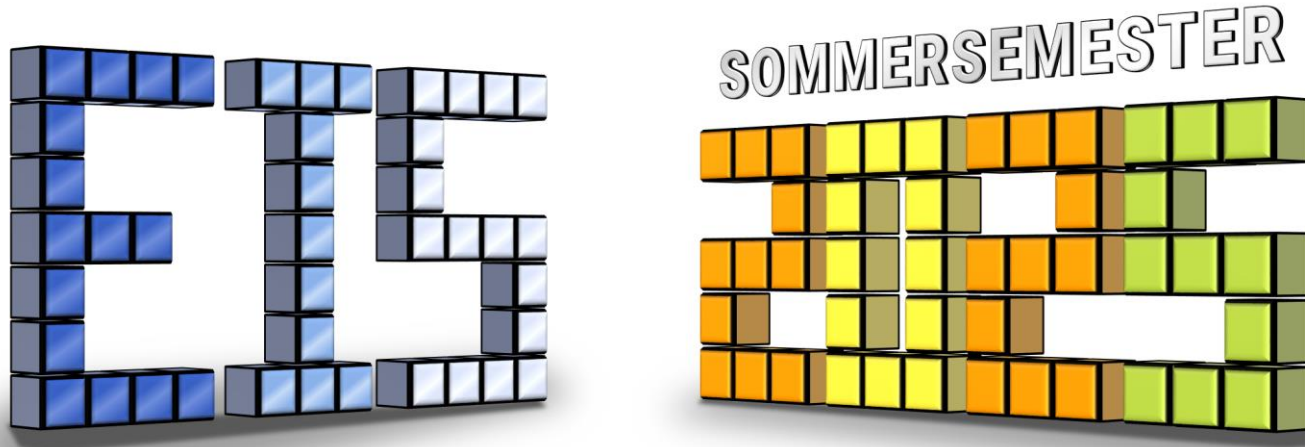


EINFÜHRUNG IN DIE SOFTWAREENTWICKLUNG

Sommersemester 2025



Foliensatz #2a

Programmiersprachen: Python + MyPy

Michael Wand
Institut für Informatik
Michael.Wand@uni-mainz.de



Disclaimer

Wichtig!

- Unsere Vorlesung ist konzeptionell orientiert!
- Dies ist kein klassischer „Programmierkurs“
 - Wir nutzen die Programmiersprachen nicht „voll aus“
 - Wir nutzen nicht immer die elegantesten / empfohlenen Konstrukte
 - Selbststudium + Übung für „schönen“ Code
- Ziel: Grundkonzepte verstehen
 - Sprachunabhängige Ideen
 - Guter Stil im Groben, aber nicht in syntaktischen Details
 - Daran scheitert kein Projekt :-)

Übersicht

Inhalt heute

- Organisatorisches
- Programmiersprachen
 - Python + MyPy
 - C/C++
 - Java/Scala

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Programmiersprachen

(a) Python

Python

Allgemeine Eigenschaften

- Sehr „konsequent“ objekt-orientierte Sprache
 - Konzeptionell ähnlich zu „SmallTalk“
- Multi-paradigmen
 - „Kann fast alles“
- Einsteiger- und Bedienerfreundlich
 - Macht Spaß zu benutzen, schöne Bibliotheken
- Für große Projekte: Tools empfehlenswert (z.B. MyPy)
 - Trotzdem kein „Spielzeug“
- Eher ineffizient
 - Hauptnachteil: sehr langsam, hoher Ressourcenbedarf
 - Daher (derzeit) Kombination mit C/C++ üblich

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Variablen

Variablen in Python

```
# Zuweisung ohne Deklaration
```

```
a = 3
```

```
# Typen nicht beschränkt
```

```
a = 3.0
```

```
# Lesen aus Variablen
```

```
print(a * 2) # Ergibt sechs
```

```
# Kein Lesezugriff auf unbenutzte Variablen
```

```
print(b) # Fehler42
```

```
b = 2
```


Python

Daten + Variablen

- **Objektorientiert**

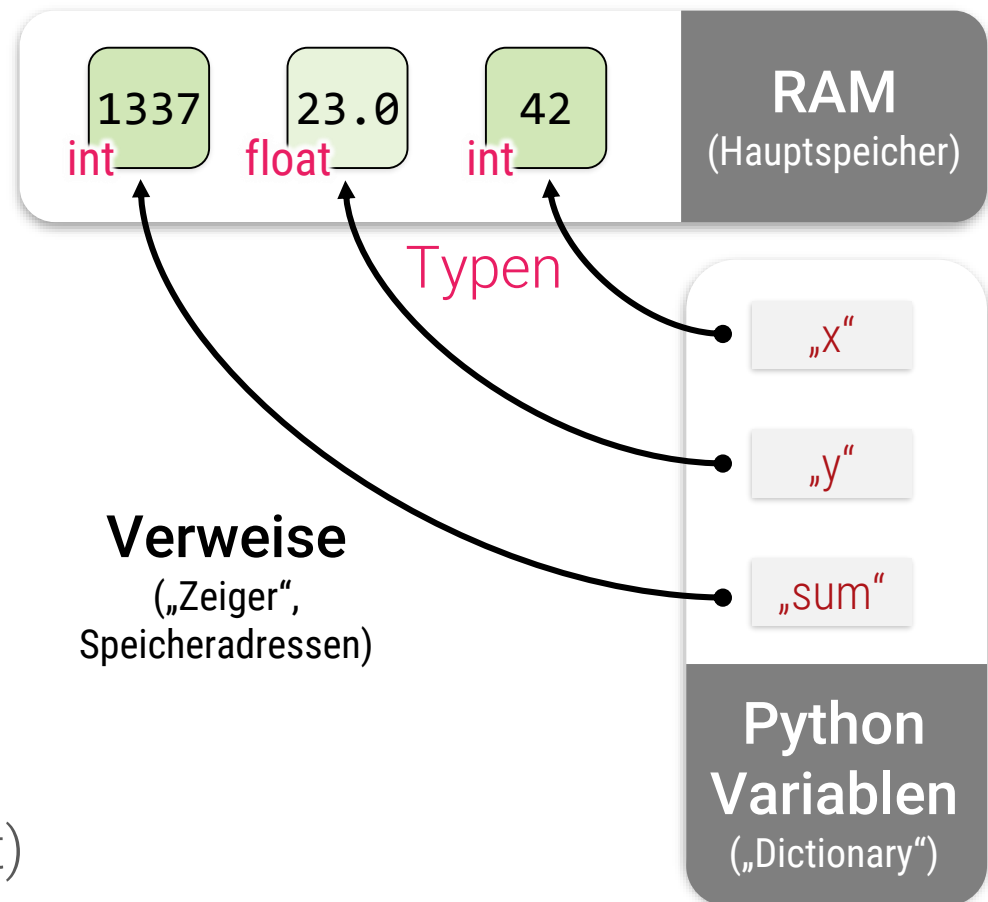
Variablen enthalten
Verweise auf Objekte

- Immer Referenzen!
- **None** = Leere Referenz

- **Dynamisch typisiert**

- Objekte haben Typen
- Variablen haben
keinen Typ (alles passt)

- Objekte für „Werte“ (**int**, **str**)
sind unveränderlich (immutable)



Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Python

Ausdrücke & Berechnungen

- Normale mathematische Notation

```
# Ein einfacher arithmetischer Ausdruck  
print(3 + 3)
```

6

```
# Mit Klammern + Punkt-vor-Strich  
print(2 * (5 + 5) + 11 * 2)
```

42

- Objektorientiert: Übersetzung in Methoden
 $a + b \rightarrow a._\text{add}_(b)$

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Befehle: Elementare Anweisungen

Anweisungen / Befehle

```
# Zuweisung (Wert: 6)
a = 3 + 3

# Unterprogrammaufruf (Ausgabe: 3)
print(a // 2)

# Methodenaufruf (Syntax)
text_obj.print_me(console) 42
```

- Objektorientiert: Methodenaufruf
 - Typ des Objektes (Klasse) hat Liste mit Methoden
 - Auswahl nach Typ
 - Auch bei Operatoren, wie „+“ \equiv `__add__`

Befehle: Kontrollstrukturen

Sequenz

```
<Anweisung 1>  
<Anweisung 2>  
<Anweisung 3>  
...  
<Anweisung n>
```

Wiederholung

```
while <Bedingung>:  
    <Anweisungen>
```

Fallunterscheidung

```
if <Bedingung>:  
    <Anweisungen>  
else:  
    <Anweisungen>
```

Unterprogramme

```
def <Name>(<Parameter>):  
    <Anweisungen>  
    return <Ergebnis>
```

Diskussion: Programmiersprache

Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Wichtige Abstraktionen

Abstraktionen

- Unterprogramme
- Klassen
 - Sammlung von Methoden für Objekte bestimmten Typs
- Komplexe Datentypen als Objekte
 - Records/Structs: Mehrere Felder in Objekten
 - Funktional: Funktionen sind (auch nur) Objekte
 - Klassen, Module (alles!) sind auch nur Objekte
- Polymorphie
 - Auswahl von Methoden nach Typ erlaubt gleichen Code für verschiedene Daten

Diskussion: Programmiersprache

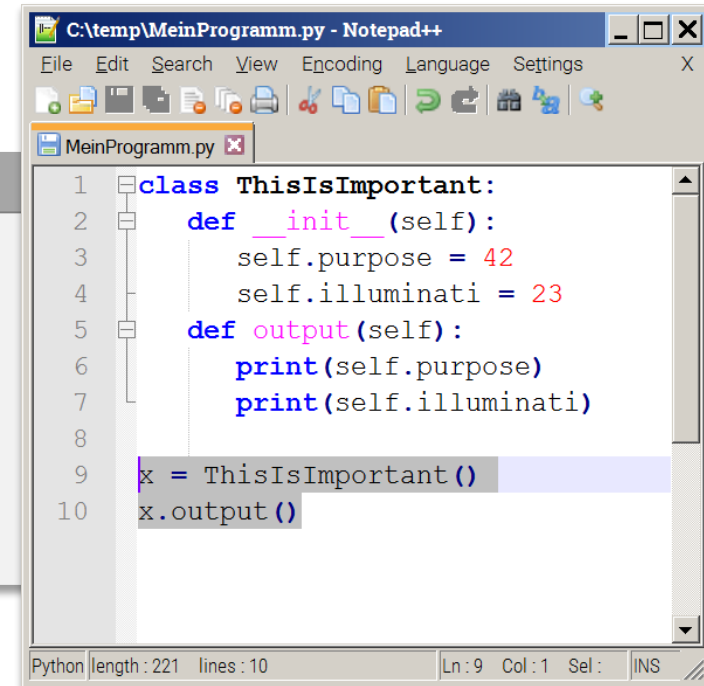
Vorlage zur Diskussion

- Daten + Variablen
- Ausdrücke und Berechnungen
- Befehle
- Abstraktionen
- Systemumgebung

Python Tools

Python Programme erstellen & ausführen

- > **Notepad++** `MeinProgramm.py`
(Sourcecode in Textdatei)
- > **python3** `MeinProgramm.py`
(Code in Bytecode übersetzt)
(wird danach direkt ausgeführt)



The screenshot shows a Notepad++ window with the title bar 'C:\temp\MeinProgramm.py - Notepad++'. The menu bar includes File, Edit, Search, View, Encoding, Language, and Settings. The toolbar contains icons for file operations and editing. The main text area displays the following Python code:

```
1 class ThisIsImportant:
2     def __init__(self):
3         self.purpose = 42
4         self.illuminati = 23
5     def output(self):
6         print(self.purpose)
7         print(self.illuminati)
8
9 x = ThisIsImportant()
10 x.output()
```

The status bar at the bottom indicates 'Python | length : 221 | lines : 10' and 'Ln : 9 | Col : 1 | Sel : | INS'.

Python Umgebung

- Verschiedene Interpreter oder (JIT-) Compiler
 - Meistens „CPython“, auf Konsole als „**python3**“
 - Optimierte Varianten wie PyPy, Cython, etc. verfügbar
- Programme als Textfiles, Module als Verzeichnisse
 - IDE zu empfehlen, z.B. VSCode oder PyCharm

Module

Python Daten können importiert werden:

- **from** `my_mod` **import** `do_something`
 - Referenzen werden in globales „Dictionary“ eingefügt
 - Dynamisches Laden zur Laufzeit
 - Keine Vorverarbeitung nötig
- Hierarchische Pakete via **import** `package.subdir.my_mod`
 - Abgebildet auf Verzeichnisstruktur
 - Verzeichnisse „zählen“ nur wenn „`__init__.py`“ vorhanden

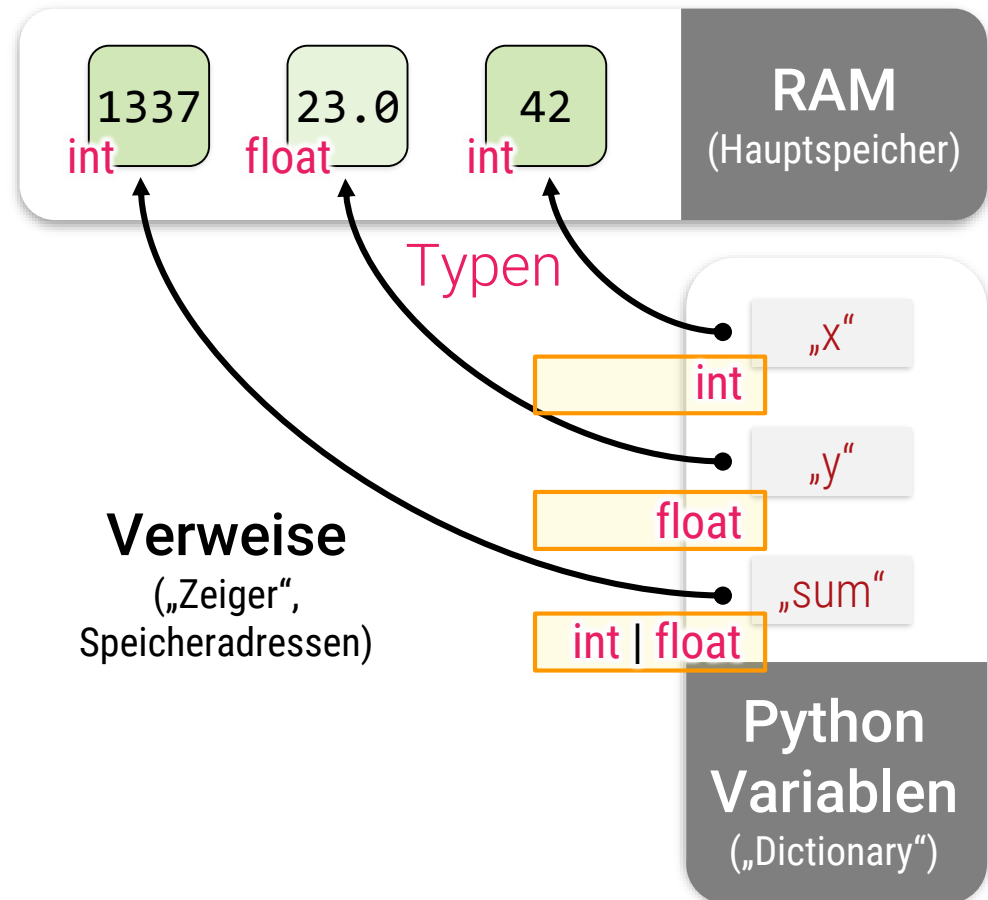
Programmiersprachen (1b): Statische Typisierung *mit* Python + MyPy

Statische Typisierung

Python

Statische Typisierung

- Variablen haben *auch* Typen
 - Nur passende Objekte erlaubt
- **Statische Analyse:** Fehler vor Ausführung bei Typinkompatibilität
 - Oft beim Übersetzen
 - In Python via separatem Type-Checker „MyPy“
 - In viele IDEs integriert



Statische Typisierung in Python mit MyPy

Statische Typisierung

Statische Typisierung

- In Python via Add-On, Typchecker „MyPy“

Notation: Optionale Typannotationen

Variablen

```
a: int = 3 + 3
```

```
b: float = 3 / 2
```

Parameter für Unterprogramme

```
def square(a: float) -> float:  
    return a*a
```

Unterprogramm ohne Rückgabe

```
def say_hello() -> None:  
    print("Hello World")
```


Statische Typisierung

Mit MyPy kann man spezifizieren

- Genaue Typen (z.B. `int`)

Aber auch z.B. (Details später!)

- Alternativen (Summentypen) (z.B. `int | float`)
- Oberklassen (z.B. `GameCharacter` erlaubt `PacMan`)
 - (bei Vererbung `class PacMan(GameCharacter)`)
- Interfaces (z.B. `Enumerable`, `Iterable`)
- Generische Typen (z.B. `list[int]`)
- Nullable Types (`Optional[<type>]`), erlaubt `None`
- uneingeschränkte dyn. Typisierung via Typ „`any`“

Nullable Types

Ein Blick auf Optional

- Variablen dürfen per Default nicht **None** sein

```
x: int = None # MyPy meldet einen Fehler!
```

- Variablen, die leer sein dürfen („Nullable“) müssen gekennzeichnet werden mit **Optional**[<type>]

```
x: Optional[int] = None # erlaubt  
x = 42 # jetzt ist was drin
```

Frage: Any vs. object?

- „**object**“: beliebiges Objekt, das nicht **None** ist
- „**Any**“ erlaubt jede Referenz, auch **None**

Nullable Types

Ein Blick auf Optional (wg. Scala, später)

- Vermeidet Laufzeitfehler: Prüfung erforderlich!

```
def square(x: Optional[int]) -> int:  
    if x is not None: # ohne if zeigt MyPy einen Fehler an!  
        return x*x  
    else:  
        return -42
```

Typumwandlung

Typumwandlung (z.B. Spezialisierung)

- Allgemein: MyPy erlaubt Typumwandlung via Prüfung

```
def my_len(x: object) -> int:
    if isinstance(x, str): # MyPy erkennt Prüfung
        return len(x)
    elif isinstance(x, int):
        return x
    else:
        raise TypeError("I like only int or str.")
```

- Assert geht auch

```
def my_len(x: object) -> int:
    assert isinstance(x, str): # MyPy erkennt Prüfung
    return len(x)
```

Systemumgebung

Wie führe ich Python Programme aus?

- Schreiben
 - Texteditor, Textdatei (UTF-8) mit Endung *.py
- Ausführen

```
> python3 my_code.py
```

- (Übersetzung in Byte-Code on-the-fly)
- Typprüfung (optional, vor Ausführung)

```
> mypy my_code.py
```

- IDE stark empfohlen (VSCODE, PyCharm, o.ä.)

Regel für EIS

Typen immer angeben

- Mit Doppelpunkt hinter Variable

- statt `x = 5` → `x: int = 5`

- statt `a = "Test"` → `a: str = "Test"`

- statt `def f(a):` → `def f(a: int) -> float:`

- Bei Funktionen (Unterprogrammen)...

- ...und bei normalen Variablen
 - Machen wir ab jetzt immer!

- Voll-dynamische Typisierung weiterhin möglich

- `x: Any = ...` (anything goes! – nur nutzen, wo sinnvoll)
 - Empfehlung: Any „explizit kennzeichnen“

Zusätzliche dynamische
Überprüfung?

Beispiel für „fieber_detector“

(Beispiel aus der EIP Vorlesung)

```
def fieber_detector(body_temp: float, fahrenheit: bool)
    if fahrenheit:
        body_temp: float = (body_temp - 32.0) * 5.0 / 9.0
    if body_temp >= 37.0:
        print("Fieber")
    else:
        print("kein Fieber")
```

`fieber_detector(38.5, False)` # ok

`fieber_detector(97, True)` # ok, Konvertierung int -> float

`fieber_detector("Hello World", True)` # Fehler (MyPy-Check)

typeguard – runtime type checks

(Beispiel aus EIP Vorlesung)

```
from typeguard import typechecked # 3rd-party library
```

`@typechecked` neu!

```
def fieber_detector(body_temp: float, fahrenheit: bool)
```

```
    if fahrenheit:
```

```
        body_temp = (body_temp - 32.0) * 5.0 / 9.0
```

```
    if body_temp >= 37.0:
```

```
        print("Fieber")
```

```
    else:
```

```
        print("kein Fieber")
```

- Python selbst ignoriert Annotationen!
- Keine Laufzeitfehler wg. Annotationen!
- Extratools nötig (z.B. `typeguard`)

Fehler würde weiterhin von MyPy erkannt

Zusätzlich: Laufzeitfehler, bei Ausführung, an dieser Stelle

```
fieber_detector("Hello World", 42 + 23)
```

Typsysteme von Programmiersprachen

Python

Typsysteme

- Statisch/dynamisch typisiert
- Stark/schwach typisiert

Python ist

- Dynamisch typisiert
 - Mit **MyPy**: optional statisch typisiert
- (Relativ) stark typisiert

Theorie: Typsysteme

Dynamisch typisiert

- Nur Objekte haben Typen
 - Variablen können jeden Typ aufnehmen
 - Typ steht erst zur Laufzeit fest
 - Vorteil: Flexibler („Polymorphe Algorithmen“)

Statisch typisierte Sprachen

- Variablen haben einen festen Typ
 - Steht vor Ausführung („compile-time“) fest
- Vorteile
 - Optimierungen möglich (potentiell schneller)
 - Prüfung durch Compiler vermeidet Fehler

Starke und schwache Typisierung

Stark typisierte Sprachen

- Typen von Daten werden überprüft
 - Sinnlose Operationen werden nicht durchgeführt
- Beispiele: Python, JAVA, m.E. C/C++/Pascal

Schwach typisierte Sprachen: „Anything goes“

- Variante 1: Naheliegendste Konvertierung
 - Basic, PHP, JavaScript
 - Objekte kennen Ihren Laufzeittyp
- Variante 2: Ungeprüfter Speicherzugriff (evtl. Crash)
 - Maschinensprache, „void*“ in C/C++
 - Objekte kennen ihren eigenen Laufzeittyp gar nicht

Python

Python ist (ohne MyPy)

- Dynamisch typisiert
- und stark typisiert

Wird gerne durcheinandergeworfen...

Theoretische Grenzen

Theoretische Informatik

Zentrale Erkenntnis aus Th.-I.

- Es gibt unentscheidbare Probleme
 - Man kann kein Programm schreiben, das Klassen von solchen Problemen löst
 - (notw.: unendlich viele Varianten als konkretes Problem)

Was ist unentscheidbar?

- Frage, ob ein Programm ein bestimmtes Verhalten zeigt
 - Z.B., wird eine spezielle Zeile Code ausgeführt?
 - Nimmt eine Variable einen bestimmten Wert an?
- Für spezielle Fälle ist es lösbar, aber nicht allgemein
 - Sobald Programmablauf eine Rolle spielt

Statische Typisierung

Typen von Daten in Variablen einschränken

- Programm macht sonst vielleicht keinen Sinn
- Beispiel `(a - b) ** c`
 - Für `a = 2, b = 3, c = 4`
 - Alles ok!
 - Für `a = "Hello", b = "World", c = "!"`
 - Macht keinen Sinn!
- Einschränkungen:
 - `a, b, c` müssen Zahlen sein
 - Python: `complex`, `float` oder `int`
 - MyPy: Schwächerer Typ erlaubt: `complex` \subset `float` \subset `int`

Typannotationen in Python

Typen von Daten in Variablen einschränken

- Programm macht sonst vielleicht keinen Sinn
- Beispiel `(a - b) ** c`
 - Für `a = 2, b = 3, c = 4`
 - Alles ok!
 - Für `a = "Hello", b = "World", c = "!"`
 - Macht keinen Sinn!
- Also:
`a: float = ...`
`b: float = ...`
`c: float = ...`
`(a - b) ** c`

Typannotations in Python

Man kann Typen von Daten einschränken

- Beispiel `(a - b) ** n` (`a, b, c: float`)
 - Für `a = 2, b = 3, c = 4` – `ok`: int erlaubt
 - Für `a = "Hello", b = "World", c = "!"` – `Fehler`: str kein float
- Wo das Typsystem trotzdem versagt:
 - Für `a = 2.0, b = 3.0, c = 0.5` → $(2 - 3)^{1/2} = \sqrt{-1}$
 - $\sqrt{-1}$ ist nicht reell (kein `float`)
 - Was passiert in echt? (getestet mit Python 3.12/mypy 1.9)
 - `mypy` erlaubt `float ** float` → `float`
 - Python erzeugt komplexe Zahlen bei Wurzeln aus negativen
 - Bei komplexem Ergebnis: Typloch („type hole“) in `mypy`

Beispiel

imaginary problems...

@typechecked # Prüft Eingaben und Rückgaben der Funktion

```
def calculation(a: float, b: float, n: float) -> float:  
    difference: float = a - b  
    result: float = difference ** n  
    return result
```

```
calculation(3, 2, 0.5) # ok!
```

```
calculation("Hello", "World", "!") # Fehler durch MyPy erkannt,  
                                     # bevor (!) Program abläuft
```

```
calculation(2, 3, 0.5) # Fehler erst zur Laufzeit (@typechecked)  
# Fehler: <<TypeError: type of the return value must be either float or int; got complex instead>>
```

Typchecks

In den meisten Programmiersprachen

- Statische Typprüfung nur auf „Gleichheit“
 - „Rechnen nur mit floats“
 - Automatische Konvertierung möglich, danach checken
 - z.B. `a: float = 4`
 - Typchecker prüft nur, ob Typen „passen“
 - Gewisse weitergehende Features möglich (Generics, Subtypes, etc – mehr dazu später)
- Was (fast^{*)}) niemand kann:
 - Überprüfung auf Eigenschaften, die sich erst während der Rechnung/Laufzeit ergeben können
 - z.B. „positive Zahlen“ für floats, die „Subtrahieren“ erlauben

^{*)} „fast“: es gibt „dependently typed“ languages, aber das ist kompliziert/speziell