



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Anhang 0 Teil 2

EINFÜHRUNG IN DIE
PROGRAMMIERUNG MIT
PYTHON

Worum es heute geht

Teil 1 hat Ihnen beigebracht, wie Sie über ein Terminal **mit Ihrem Computer sprechen**.

Jetzt beantworten wir eine größere Frage: **Warum sollten Sie das überhaupt wollen?**

Wir bauen gemeinsam eine Sache -- eine Wetterstation -- von Grund auf. Unterwegs werden Sie an Grenzen stoßen. Jede Grenze zwingt Sie, nach einem neuen Werkzeug zu greifen. Am Ende haben Sie die vollständige mentale Landkarte, wie Programmierer denken.

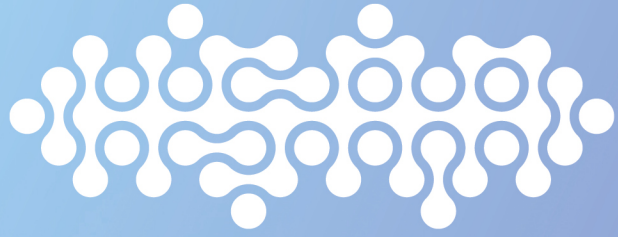
Sie müssen keine Syntax auswendig lernen. Sie müssen Intuition aufbauen.

Kurze Umfrage

Wie würden Sie Ihre Python-Erfahrung einschätzen?

- **A)** Noch nie eine Zeile Python geschrieben
- **B)** Ein, zwei Tutorials gemacht, aber nicht sicher
- **C)** Kann einfache Skripte schreiben
- **D)** Komfortabel mit Funktionen und Klassen

Keine falsche Antwort! Diese Sitzung baut von Null auf.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Warum schreiben Menschen Programme?

1

Jede App beginnt mit einem Problem

Denken Sie an die Apps auf Ihrem Smartphone.

- **Instagram** -- jemand wollte Fotos einfach mit Filtern teilen
- **Spotify** -- jemand wollte Musik ohne Piraterie zugänglich machen
- **Google** -- jemand wollte Dinge in einem wachsenden Ozean von Webseiten finden

Das sind alles **Probleme**, die mit **Programmen** gelöst wurden.

Und es sind nicht nur Consumer-Apps -- Wissenschaft, Ingenieurwesen, Medizin, Finanzen, Kunst. **Programmierung ist überall.**

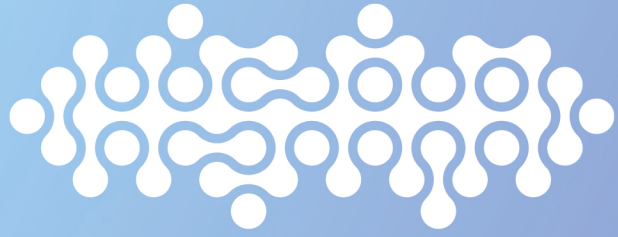
Von großen Ideen zu kleinen Schritten

Sie können nicht am ersten Tag Netflix bauen. Die Ingenieure bei Netflix konnten das auch nicht.

Der Trick: Finden Sie Probleme, die **klein genug sind, um sie heute zu lösen**, aber **bedeutsam genug, um sich dafür zu interessieren**.

Unser heutiges Problem: **eine Wetterstation**. Jeder versteht Wetter. Es beinhaltet Zahlen, Text und Entscheidungen. Einfach genug zum Anfangen, reichhaltig genug, um zu zeigen, warum jede neue Idee wichtig ist.

Fangen wir an.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Warum Python?

2

Hunderte Sprachen -- warum diese?

- 1. Kein Kompilierungsschritt.** Viele Sprachen erfordern erst Kompilieren, dann Ausführen. Python überspringt den Zwischenschritt. Sie schreiben Code und führen ihn sofort aus. Experimentieren geht schnell und schmerzlos.
- 2. Für Menschen gebaut.** Guido van Rossum (1991): Code soll leicht zu lesen, leicht zu schreiben und schnell von der Idee zum funktionierenden Programm führen.
- 3. Die Sprache der Wissenschaft und KI.** NumPy, Pandas, TensorFlow, PyTorch -- alle Python-first. Wenn Sie mit Daten oder KI arbeiten, werden Sie Python begegnen.

Schreiben. Ausführen. Fertig.

C / Java (kompiliert):

1. Code schreiben
2. Kompilieren
3. Binary ausführen

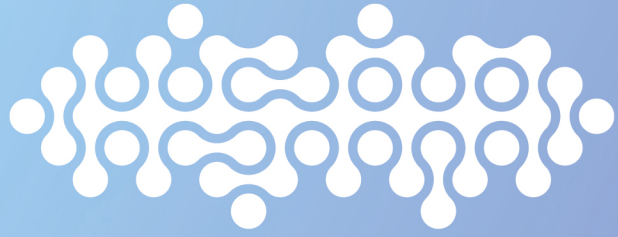
```
$ gcc main.c -o main  
$ ./main
```

Python (interpretiert):

1. Code schreiben
2. Ausführen!

```
$ python hello.py  
Hello from Python!
```

Kein Compiler. Kein Build-Schritt. **Von der Idee zum Ergebnis in Sekunden.** Deshalb nutzen wir es.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Es ist ein Computer -- lass
ihn rechnen

3

Die Bausteine: Variablen

Jedes Problem, das Sie mit Code lösen, beginnt mit Daten. Eine **Variable** ist ein Name, der an einen Wert gebunden ist -- eine beschriftete Box. Sie wählen den Namen, Python merkt sich den Wert.

```
temperature_celsius = 23.5      # float (Dezimalzahl)
number_of_sensors = 4          # int (ganze Zahl)
city = "Aalen"                  # str (Text, ein "String")
is_raining = False              # bool (True oder False)
```

Vier Typen reichen erstmal: `int`, `float`, `str`, `bool`.

Wenn Sie jemals Mathe gemacht haben, verstehen Sie diese bereits.

Es ist ein Computer -- natürlich kann er rechnen

Die Maschine vor Ihnen wurde buchstäblich zum Berechnen gebaut. Also lassen Sie sie:

```
celsius = 23.5  
fahrenheit = celsius * 9 / 5 + 32  
print(fahrenheit)  # 74.3
```

Sie kennen bereits Arithmetik. Python auch: `+` `-` `*` `/` `//` `%` `**`

Und es kann vergleichen: `>` `<` `==` `>=` `<=` `!=` -- das Ergebnis ist immer `True` oder `False`.

```
temperature > 30      # False  
temperature == 23.5   # True
```

Super. Wir können einen Wert speichern und damit rechnen. Aber ein einzelner Wert ist nicht sehr nützlich...

Ein Wert ist einsam -- Listen kommen ins Spiel

Eine Wetterstation zeichnet nicht eine Temperatur auf. Sie zeichnet jede Stunde eine auf. Sie brauchen eine Möglichkeit, **mehrere Werte zusammen** zu speichern.

```
daily_temps = [18.2, 21.0, 19.5, 23.5, 25.1, 22.8, 20.3]

print(daily_temps[0])      # 18.2  (Zählung beginnt bei 0!)
print(len(daily_temps))    # 7
```

Und Sie können sie durchlaufen:

```
total = 0
for temp in daily_temps:
    total = total + temp
average = total / len(daily_temps)    # 21.49
```

Listen sind leistungsstarke Behälter. Aber wenn Ihr Code wächst, reichen Behälter allein nicht aus. Sehen wir warum.

Die Mauer: Copy-Paste bringt uns um

Sie überwachen fünf Städte. Jede Temperatur muss umgerechnet werden:

```
celsius_aalen = 23.5
fahrenheit_aalen = celsius_aalen * 9 / 5 + 32

celsius_berlin = 19.0
fahrenheit_berlin = celsius_berlin * 9 / 5 + 32

# ... dieselbe Formel noch 3 Mal
```

Dieselbe Formel erscheint **fünfmal**. Stellen Sie sich vor:

- Was, wenn Sie **100 Städte** hätten?
- Was, wenn sich die Formel **ändern** müsste?
- Was, wenn Sie in einer Zeile einen **Tippfehler** machen?

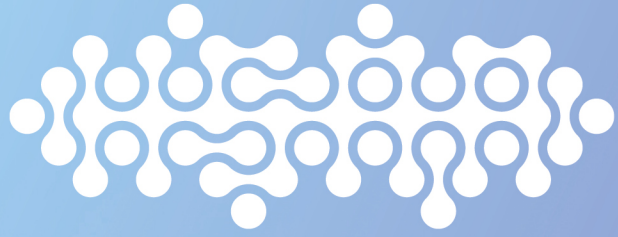
Copy-Paste ist ein Rezept für Bugs. Es muss einen besseren Weg geben.

Hands-On: Spüren Sie das Problem

Öffnen Sie das [Guided Hands-On Notebook](#) und gehen Sie zu **Abschnitt 1-2**:

1. Erstellen Sie eine Variable `celsius` mit der heutigen ungefähren Temperatur
2. Rechnen Sie in Fahrenheit um: `fahrenheit = celsius * 9 / 5 + 32`
3. Erstellen Sie eine Liste mit fünf verschiedenen Temperaturen
4. Rechnen Sie **jede einzelne** in Fahrenheit um -- bemerken Sie, wie nervig die Wiederholung ist

Sie haben 3 Minuten. Spüren Sie den Schmerz. Das nächste Kapitel ist die Lösung.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Funktionen: Wiederverwendung statt Wiederholung

4

Sie kennen Funktionen bereits

Im Matheunterricht hat Ihr Lehrer geschrieben: $f(x) = x \cdot \frac{9}{5} + 32$

Sie haben die Formel nicht jedes Mal neu geschrieben. Sie haben einfach $f(23,5)$ aufgerufen und 74,3 bekommen.

Python funktioniert genauso:

```
def celsius_to_fahrenheit(celsius):  
    return celsius * 9 / 5 + 32
```

`def` = "definiere eine Funktion." Ein Input, ein Output. **Die Formel existiert einmal.**

```
print(celsius_to_fahrenheit(23.5))    # 74.3  
print(celsius_to_fahrenheit(19.0))    # 66.2
```

Das Wiederholungsproblem -- gelöst

Erinnern Sie sich an die fünf kopierten Umrechnungen? Kombinieren Sie eine Funktion mit einer Liste:

```
cities = [  
    ["Aalen", 23.5], ["Berlin", 19.0], ["Munich", 21.3],  
    ["Stuttgart", 20.7], ["Hamburg", 16.8],  
]  
  
for city_data in cities:  
    print(city_data[0] + ":", celsius_to_fahrenheit(city_data[1]))
```

Fünf Städte oder fünftausend -- **der Code bleibt gleich lang.**

Ändern Sie die Formel an einer Stelle, und sie aktualisiert sich überall. Das ist die Kraft.

Funktionen können mehr

Mehrere Eingaben, mehrere Ausgaben -- kein Problem:

```
def analyze_temperatures(readings):  
    min_val = min(readings)  
    max_val = max(readings)  
    avg_val = sum(readings) / len(readings)  
    return min_val, max_val, avg_val  
  
lowest, highest, average = analyze_temperatures(daily_temps)
```

Funktionen sind ein **riesiger Fortschritt**. Sie beseitigen Wiederholung und machen Code lesbar.

Aber wenn Ihre Wetterstation wächst, stoßen Sie an eine neue Mauer...

Die Mauer: Zu viele lose Werte

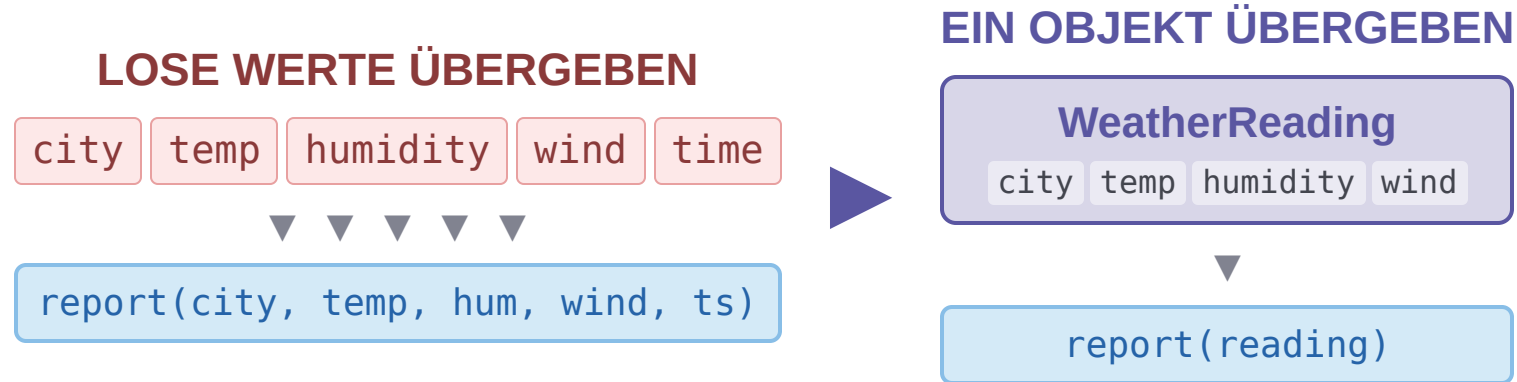
Ihre Station erfasst Temperatur, Luftfeuchtigkeit, Windgeschwindigkeit, Stadt, Zeitstempel. Jetzt braucht jede Funktion **alle davon**:

```
report_weather("Aalen", 23.5, 65.0, 12.3, "2026-03-11 14:00")  
wind_chill = feels_like(23.5, 65.0, 12.3)
```

Die Werte `23.5`, `65.0`, `12.3` **gehören zusammen** -- sie sind alle eine Messung. Aber wir übergeben sie als separate, lose Werte.

Was, wenn Sie versehentlich Luftfeuchtigkeit und Windgeschwindigkeit vertauschen? **Python beschwert sich nicht. Ihre Ergebnisse sind einfach falsch.**

Was wir stattdessen wollen



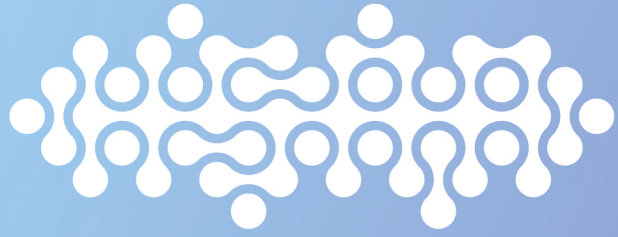
Wenn Werte **zusammengehören**, verlangen sie danach, **zusammengefasst** zu werden.

Hands-On: Spüren Sie den Schmerz loser Werte

Öffnen Sie das [Guided Hands-On Notebook](#) und gehen Sie zu **Abschnitt 3**:

1. Schreiben Sie `celsius_to_fahrenheit(celsius)` -- rechnet C in F um
2. Schreiben Sie `feels_like(temp, humidity, wind)` -- gibt `temp - (wind * 0.7)` zurück
3. Rufen Sie **beide** für drei verschiedene Städte auf
4. Zählen Sie, wie oft Sie dieselben Zahlen tippen

Sie haben 3 Minuten. Beachten Sie die Wiederholung -- dieselben Werte, immer wieder.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Klassen: Zusammenfassen, was zusammengehört

5

Stellen Sie sich ein leeres Formular vor

Eine **Klasse** ist ein leeres Formular -- eine Vorlage mit beschrifteten Feldern. Jedes Mal, wenn Sie eines ausfüllen, erstellen Sie eine **Instanz**: eine konkrete Messung mit konkreten Werten.

```
class WeatherReading:
    def __init__(self, city, temperature, humidity, wind_speed):
        self.city = city
        self.temperature = temperature
        self.humidity = humidity
        self.wind_speed = wind_speed
```

- `class` -- definiert das Formular
- `__init__` -- wird beim Ausfüllen ausgeführt
- `self` -- "dieses bestimmte Exemplar"

Daten, die zusammengehören, leben jetzt zusammen

```
aalen = WeatherReading("Aalen", 23.5, 65.0, 12.3)
berlin = WeatherReading("Berlin", 19.0, 72.0, 8.5)

print(aalen.city)           # Aalen
print(aalen.temperature)    # 23.5
print(berlin.humidity)      # 72.0
```

Und die chaotische Fünf-Parameter-Funktion? **Jetzt nur noch ein Parameter:**

```
def report_weather(reading):
    print(reading.city, reading.temperature, "C")

report_weather(aalen)  # sauber, kein Vertauschen möglich
```

Das ist eine echte Verbesserung. Aber wir sind noch nicht fertig...

Die Mauer: Funktionen leben immer noch draußen

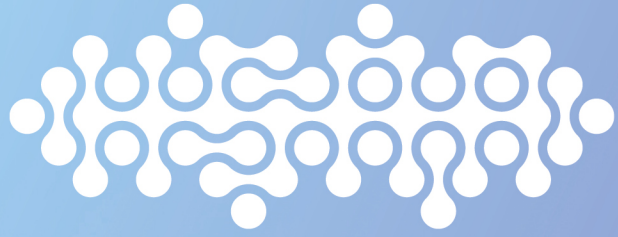
Wir haben eine `WeatherReading`-Klasse. Und wir haben eigenständige Funktionen:

```
celsius_to_fahrenheit(aalen.temperature)
feels_like(aalen.temperature, aalen.humidity, aalen.wind_speed)
```

Aber `celsius_to_fahrenheit` **ergibt nur Sinn** für eine Wettermessung. Und `feels_like` zieht drei Felder aus dem Objekt, das wir gerade sorgfältig zusammengefasst haben.

Warum die Funktionen **getrennt** von den Daten halten, auf denen sie arbeiten?

Wäre es nicht sinnvoller, sie **in die Klasse** zu packen?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Methoden: Verhalten trifft Daten

6

Funktionen nach innen verschieben

Eine **Methode** ist eine Funktion, die in einer Klasse lebt. Sie hat bereits Zugriff auf die Daten über `self` -- nichts muss übergeben werden.

```
class WeatherReading:
    def __init__(self, city, temperature, humidity, wind_speed):
        self.city = city
        self.temperature = temperature
        self.humidity = humidity
        self.wind_speed = wind_speed

    def to_fahrenheit(self):
        return self.temperature * 9 / 5 + 32

    def feels_like(self):
        return self.temperature - (self.wind_speed * 0.7)
```

Schauen Sie, wie sauber das jetzt ist

Vorher (eigenständig):

```
# Sie müssen die Reihenfolge kennen  
feels_like(23.5, 65.0, 12.3)
```

Fehleranfällig, umständlich.

Nachher (Methode):

```
aalen = WeatherReading(  
    "Aalen", 23.5, 65.0, 12.3)  
aalen.feels_like()
```

Das Objekt **kennt seine eigenen Daten**.

Eine ganze Liste verarbeiten -- elegant:

```
for station in stations:  
    print(station.city, "- Fahrenheit:", station.to_fahrenheit())
```

Objekte druckbar machen: `__str__`

Was passiert, wenn Sie `print(aalen)` versuchen?

```
<__main__.WeatherReading object at 0x7f8b2c3d4e50>
```

Nicht hilfreich. Python weiß nicht, wie es Ihr Objekt anzeigen soll. Sie können es ihm beibringen mit `__str__`:

```
def __str__(self):  
    return (self.city + ": " + str(self.temperature) + " C, "  
            + str(self.humidity) + "% humidity")  
  
print(aalen)    # Aalen: 23.5 C, 65.0% humidity
```

`__init__`, `__str__` -- das sind **Dunder-Methoden** (Double Underscore). Sie passen an, wie Python mit Ihren Objekten umgeht.

Wo wir jetzt stehen

WeatherReading

- + *str* city
- + *float* temperature
- + *float* humidity
- + *float* wind_speed

- + to_fahrenheit()
- + feels_like()
- + __str__()

Daten und Verhalten, **zusammen an einem Ort**. Wunderbar. Aber es gibt noch eine Mauer...

Die Mauer: Eine neue Art von Station

Ihre Stadt installiert **erweiterte** Wetterstationen. Sie messen auch UV-Index und Luftdruck. Sie brauchen eine neue Klasse -- aber sie soll weiterhin **alles können**, was `WeatherReading` bereits kann.

Kopieren Sie die gesamte Klasse und fügen zwei Felder hinzu?

Das würde funktionieren. Aber Sie hätten **zwei Kopien** desselben Codes. Beheben Sie einen Bug in einer, vergessen Sie die andere. Diesen Film haben wir schon gesehen.

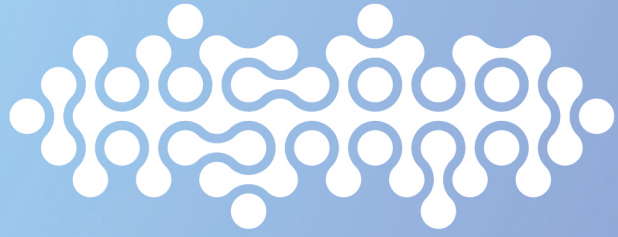
Es muss einen Weg geben zu sagen: "Mach alles, was die alte Klasse kann, plus mehr."

Hands-On: Bauen Sie ein WeatherReading

Öffnen Sie das [Guided Hands-On Notebook](#) und gehen Sie zu **Abschnitt 4-5**:

1. Definieren Sie `WeatherReading` mit `city`, `temperature`, `humidity`, `wind_speed`
2. Fügen Sie `to_fahrenheit(self)` und `feels_like(self)` als Methoden hinzu
3. Fügen Sie `is_comfortable(self)` hinzu: `True` wenn $18 \leq \text{temp} \leq 26$ und Luftfeuchtigkeit < 70
4. Erstellen Sie drei Instanzen, speichern Sie sie in einer Liste, iterieren und drucken

Sie haben 5 Minuten. Das ist das Kernmuster des gesamten Kurses.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Vererbung: Erweitern ohne Neuschreiben

7

Auf dem aufbauen, was bereits funktioniert

```
class AdvancedReading(WeatherReading):
    def __init__(self, city, temperature, humidity,
                  wind_speed, uv_index, pressure):
        super().__init__(city, temperature, humidity, wind_speed)
        self.uv_index = uv_index
        self.pressure = pressure
```

- `(WeatherReading)` -- "erbt alles vom Elternteil"
- `super().__init__(...)` -- lässt das Elternteil seine eigenen Felder einrichten
- Wir fügen nur hinzu, was **neu** ist: `uv_index` und `pressure`

Kein Copy-Paste. Kein duplizierter Code.

Es funktioniert einfach

```
reading = AdvancedReading("Aalen", 23.5, 65.0, 12.3, 6.2, 1013.25)

print(reading.to_fahrenheit())    # 74.3  -- geerbt!
print(reading.feels_like())       # 14.89 -- geerbt!
print(reading.city)               # Aalen -- geerbt!

print(reading.uv_index)           # 6.2   -- neu!
print(reading.pressure)           # 1013.25 -- neu!
```

Sie haben **null** Umrechnungscode geschrieben. `AdvancedReading` hat alles kostenlos bekommen.

Aber was, wenn die erweiterte Version `feels_like` anders berechnen muss?

Überschreiben, was sich ändern muss

Die erweiterte Station soll den UV-Index berücksichtigen. Definieren Sie die Methode einfach neu:

```
class AdvancedReading(WeatherReading):  
    # ... __init__ wie zuvor ...  
  
    def feels_like(self):  
        base = self.temperature - (self.wind_speed * 0.7)  
        uv_adjustment = self.uv_index * 0.5  
        return base + uv_adjustment
```

Gleicher Methodenname, **anderes Verhalten** im Kind. Die Version des Elternteils wird ersetzt.

`to_fahrenheit()` bleibt unverändert geerbt -- **nur ändern, was sich ändern muss.**

Die Magie: Eine Schleife, viele Typen

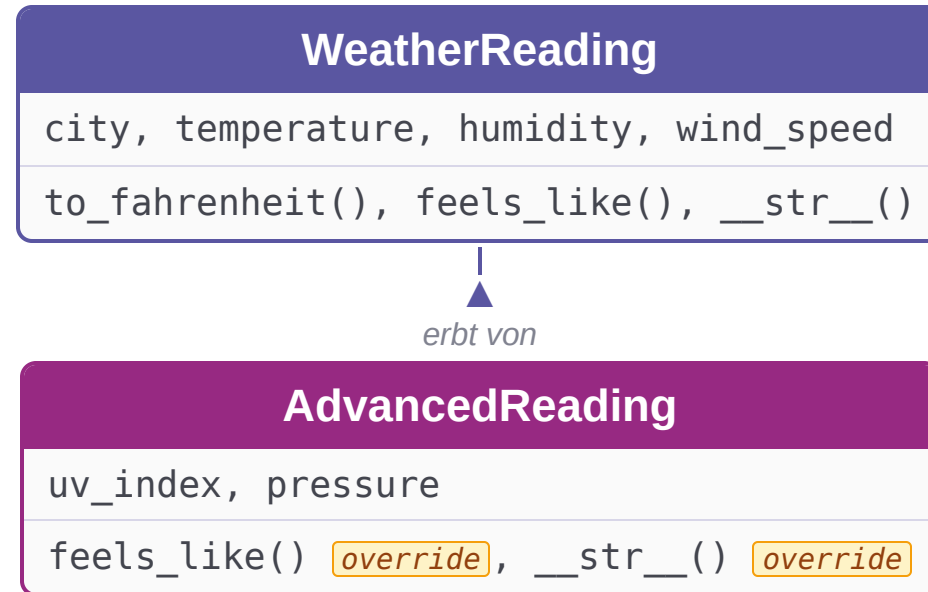
Mischen Sie Basis- und erweiterte Messungen in derselben Liste:

```
stations = [  
    WeatherReading("Aalen", 23.5, 65.0, 12.3),  
    AdvancedReading("Berlin", 19.0, 72.0, 8.5, 3.1, 1015.0),  
    WeatherReading("Munich", 21.3, 58.0, 15.1),  
]  
  
for station in stations:  
    print(station.city, "- feels like:", station.feels_like())
```

Python ruft **automatisch die richtige Version** von `feels_like()` auf. Basisstationen bekommen Windchill. Erweiterte Stationen bekommen UV-Anpassung. Gleicher Code, anderes Verhalten.

Das ist **Polymorphismus** -- griechisch für "viele Formen."

Vererbungsdiagramm



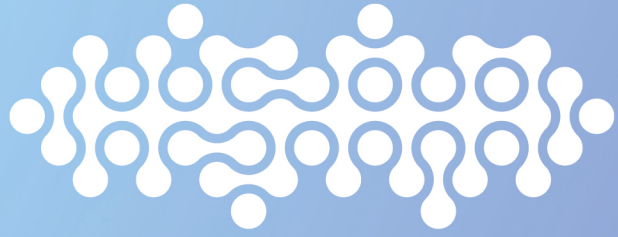
Diskussion: Wo begegnet uns das noch?

Polymorphismus ist überall:

- Ein **Medioplayer** ruft `.play()` auf bei Audio, Video, Streams -- gleiche Schnittstelle
- Ein **Bezahlungssystem** ruft `.charge()` auf bei Kreditkarten, PayPal, Krypto
- Eine **Game Engine** ruft `.render()` auf bei Sprites, 3D-Modellen, Partikeln

Gleicher Methodename. Anderes Verhalten. Der Aufrufer muss sich nicht darum kümmern.

Fallen Ihnen weitere Beispiele aus Ihrem Alltag ein?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Bibliotheken: Das Rad nicht neu erfinden

8

Ein Geheimnis, das jeder erfahrene Programmierer kennt

Der meiste Code, den Sie brauchen, wurde bereits von jemand anderem geschrieben.

Erinnern Sie sich an die Schleife, die wir für den Durchschnitt geschrieben haben? Es geht besser:

```
import statistics

daily_temps = [18.2, 21.0, 19.5, 23.5, 25.1, 22.8, 20.3]

print(statistics.mean(daily_temps))    # 21.49
print(statistics.median(daily_temps))  # 21.0
print(statistics.stdev(daily_temps))   # 2.34
```

Eine Zeile statt fünf. Und sie behandelt Sonderfälle, die unsere Schleife nicht abdeckte.

Auf den Schultern von Riesen stehen

Jenseits der Standardbibliothek -- Hunderttausende Community-Pakete:

```
$ uv add numpy pandas matplotlib pytest
```

Diese werden Sie im Kurs häufig antreffen:

- **NumPy** -- schnelle numerische Berechnungen
- **Pandas** -- Datenanalyse und -manipulation
- **Pytest** -- Testen Ihres Codes (werden Sie **sehr oft** nutzen)

Programmieren heißt nicht, alles von Grund auf zu schreiben. Es heißt, Ihren Code mit bestehenden Werkzeugen zu kombinieren, um Probleme effizient zu lösen.

Ihr eigener Code wird auch zur Bibliothek

Wenn Ihre Datei lang wird, teilen Sie sie auf. Gleiche Idee -- einmal definieren, überall wiederverwenden:

weather.py:

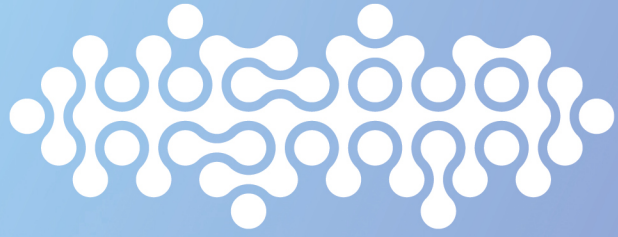
```
class WeatherReading:
    def __init__(self, city, temp,
                  hum, wind):
        self.city = city
        # ...
    def to_fahrenheit(self):
        return self.temperature * 9/5 + 32
```

main.py:

```
from weather import WeatherReading

aalen = WeatherReading(
    "Aalen", 23.5, 65.0, 12.3)
print(aalen.to_fahrenheit())
```

Dasselbe `import` wie bei `statistics` -- nur dass die Bibliothek **Ihr eigener Code** ist.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Das große Ganze

9

Jedes Konzept wurde von einer Mauer erzwungen

 **Einfache Berechnungen**
Variablen + Operatoren

Zu viel Wiederholung ▼

 **Funktionen**
wiederverwendbare Logikblöcke

Zu viele lose Parameter ▼

 **Klassen**
zusammengehörige Daten gruppieren

Funktionen getrennt von Daten ▼

 **Methoden**
Verhalten innerhalb von Klassen

Erweitern ohne Neuschreiben ▼

 **Vererbung & Polymorphismus**
auf Bestehendem aufbauen

Das Rad nicht neu erfinden ▼

 **Bibliotheken & Imports**
Arbeit anderer wiederverwenden

Das ist kein Lehrtrick

Diese Progression spiegelt wider, wie sich **das Fachgebiet selbst** über Jahrzehnte entwickelt hat:

- Frühe Programme waren nur Folgen von Berechnungen
- Funktionen wurden erfunden, um Komplexität zu bewältigen
- Klassen entstanden, als Programme zu groß für lose Daten wurden
- Bibliotheken entstanden, weil niemand dasselbe Problem zweimal lösen will

Sie sind gerade denselben Weg gegangen -- an einem Nachmittag.

Und das ist wirklich alles, was Sie über Programmierung für diesen Kurs wissen müssen. Die Syntax wird durch Übung zur Gewohnheit.

Eigenständig üben

Bereit, alles Gelernte anzuwenden? Öffnen Sie das eigenständige Übungs-Notebook und bauen Sie ein komplettes Wetter-Dashboard von Grund auf:

Weather Dashboard Notebook

- Läuft im Browser -- keine Installation nötig
- Analysieren Sie eine Woche Wetterdaten aus drei Städten
- Füllen Sie den Code aus, führen Sie die Tests aus, erhalten Sie sofortiges Feedback
- Geschätzte Zeit: **20--30 Minuten**

Was als Nächstes kommt

Sie müssen **nicht** alles auswendig lernen. Dies hat Ihnen die **konzeptuelle Landkarte** gegeben.

- **Vorlesung 1** -- Python, Git, VS Code einrichten. Echter Code vom ersten Tag an.
- **Anhang 2** -- tieferer Einblick in Imports, Module, Pakete
- **Jede Vorlesung** baut auf den heutigen Konzepten auf

Die wichtigste Erkenntnis:

Programmieren ist **strukturiertes Problemlösen**. Jedes Konzept, das Sie heute gelernt haben, existiert, weil jemand an eine Mauer gestoßen ist -- und einen besseren Weg gefunden hat. Behalten Sie diese Denkweise bei, und die Syntax folgt von selbst.

Kurzreferenz Spickzettel

Variable:

```
name = "Aalen"
```

Liste:

```
temps = [18.2, 21.0]
```

Funktion:

```
def c_to_f(c):  
    return c * 9/5 + 32
```

Klasse + Methode:

```
class WeatherReading:  
    def __init__(self, city, temp):  
        self.city = city  
        self.temperature = temp  
    def to_f(self):  
        return self.temperature * 9/5 + 32
```

Import:

```
import statistics  
from datetime import datetime
```



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Fragen?

VIELEN DANK

Ihr Feedback ist willkommen.