 7 # Create an integer variable holding an integer.
2 print(type(int_var)) # This prints "<class 'int'>".
3
4 float_var: float = 3.0 # 3.0 is a float and it is stored in float_var.
5 print(type(float_var)) # This prints "<class 'float'>".
6
7 str_var: str = f"{float_var = }" # Render an f-string.
8 print(type(str_var)) # This prints "<class 'str'>".
9
10 bool_var: bool = (1 == 0) # 1 == 0 is False, so a bool is stored.
11 print(type(bool_var)) # This prints "<class 'bool'>".
12
13 none_var: None = None # We create none_var which, well, holds None.
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```



## Weiteres Beispiel

- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.
- Die Variable `int_var`, in der wir die Ganzzahl `8` speichern, wird mit `: int` annotiert.
- Die Variable `float_var`, in der wir die Fließkommazahl `3.0` speichern, wird mit `: float` annotiert.

```
1 int_var: int = 1 + 7 # Create an integer variable holding an integer.
2 print(type(int_var)) # This prints "<class 'int'>".
3
4 float_var: float = 3.0 # 3.0 is a float and it is stored in float_var.
5 print(type(float_var)) # This prints "<class 'float'>".
6
7 str_var: str = f"{float_var = }" # Render an f-string.
8 print(type(str_var)) # This prints "<class 'str'>".
9
10 bool_var: bool = (1 == 0) # 1 == 0 is False, so a bool is stored.
11 print(type(bool_var)) # This prints "<class 'bool'>".
12
13 none_var: None = None # We create none_var which, well, holds None.
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```



## Weiteres Beispiel

- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.
- Die Variable `int_var`, in der wir die Ganzzahl `8` speichern, wird mit `: int` annotiert.
- Die Variable `float_var`, in der wir die Fließkommazahl `3.0` speichern, wird mit `: float` annotiert.
- Die Variable `str_var`, in der wir den String `"float_var = 3.0"`, wird mit `: str` annotiert.



## Weiteres Beispiel

- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.
- Die Variable `int_var`, in der wir die Ganzzahl `8` speichern, wird mit `: int` annotiert.
- Die Variable `float_var`, in der wir die Fließkommazahl `3.0` speichern, wird mit `: float` annotiert.
- Die Variable `str_var`, in der wir den String `"float_var = 3.0"`, wird mit `: str` annotiert.
- Die Variable `bool_var`, in der wir den Wert `False` speichern, wird mit `: bool` annotiert.



## Weiteres Beispiel

- Annotieren wir nun unser gutes altes Programm `variable_types.py` mit Type Hints.
- Die Variable `int_var`, in der wir die Ganzzahl `8` speichern, wird mit `: int` annotiert.
- Die Variable `float_var`, in der wir die Fließkommazahl `3.0` speichern, wird mit `: float` annotiert.
- Die Variable `str_var`, in der wir den String `"float_var = 3.0"`, wird mit `: str` annotiert.
- Die Variable `bool_var`, in der wir den Wert `False` speichern, wird mit `: bool` annotiert.
- Und die Variable `none_var`, in der wir `None` speichern, wird mit `: None` annotiert.

# Weiteres Beispiel



- Wir bekommen das neue Programm `variable_types_hints.py`.

```
1 int_var: int = 1 + 7 # Create an integer variable holding an integer.  
2 print(type(int_var)) # This prints "<class 'int'>".  
3  
4 float_var: float = 3.0 # 3.0 is a float and it is stored in float_var.  
5 print(type(float_var)) # This prints "<class 'float'>".  
6  
7 str_var: str = f"{float_var = }" # Render an f-string.  
8 print(type(str_var)) # This prints "<class 'str'>".  
9  
10 bool_var: bool = (1 == 0) # 1 == 0 is False, so a bool is stored.  
11 print(type(bool_var)) # This prints "<class 'bool'>".  
12  
13 none_var: None = None # We create none_var which, well, holds None.  
14 print(type(none_var)) # This prints "<class 'NoneType'>".
```



## Weiteres Beispiel

- Wir bekommen das neue Programm `variable_types_hints.py`.
- Wir können es Mypy prüfen, und das flutscht problemlos durch.

```
1 $ mypy variable_types_hints.py --no-strict-optional --check-untyped-defs
2 Success: no issues found in 1 source file
3 # mypy 1.19.1 succeeded with exit code 0.
```



## Weiteres Beispiel

- Wir bekommen das neue Programm `variable_types_hints.py`.
- Wir können es Mypy prüfen, und das flutscht problemlos durch.
- Es wird auch klar dass es keinen Sinn hat, den Datentyp in den Variablennamen einzufügen.



## Weiteres Beispiel

- Wir bekommen das neue Programm `variable_types_hints.py`.
- Wir können es Mypy prüfen, und das flutscht problemlos durch.
- Es wird auch klar dass es keinen Sinn hat, den Datentyp in den Variablennamen einzufügen.
- Stattdessen sollten wir unsere Intention immer mit Type Hints ausdrücken.

# Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?





## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.



## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie psycopg<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.



## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie `psycopg`<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. `Matplotlib`<sup>34</sup>, `NumPy`<sup>59</sup>, und `Pandas`<sup>65</sup>.



## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie psycopg<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. Matplotlib<sup>34</sup>, NumPy<sup>59</sup>, und Pandas<sup>65</sup>.
  3. Andere Pakete wie Scikit-learn und SciPy unterstützen Type Hints nicht ... geben als Grund aber an, dass das mit ihrer existierener Kodebasis zu schwer umzusetzen wäre<sup>1,10</sup>.



## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr, wenn sie doch bis vor Kuzem nicht mal Teil der Programmiersprache waren?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie psycopg<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. Matplotlib<sup>34</sup>, NumPy<sup>59</sup>, und Pandas<sup>65</sup>.
  3. Andere Pakete wie Scikit-learn und SciPy unterstützen Type Hints nicht ... geben als Grund aber an, dass das mit ihrer existierener Kodebasis zu schwer umzusetzen wäre<sup>1,10</sup>.
- Das viele populäre Werkzeuge Type Hints nicht oder nur teilweise nutzen liegt daran, dass der Standard PEP 484<sup>97</sup> erst von 2014 ist.



## Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie psycopg<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. Matplotlib<sup>34</sup>, NumPy<sup>59</sup>, und Pandas<sup>65</sup>.
  3. Andere Pakete wie Scikit-learn und SciPy unterstützen Type Hints nicht ... geben als Grund aber an, dass das mit ihrer existierener Kodebasis zu schwer umzusetzen wäre<sup>1,10</sup>.
- Das viele populäre Werkzeuge Type Hints nicht oder nur teilweise nutzen liegt daran, dass der Standard PEP 484<sup>97</sup> erst von 2014 ist.
- Wir lernen also:



# Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie `psycopg`<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. `Matplotlib`<sup>34</sup>, `NumPy`<sup>59</sup>, und `Pandas`<sup>65</sup>.
  3. Andere Pakete wie `Scikit-learn` und `SciPy` unterstützen Type Hints nicht ... geben als Grund aber an, dass das mit ihrer existierener Kodebasis zu schwer umzusetzen wäre<sup>1,10</sup>.
- Das viele populäre Werkzeuge Type Hints nicht oder nur teilweise nutzen liegt daran, dass der Standard PEP 484<sup>97</sup> erst von 2014 ist.
- Wir lernen also:

## Gute Praxis

Es ist wichtig, Type Hints bereits von Anfang an in einem Projekt zu nutzen.



# Brauchen wir das wirklich?

- Warum hype ich Type Hints so sehr?
- Schauen wir uns dazu mal das Python-Ökosystem an, um herauszufinden, was erfahrene Programmierer wahrscheinlich denken.
  1. Es gibt mehrere wichtige Werkzeuge, wie `psycopg`<sup>100</sup>, den PostgreSQL-Python Adapter, die vollständig mit Type Hints nach PEP 484<sup>97</sup> annotiert sind.
  2. Viele andere Bibliotheken benutzen Type Hints zumindest teilweise und versuchen zumindest, das aller neuer Kode annotiert ist, z. B. `Matplotlib`<sup>34</sup>, `NumPy`<sup>59</sup>, und `Pandas`<sup>65</sup>.
  3. Andere Pakete wie `Scikit-learn` und `SciPy` unterstützen Type Hints nicht ... geben als Grund aber an, dass das mit ihrer existierener Kodebasis zu schwer umzusetzen wäre<sup>1,10</sup>.
- Das viele populäre Werkzeuge Type Hints nicht oder nur teilweise nutzen liegt daran, dass der Standard PEP 484<sup>97</sup> erst von 2014 ist.
- Wir lernen also:

## Gute Praxis

Es ist wichtig, Type Hints bereits von Anfang an in einem Projekt zu nutzen. Die Idee, dass man ja Kode später annotieren kann, ist falsch.



## Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.





# Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.





# Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.
- Solche Angriffe auf Datenbanken können verhindert werden, wenn SQL-Anfragen niemals mit Hilfe von String-Operationen oder f-Strings erstellt werden, sondern immer nur String-Literale sind.



# Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.
- Solche Angriffe auf Datenbanken können verhindert werden, wenn SQL-Anfragen niemals mit Hilfe von String-Operationen oder f-Strings erstellt werden, sondern immer nur String-Literale sind.
- Python bietet den Datentyp `LiteralString` für String-Literale an<sup>85</sup>.



# Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.
- Solche Angriffe auf Datenbanken können verhindert werden, wenn SQL-Anfragen niemals mit Hilfe von String-Operationen oder f-Strings erstellt werden, sondern immer nur String-Literale sind.
- Python bietet den Datentyp `LiteralString` für String-Literale an<sup>85</sup>.
- Implementierungen der Python DB Application Programming Interface (API) wie `psycopg`<sup>100</sup> können also so annotiert werden, dass sie SQL-Anfragen nur annehmen, wenn sie von diesem Datentyp sind.



# Type Hints und Sicherheit (Security)

- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.
- Solche Angriffe auf Datenbanken können verhindert werden, wenn SQL-Anfragen niemals mit Hilfe von String-Operationen oder f-Strings erstellt werden, sondern immer nur String-Literale sind.
- Python bietet den Datentyp `LiteralString` für String-Literale an<sup>85</sup>.
- Implementierungen der Python DB Application Programming Interface (API) wie `psycopg`<sup>100</sup> können also so annotiert werden, dass sie SQL-Anfragen nur annehmen, wenn sie von diesem Datentyp sind.
- Ein Type Checker könnte also feststellen, wenn fälschlicherweise eine SQL-Anfrage kein String-Literal ist.

# Type Hints und Sicherheit (Security)



- Statische Type-Checker können auch einen positiven Einfluss auf Sicherheitsaspekte haben, wie wir in unserem *Databases*-Buch erläutern<sup>103</sup>.
- So genannte Injection-Angriffe wie SQL Injection Attacken (SQLi attack) sind seit vielen Jahren ein Sicherheitsproblem.
- Solche Angriffe auf Datenbanken können verhindert werden, wenn SQL-Anfragen niemals mit Hilfe von String-Operationen oder f-Strings erstellt werden, sondern immer nur String-Literale sind.
- Python bietet den Datentyp `LiteralString` für String-Literale an<sup>85</sup>.
- Implementierungen der Python DB Application Programming Interface (API) wie `psycopg`<sup>100</sup> können also so annotiert werden, dass sie SQL-Anfragen nur annehmen, wenn sie von diesem Datentyp sind.
- Ein Type Checker könnte also feststellen, wenn fälschlicherweise eine SQL-Anfrage kein String-Literal ist.
- Noch unterstützt das Mypy aber noch nicht – dieses Feature ist zu neu<sup>100,111</sup>.



# Zusammenfassung



# Zusammenfassung

- Python ist eine stark- und dynamisch typisierte Programmiersprache.



# Zusammenfassung

- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.



# Zusammenfassung

- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.



# Zusammenfassung



- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.
- Werkzeuge wie Mypy können unseren Kode auf mögliche Fehler in der Typisierung von Variablen und Ausdrücken analysieren.

# Zusammenfassung



- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.
- Werkzeuge wie Mypy können unseren Kode auf mögliche Fehler in der Typisierung von Variablen und Ausdrücken analysieren.
- Sie können erkennen, wenn wir Werte zuweisen, deren Typen nicht passen.

# Zusammenfassung



- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.
- Werkzeuge wie Mypy können unseren Kode auf mögliche Fehler in der Typisierung von Variablen und Ausdrücken analysieren.
- Sie können erkennen, wenn wir Werte zuweisen, deren Typen nicht passen.
- Sie können das als mögliche Fehler erkennen ... Da sie aber nicht wissen, warum wir das machen, können sie nicht verstehen, welche Art Fehler das sind.

# Zusammenfassung



- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.
- Werkzeuge wie Mypy können unseren Kode auf mögliche Fehler in der Typisierung von Variablen und Ausdrücken analysieren.
- Sie können erkennen, wenn wir Werte zuweisen, deren Typen nicht passen.
- Sie können das als mögliche Fehler erkennen ... Da sie aber nicht wissen, warum wir das machen, können sie nicht verstehen, welche Art Fehler das sind.
- Type Hints helfen uns dagegen, explizit auszudrücken, was für Typen Variablen und Parameter haben sollen.

# Zusammenfassung



- Python ist eine stark- und dynamisch typisierte Programmiersprache.
- Das hat viele Vorteile, macht es aber auch einfach, bestimmte Fehler zu machen.
- Nun haben Sie zwei weitere Bausteine zum Erstellen von sauberem und sicheren Kode kennengelernt.
- Werkzeuge wie Mypy können unseren Kode auf mögliche Fehler in der Typisierung von Variablen und Ausdrücken analysieren.
- Sie können erkennen, wenn wir Werte zuweisen, deren Typen nicht passen.
- Sie können das als mögliche Fehler erkennen ... Da sie aber nicht wissen, warum wir das machen, können sie nicht verstehen, welche Art Fehler das sind.
- Type Hints helfen uns dagegen, explizit auszudrücken, was für Typen Variablen und Parameter haben sollen.
- Sie bringen statische Typisierung nach Python – und funktionieren mit Type Checkern wie Mypy zusammen.



谢谢您们！  
Thank you!  
Vielen Dank!



# References I



- [1] Dowon „[akahard2dj](#)“, Ilhan „[ilayn](#)“ Polat, Pauli „[pv](#)“ Virtanen, Daniel „[dpinol](#)“ Pinyol, Lucas „[lucascolley](#)“ Colley, Matt „[mdhaber](#)“ Haberland und Lars „[lagru](#)“ Grüter. *Issue #9038: Applying Type Hint(PEP 484) for SciPy*. San Francisco, CA, USA: GitHub Inc, 16. Juli 2018–16. Juni 2024. URL: <https://github.com/scipy/scipy/issues/9038> (besucht am 2025-02-02) (siehe S. 201–209).
- [2] 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. 241, 244).
- [3] David M. Beazley. "Data Processing with Pandas". *login: Usenix Magazin* 37(6), Dez. 2012. Berkeley, CA, USA: USENIX Association. ISSN: [1044-6397](#). URL: <https://www.usenix.org/publications/login/december-2012-volume-37-number-6/data-processing-pandas> (besucht am 2024-06-25) (siehe S. 242).
- [4] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: [978-0-596-00743-0](#) (siehe S. 244).
- [5] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, Mai 2008. ISBN: [978-0-321-35668-0](#) (siehe S. 241).
- [6] Paul Boddie, Skip Montanaro, Michael Foord und Alex Martelli. "Why is Python a dynamic language and also a strongly typed language". In: *The Python Wiki*. Beaverton, OR, USA: Python Software Foundation (PSF), 12. Aug. 2008–24. Feb. 2012. URL: <https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language> (besucht am 2025-08-08) (siehe S. 14–20).
- [7] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: [978-0-13-769132-6](#) (siehe S. 242).
- [8] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: [978-1-78862-936-2](#) (siehe S. 240).
- [9] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: <https://fstring.help> (besucht am 2024-07-25) (siehe S. 241).

# References II



- [10] Santiago „[bryant1410](#)“ Castro, Thomas J. „[thomasjpfan](#)“ Fan, Joel „[jnothman](#)“ Nothman, Roman „[rth](#)“ Yurchak, Guillaume „[glemaire](#)“ Lemaitre und Nicolas „[NicolasHug](#)“ Hug. *Issue #16705: Support Typing*. San Francisco, CA, USA: GitHub Inc, 17. März–31. Aug. 2020. URL: <https://github.com/scikit-learn/scikit-learn/issues/16705> (besucht am 2025-02-02) (siehe S. 201–209).
- [11] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dez. 2024. ISBN: 978-1-990783-55-5 (siehe S. 241).
- [12] Donald D. Chamberlin. “50 Years of Queries”. *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:[10.1145/3649887](https://doi.org/10.1145/3649887). URL: <https://cacm.acm.org/research/50-years-of-queries> (besucht am 2025-01-09) (siehe S. 243).
- [13] 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. 244).
- [14] Edgar Frank „[Ted](#)“ Codd. “A Relational Model of Data for Large Shared Data Banks”. *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:[10.1145/362384.362685](https://doi.org/10.1145/362384.362685). URL: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf> (besucht am 2025-01-05) (siehe S. 243).
- [15] Ignacio Samuel Crespo-Martínez, Adrián Campazas Vega, Ángel Manuel Guerrero-Higueras, Virginia Riego-Del Castillo, Claudia Álvarez-Aparicio und Camino Fernández Llamas. “SQL Injection Attack Detection in Network Flow Data”. *Computers & Security* 127:103093, Apr. 2023. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0167-4048. doi:[10.1016/J.COSE.2023.103093](https://doi.org/10.1016/J.COSE.2023.103093) (siehe S. 243).
- [16] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 243).
- [17] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: <https://dataschool.com/how-to-teach-people-sql> (besucht am 2025-02-27) (siehe S. 243).
- [18] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 243).

# References III



- [19] Justin Dennison, Cherokee Boose und Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, Juni 2024. ISBN: [978-1-83620-863-1](#) (siehe S. 242).
- [20] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: [979-8-8688-0224-9](#) (siehe S. 240).
- [21] *ECMAScript Language Specification*. Standard ECMA-262, 3rd Edition. Geneva, Switzerland: Ecma International, Dez. 1999. URL: [https://ecma-international.org/wp-content/uploads/ECMA-262\\_3rd\\_edition\\_december\\_1999.pdf](https://ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf) (besucht am 2024-12-15) (siehe S. 241).
- [22] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: [978-1-83763-564-1](#) (siehe S. 242).
- [23] "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. 241).
- [24] 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. 241).
- [25] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: <https://www.ibm.com/topics/api> (besucht am 2024-12-12) (siehe S. 240).
- [26] 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. 241).
- [27] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: [978-0-443-23791-1](#) (siehe S. 243).

# References IV



- [28] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 243).
- [29] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli „pv“ Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke und Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (siehe S. 242).
- [30] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 241).
- [31] 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. 244).
- [32] 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. 243).
- [33] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, Mai–Juni 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (siehe S. 242).
- [34] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom und The Matplotlib Development Team. "Coding Guidelines". In: *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: [https://matplotlib.org/devdocs/dev/coding\\_guide.html](https://matplotlib.org/devdocs/dev/coding_guide.html) (besucht am 2025-02-02) (siehe S. 201–209).
- [35] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom und The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: <https://matplotlib.org> (besucht am 2025-02-02) (siehe S. 242).

# References V



- [36] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1.* International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: [https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO\\_IEC\\_9075-1\\_2023\\_ed\\_6\\_-\\_id\\_76583\\_Publication\\_PDF\\_\(en\).zip](https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip) (besucht am 2025-01-08). Consists of several parts, see <https://modern-sql.com/standard> for information where to obtain them. (Siehe S. 243).
- [37] Paul Jansen. *TIOBE Index for January 2025 – January Headline: Python is TIOBE's programming language of the year 2024!* Eindhoven, The Netherlands: TIOBE Software BV, Jan. 2025. URL: <https://www.tiobe.com/tiobe-index> (besucht am 2025-01-07) (siehe S. 184–190).
- [38] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4246-9 (siehe S. 242, 243).
- [39] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: <https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf> (besucht am 2024-08-23) (siehe S. 241).
- [40] "exit – Terminate a Process". 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/exit.html> (besucht am 2024-10-30) (siehe S. 240).
- [41] Faizan Khan, Boqi Chen, Dániel Varró und Shane McIntosh. "An Empirical Study of Type-Related Defects in Python Projects". *IEEE Transactions on Software Engineering* 48(8):3145–3158, 1. Aug. 2022. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 0098-5589. doi:10.1109/TSE.2021.3082068. URL: <https://www.researchgate.net/publication/351729684> (besucht am 2024-08-16) (siehe S. 79–81).
- [42] Animesh Kumar, Sandip Dutta und Prashant Pranav. "Analysis of SQL Injection Attacks in the Cloud and in WEB Applications". *Security and Privacy* 7(3), Mai–Juni 2024. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 2475-6725. doi:10.1002/SPY2.370 (siehe S. 243).

# References VI



- [43] 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. 244).
- [44] 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. 243).
- [45] Michael Lee, Ivan Levkivskyi und Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), 14. März 2019. URL: <https://peps.python.org/pep-0586> (besucht am 2024-12-17) (siehe S. 242).
- [46] 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. 90–96, 107–114, 242).
- [47] Marc-André Lemburg. *Python Database API Specification v2.0*. Python Enhancement Proposal (PEP) 249. Beaverton, OR, USA: Python Software Foundation (PSF), 12. Apr. 1999. URL: <https://peps.python.org/pep-0249> (besucht am 2025-02-02) (siehe S. 243).
- [48] Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. New York, NY, USA: Apress Media, LLC, Dez. 2018. ISBN: 978-1-4842-4281-0 (siehe S. 240).
- [49] Reuven M. Lerner. *Pandas Workout*. Shelter Island, NY, USA: Manning Publications, Juni 2024. ISBN: 978-1-61729-972-8 (siehe S. 242).
- [50] Marc Loy, Patrick Niemeyer und Daniel Leuck. *Learning Java*. 5. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2020. ISBN: 978-1-4920-5627-0 (siehe S. 241).
- [51] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 243).
- [52] Charlie Marsh. "Ruff". In: URL: <https://pypi.org/project/ruff> (besucht am 2025-08-29) (siehe S. 243).
- [53] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: <https://docs.astral.sh/ruff> (besucht am 2024-08-23) (siehe S. 243).

# References VII



- [54] 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. 241).
- [55] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components.* The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISBN: 978-1-55860-456-8 (siehe S. 243).
- [56] Carl Meyer. *Python Virtual Environments.* Python Enhancement Proposal (PEP) 405. Beaverton, OR, USA: Python Software Foundation (PSF), 13. Juni 2011–24. Mai 2012. URL: <https://peps.python.org/pep-0405> (besucht am 2024-12-25) (siehe S. 245).
- [57] 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. 240).
- [58] NumPy Team. *NumPy.* San Francisco, CA, USA: GitHub Inc und Austin, TX, USA: NumFOCUS, Inc. URL: <https://numpy.org> (besucht am 2025-02-02) (siehe S. 242).
- [59] NumPy Team. "Typing (`numpy.typing`)". In: *NumPy.* San Francisco, CA, USA: GitHub Inc und Austin, TX, USA: NumFOCUS, Inc. URL: <https://numpy.org/devdocs/reference/typing.html> (besucht am 2025-02-02) (siehe S. 201–209).
- [60] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running.* 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 242).
- [61] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991).* 26.–30. Okt. 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Kap. Paper P2.3, S. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (siehe S. 244).
- [62] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dez. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: <https://www.researchgate.net/publication/255673967> (besucht am 2025-09-05) (siehe S. 244).
- [63] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3.* New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (siehe S. 242).

# References VIII



- [64] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dez. 2021. ISBN: 978-1-4842-7854-3 (siehe S. 244).
- [65] Pandas Developers. *Contributing to the Code Base*. Austin, TX, USA: NumFOCUS, Inc. und Montreal, QC, Canada: OVHcloud. URL: [https://pandas.pydata.org/docs/development/contributing\\_codebase.html](https://pandas.pydata.org/docs/development/contributing_codebase.html) (besucht am 2025-02-02) (siehe S. 201–209).
- [66] Pandas Developers. *Pandas*. Austin, TX, USA: NumFOCUS, Inc. und Montreal, QC, Canada: OVHcloud. URL: <https://pandas.pydata.org> (besucht am 2025-02-02) (siehe S. 242).
- [67] Alan Paul, Vishal Sharma und Oluwafemi Olukoya. "SQL Injection Attack: Detection, Prioritization & Prevention". *Journal of Information Security and Applications* 85:103871, Sep. 2024. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 2214-2126. doi:10.1016/J.JISA.2024.103871 (siehe S. 243).
- [68] Fabian Pedregos, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot und Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research (JMLR)* 12:2825–2830, Okt. 2011. Cambridge, MA, USA: MIT Press. ISSN: 1532-4435. doi:10.5555/1953048.2078195 (siehe S. 243).
- [69] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 242).
- [70] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: <https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf> (besucht am 2024-06-29) (siehe S. 240).
- [71] Sebastian Raschka, Yuxi Liu und Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-learn*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2022. ISBN: 978-1-80181-931-2 (siehe S. 243).
- [72] Ori Roth. "Python Type Hints Are Turing Complete (Pearl/Brave New Idea)". In: *37th European Conference on Object-Oriented Programming (ECOOP'2023)*. 17.–21. Juli 2023, Seattle, WA, USA. Hrsg. von Karim Ali und Guido Salvaneschi. Bd. 263 der Reihe Leibniz International Proceedings in Informatics (LIPIcs). Wadern, Saarland, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 44:1–44:15. ISSN: 1868-8969. ISBN: 978-3-95977-281-5. doi:10.4230/LIPICS.ECOOP.2023.44 (siehe S. 155–169).

# References IX



- [73] Ernest E. Rothman, Rich Rosen und Brian Jepson. *Mac OS X for Unix Geeks*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2008. ISBN: [978-0-596-52062-5](#) (siehe S. 242).
- [74] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, Juli–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: [0740-7459](#). doi:[10.1109/MS.2006.91](#) (siehe S. 244).
- [75] Yeonhee Ryou, Sangwoo Joh, Joomo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: [978-1-4503-9475-8](#). doi:[10.1145/3551349.3559497](#) (siehe S. 241).
- [76] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: [978-1-80602-393-6](#) (siehe S. 245).
- [77] Stephan Scheuermann. "SQL Injection". In: *1st Kassel Student Workshop on Security in Distributed Systems (KaSWoSDS'2008)*. 13. Feb. 2008, Kassel, Hessen, Germany. Hrsg. von Thomas Weise (汤卫恩) und Philipp Andreas Baer. Bd. 2008-1 der Reihe Kasseler Informatikschriften (KIS). Kassel, Hessen, Germany: Universität Kassel, Universitätsbibliothek, 14. Apr. 2008. Kap. 3, S. 35–49. URL: <https://kobra.uni-kassel.de/items/a6134d61-d5c3-4cb8-804f-bbb89a3f59a7> (besucht am 2025-08-23) (siehe S. 243).
- [78] 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. 241).
- [79] Bryan Sills, Brian Gardner, Kristin Marsicano und Chris Stewart. *Android Programming: The Big Nerd Ranch Guide*. 5. Aufl. Reading, MA, USA: Addison-Wesley Professional, Mai 2022. ISBN: [978-0-13-764579-4](#) (siehe S. 240).
- [80] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: [978-1-0981-3391-7](#) (siehe S. 241).
- [81] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials (2025)*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: [978-1-80580-309-6](#) (siehe S. 241, 242).

# References X



- [82] 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. 241).
- [83] John Miles Smith und Philip Yen-Tang Chang. “Optimizing the Performance of a Relational Algebra Database Interface”. *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:[10.1145/361020.361025](https://doi.org/10.1145/361020.361025) (siehe S. 243).
- [84] “SQL Commands”. In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: <https://www.postgresql.org/docs/17/sql-commands.html> (besucht am 2025-02-25) (siehe S. 243).
- [85] Pradeep Kumar Srinivasan und Graham Bleaney. *Arbitrary Literal String Type*. Python Enhancement Proposal (PEP) 675. Beaverton, OR, USA: Python Software Foundation (PSF), 30. Nov. 2021–7. Feb. 2022. URL: <https://peps.python.org/pep-0675> (besucht am 2025-03-04) (siehe S. 210–216, 243).
- [86] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Tech Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 236, 243).
- [87] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burghann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of<sup>86</sup> (siehe S. 243).
- [88] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 243).
- [89] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 242).
- [90] *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: <https://docs.python.org/3/tutorial> (besucht am 2025-04-26).

# References XI



- [91] . "Literals". In: *Static Typing with Python*. Hrsg. von The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: <https://typing.python.org/en/latest/spec/literal.html> (besucht am 2025-08-29) (siehe S. 242).
- [92] The Python Typing Team, Hrsg. *Static Typing with Python*. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: <https://typing.python.org> (besucht am 2025-08-28) (siehe S. 155–169).
- [93] George K. Thiruvathukal, Konstantin Läufer und Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dez. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: <https://www.researchgate.net/publication/220094077> (besucht am 2024-10-01) (siehe S. 244).
- [94] 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. 241).
- [95] 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. 241, 245).
- [96] Bruce M. Van Horn II und Quan Nguyen. *Hands-On Application Development with PyCharm*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-235-0 (siehe S. 243).
- [97] 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. 155–169, 201–209, 244).
- [98] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 241).
- [99] Daniele „dvarrazzo“ Varrazzo, Federico „fogzot“ Di Gregorio und Jason „jerickso“ Erickson. *Psycopg*. London, England, UK: The Psycopg Team, 2010–2023. URL: <https://www.psycopg.org> (besucht am 2025-02-02) (siehe S. 243).
- [100] Daniele „dvarrazzo“ Varrazzo, Federico „fogzot“ Di Gregorio und Jason „jerickso“ Erickson. "Static Typing". In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, 21. Juni 2022. URL: <https://www.psycopg.org/psycopg3/docs/advanced/typing.html> (besucht am 2025-03-04) (siehe S. 201–216, 243).

# References XII



- [101] Pauli „pv“ Virtanen, Ralf Gommers, Travis E. Oliphant, Matt „mdhaber“ Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan „ilayn“ Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregos, Paul van Mulbregt und SciPy 1.0 Contributors. „SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python“. *Nature Methods* 17:261–272, 2. März 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: <http://arxiv.org/abs/1907.10121> (besucht am 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Siehe S. 243).
- [102] „Virtual Environments and Packages“. In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 12. URL: <https://docs.python.org/3/tutorial/venv.html> (besucht am 2024-12-24) (siehe S. 245).
- [103] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2025. URL: <https://thomasweise.github.io/databases> (besucht am 2025-01-05) (siehe S. 210–216, 240, 243).
- [104] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: <https://thomasweise.github.io/programmingWithPython> (besucht am 2025-01-05) (siehe S. 242, 243).
- [105] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: <https://www.ibm.com/think/topics/relational-databases> (besucht am 2025-01-05) (siehe S. 243).
- [106] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 243).
- [107] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2022. ISBN: 978-1-83763-244-2 (siehe S. 243).
- [108] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: <https://www.techtarget.com/searchdatamanagement/definition/database-management-system> (besucht am 2025-01-11) (siehe S. 240).

# References XIII



- [109] Moshe Zadka und Guido van Rossum. *Changing the Division Operator*. Python Enhancement Proposal (PEP) 238. Beaverton, OR, USA: Python Software Foundation (PSF), 11. März–27. Juli 2001. URL: <https://peps.python.org/pep-0238> (besucht am 2025-07-28) (siehe S. 39–45).
- [110] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 240).
- [111] Jelle „[JelleZijlstra](#)“ Zijlstra, Mehdi „[hmc-cs-mdrissi](#)“ Drissi, Alex „[AlexWaygood](#)“ Waygood, Daniele „[dvarrazzo](#)“ Varrazzo, Shantanu „[hauntsaninja](#)“, François-Michel „[FinchPowers](#)“ L’Heureux und Rupesh „[rupeshs](#)“ Seeraman. *Issue #12554: Support PEP 675 (LiteralString)*. San Francisco, CA, USA: GitHub Inc, 9. Apr. 2022–29. Nov. 2024. URL: <https://github.com/python/mypy/issues/12554> (besucht am 2025-03-05) (siehe S. 210–216, 243).

# Glossary (in English) I



**Android** is a common operating system for mobile phones<sup>79</sup>.

**API** An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another<sup>25</sup>.

**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>8,57,110</sup>. Learn more at <https://www.gnu.org/software/bash>.

**C** is a programming language, which is very successful in system programming situations<sup>20,70</sup>.

**CI** *Continuous Integration* is a software development process where developers integrate new code into a codebase hosted in a Version Control Systems (VCS), after which automated tools run an automated build process including code analysis (such as linters) and unit test execution<sup>48</sup>. If the build succeeds and no errors or problems with the code are, the code may automatically be deployed (if the CI system is configured to do so).

**DB** A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*<sup>103</sup>.

**DBMS** A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB<sup>108</sup>.

**exit code** When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure<sup>40</sup>. Per convention, an exit code of 0 means success. Any non-zero exit code indicates an error. Under Python, you can terminate the current process at any time by calling `exit` and optionally passing in the exit code that should be returned. If `exit` is not explicitly called, then the interpreter will return an exit code of 0 once the process normally terminates. If the process was terminated by an uncaught `Exception`, a non-zero exit code, usually 1, is returned.

# Glossary (in English) II



- f-string** let you include the results of expressions in strings<sup>9,23,24,26,54,82</sup>. They can contain expressions (in curly braces) like `f"{}a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.
- Git** is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes<sup>80,95</sup>. Learn more at <https://git-scm.com>.
- IDE** An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For Python, we recommend using PyCharm. On Apple systems, Xcode is often used.
- iOS** is the operating system that powers Apple iPhones<sup>11,81</sup>. Learn more at <https://www.apple.com/ios>.
- Java** is another very successful programming language, with roots in the C family of languages<sup>5,50</sup>.
- JavaScript** JavaScript is the predominant programming language used in websites to develop interactive contents for display in browsers<sup>21</sup>.
- linter** A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles<sup>39,75</sup>. Ruff is an example for a linter used in the Python world.
- Linux** is the leading open source operating system, i.e., a free alternative for Microsoft Windows<sup>2,30,78,94,98</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.

# Glossary (in English) III



**literal** A literal is a specific concrete value, something that is written down as-is<sup>45,91</sup>. In Python, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

**macOS** or Mac OS is the operating system that powers Apple Mac(intosh) computers<sup>73,81</sup>. Learn more at <https://www.apple.com/macos>.

**Matplotlib** is a Python package for plotting diagrams and charts<sup>33,35,38,63</sup>. Learn more at at <https://matplotlib.org><sup>35</sup>.

**Microsoft Windows** is a commercial proprietary operating system<sup>7</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>46</sup> that makes use of type hints. Learn more at <https://github.com/python/mypy> and in<sup>104</sup>.

**NumPy** is a fundamental package for scientific computing with Python, which offers efficient array datastructures<sup>19,29,38</sup>. Learn more at <https://numpy.org><sup>58</sup>.

**OS** Operating System, the system that runs your computer, see, e.g., Linux, Microsoft Windows, macOS, and Android.

**Pandas** is a Python data analysis and manipulation library<sup>3,49</sup>. Learn more at <https://pandas.pydata.org><sup>66</sup>.

**PostgreSQL** An open source object-relational database management system (DBMS)<sup>22,60,69,89</sup>. See <https://postgresql.org> for more information.

# Glossary (in English) IV



- psycopg** or, more exactly, [psycopg](#) 3, is the most popular PostgreSQL adapter for Python, implementing the Python DB API 2.0 specification<sup>47</sup>. Learn more at <https://www.psycopg.org><sup>99</sup>.
- PyCharm** is the convenient Python Integrated Development Environment (IDE) that we recommend for this course<sup>96,106,107</sup>. It comes in a free community edition, so it can be downloaded and used at no cost. Learn more at <https://www.jetbrains.com/pycharm>.
- Python** The Python programming language<sup>32,44,51,104</sup>, i.e., what you will learn about in our book<sup>104</sup>. Learn more at <https://python.org>.
- relational database** A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other<sup>14,27,28,83,88,103,105</sup>.
- Ruff** is a linter and code formatting tool for Python<sup>52,53</sup>. Learn more at <https://docs.astral.sh/ruff> or in<sup>104</sup>.
- Scikit-learn** is a Python library offering various machine learning tools<sup>68,71</sup>. Learn more at <https://scikit-learn.org>.
- SciPy** is a Python library for scientific computing<sup>38,101</sup>. Learn more at <https://scipy.org>.
- SQL** The *Structured Query Language* is basically a programming language for querying and manipulating relational databases<sup>12,16–18,36,55,84,86–88</sup>. It is understood by many DBMSes. You find the SQL commands supported by PostgreSQL in the reference<sup>84</sup>.
- SQLi attack** A SQL injection attack is an attack that is used to target data stored in DBMS by injecting malicious input into code that constructs SQL queries by string concatenation in order to subvert application functionality and perform unauthorized operations<sup>15,42,67,77,103</sup>. In order to prevent such attacks, queries to DBs should *never* be constructed via string concatenation or the likes of Python f-strings. Assume that `user_id` was a string variable in a Python program and we construct the query `f"SELECT * FROM data WHERE user_id = {user_id}"`. Notice that the `{user_id}` will be replaced with the value of variable `user_id` during (string) interpolation. If `user_id == "user123; DROP TABLE data;"`, mayhem would ensue when we execute the query<sup>85</sup>. Some programming languages, like Python, offer built-in datatypes (such as `LiteralString`<sup>85</sup>) to annotate string constants that can be used by static type-checkers. At the time of this writing, Mypy does not support this yet<sup>100,111</sup>.

# Glossary (in English) V



**(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>2,13</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 + , dann Schreiben von `cmd`, dann Druck auf . Under Ubuntu Linux, + + 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>43,97</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.

**Ubuntu** is a variant of the open source operating system Linux<sup>13,31</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>.

**unit test** Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification<sup>4,61,62,64,74,93</sup>. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program.



# Glossary (in English) VI

**VCS** A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code<sup>95</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.

**virtual environment** A virtual environment is a directory that contains a local Python installation<sup>56,102</sup>. It comes with its own package installation directory. Multiple different virtual environments can be installed on a system. This allows different applications to use different versions of the same packages without conflict, because we can simply install these applications into different virtual environments.

**Xcode** is offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as macOS and iOS<sup>76</sup>.