



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Appendix 0 Part 2

INTRODUCTION TO
PROGRAMMING WITH PYTHON

What This Session Is About

Part 1 taught you how to **talk to your computer** through a terminal.

Now we answer a bigger question: **why would you want to?**

We will build one thing -- a weather station -- from scratch. Along the way, you will hit walls. Each wall will force you to reach for a new tool. By the end, you will have the complete mental map of how programmers think.

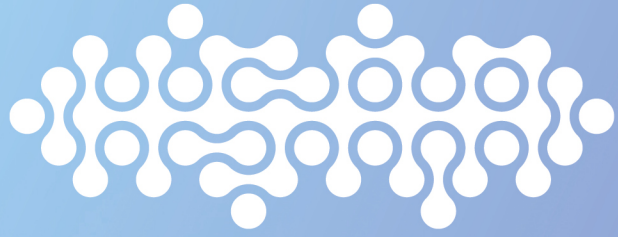
You do not need to memorize syntax. You need to build intuition.

Quick Poll

How would you rate your Python experience?

- **A)** Never written a line of Python
- **B)** Done a tutorial or two, but not confident
- **C)** Can write simple scripts
- **D)** Comfortable with functions and classes

No wrong answer! This session builds from zero.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why Do People Write Programs?

1

Every App Starts with a Problem

Think about the apps on your phone right now.

- **Instagram** -- someone wanted a simple way to share photos with filters
- **Spotify** -- someone wanted music accessible without piracy
- **Google** -- someone wanted to find things in a growing ocean of web pages

These are all **problems** that were solved with **programs**.

And it is not just consumer apps -- science, engineering, medicine, finance, art.

Programming is everywhere.

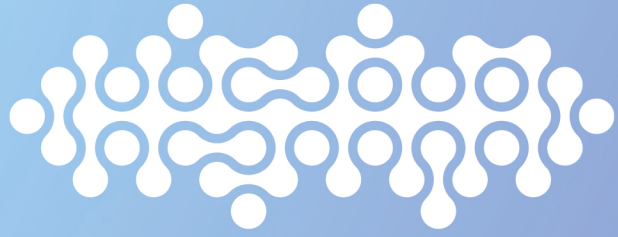
From Big Ideas to Small Steps

You cannot build Netflix on day one. Netflix's engineers did not either.

The trick: find problems that are **small enough to solve today** but **meaningful enough to care about**.

Our problem today: **a weather station**. Everyone understands weather. It involves numbers, text, and decisions. Simple to start with, rich enough to show why each new idea matters.

Let us begin.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Why Python?

2

Hundreds of Languages -- Why This One?

- 1. No compilation step.** Many languages make you compile first, then run. Python skips the middle step. You write code and run it immediately. Experimenting is fast and painless.
- 2. Built for humans.** Guido van Rossum (1991): code should be easy to read, easy to write, and quick to turn an idea into a working program.
- 3. The language of science and AI.** NumPy, Pandas, TensorFlow, PyTorch -- all Python-first. If you work with data or AI, you will encounter Python.

Write It. Run It. Done.

C / Java (compiled):

1. Write code
2. Compile it
3. Run the binary

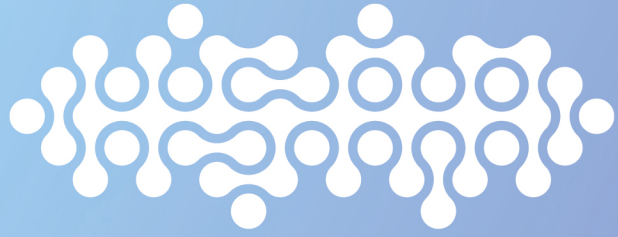
```
$ gcc main.c -o main  
$ ./main
```

Python (interpreted):

1. Write code
2. Run it!

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

No compiler. No build step. **Idea to result in seconds.** That is why we use it.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

It's a Computer -- Let It Compute

3

The Building Blocks: Variables

Any problem you solve with code starts with data. A **variable** is a name attached to a value -- a labeled box. You choose the name, Python remembers the value.

```
temperature_celsius = 23.5      # float (decimal number)
number_of_sensors = 4           # int (whole number)
city = "Aalen"                  # str (text, a "string")
is_raining = False              # bool (True or False)
```

Four types is all you need right now: `int`, `float`, `str`, `bool`.

If you have done any math at all, you already understand these.

It's a Computer -- Of Course It Can Do Math

The machine in front of you was literally built to calculate. So let it:

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

You already know arithmetic. Python does too: `+` `-` `*` `/` `//` `%` `**`

And it can compare: `>` `<` `==` `>=` `<=` `!=` -- the result is always `True` or `False`.

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

Great. We can store a value and compute with it. But one value is not very useful...

One Value Is Lonely -- Enter Lists

A weather station does not record one temperature. It records one **every hour**. You need a way to store **multiple values together**.

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

print(daily_temps[0])    # 18.2 (counting starts at 0!)
print(len(daily_temps))  # 7
```

And you can loop through them:

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

Lists are powerful containers. But as your code grows, containers alone are not enough. Let us see why.

The Wall: Copy-Paste Is Killing Us

You monitor five cities. You need to convert each temperature:

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

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

# ... same formula 3 more times
```

The same formula appears **five times**. Now imagine:

- What if you had **100 cities**?
- What if the formula needed to **change**?
- What if you made a **typo** in one line but not the others?

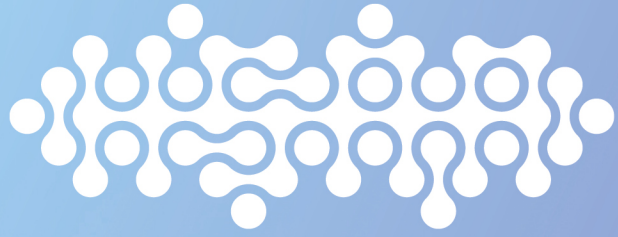
Copy-pasting is a recipe for bugs. There must be a better way.

Hands-On: Feel the Problem

Open the [Guided Hands-On Notebook](#) and go to **Sections 1-2**:

1. Create a variable `celsius` with today's approximate temperature
2. Convert it to Fahrenheit: `fahrenheit = celsius * 9 / 5 + 32`
3. Create a list of five different temperatures
4. Convert **each one** to Fahrenheit -- notice how annoying the repetition feels

You have 3 minutes. Feel the pain. The next chapter is the cure.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Functions: Reuse Instead of Repetition

4

You Already Know Functions

In math class, your teacher wrote: $f(x) = x \cdot \frac{9}{5} + 32$

You did not rewrite the formula every time. You just called $f(23.5)$ and got 74.3.

Python works the same way:

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

`def` = "define a function." One input, one output. **The formula exists once.**

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

The Repetition Problem -- Solved

Remember those five copy-pasted conversions? Now combine a function with a list:

```
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]))
```

Five cities or five thousand -- **the code stays the same length.**

Change the formula in one place, it updates everywhere. That is the power.

Functions Can Do More

Multiple inputs, multiple outputs -- no 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)
```

Functions are a **huge step forward**. They eliminate repetition and make code readable.

But as your weather station grows, you will hit a new wall...

The Wall: Too Many Loose Values

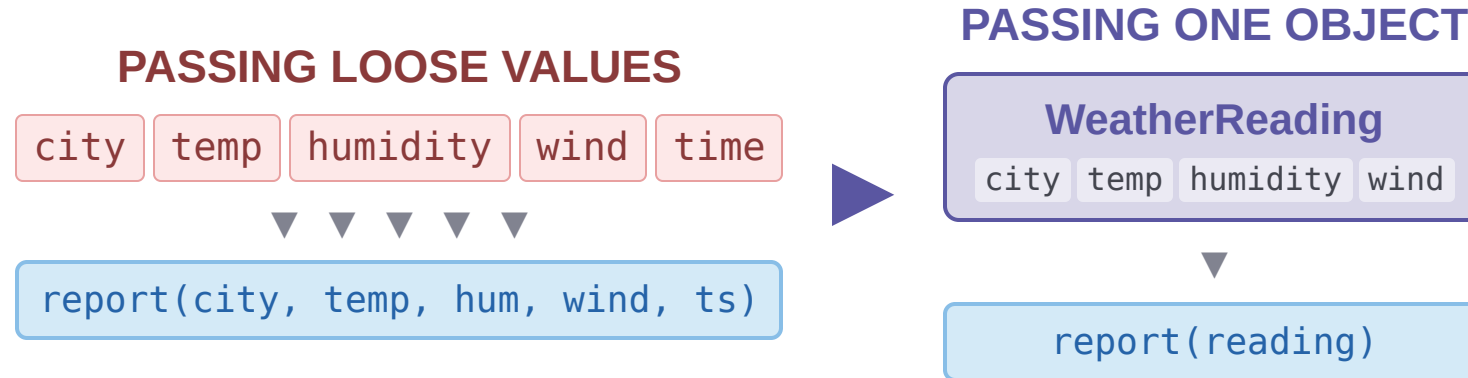
Your station tracks temperature, humidity, wind speed, city, timestamp. Now every function needs **all of them**:

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

The values `23.5`, `65.0`, `12.3` **belong together** -- they are all one reading. But we pass them as separate, loose values.

What if you accidentally swap humidity and wind speed? **Python will not complain. Your results will just be wrong.**

What We Want Instead



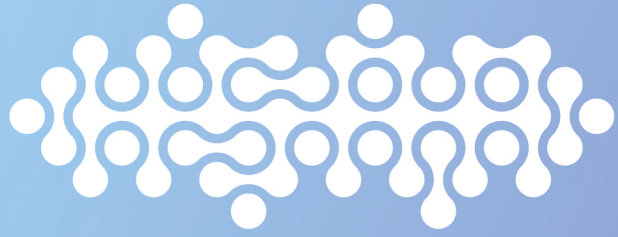
When values **belong together**, they are asking to be **grouped together**.

Hands-On: Feel the Loose-Value Pain

Open the [Guided Hands-On Notebook](#) and go to **Section 3**:

1. Write `celsius_to_fahrenheit(celsius)` -- converts C to F
2. Write `feels_like(temp, humidity, wind)` -- returns `temp - (wind * 0.7)`
3. Call **both** for three different cities
4. Count how many times you type the same numbers

You have 3 minutes. Notice the repetition -- same values, over and over.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Classes: Grouping What Belongs Together

5

Think of a Blank Form

A **class** is a blank form -- a template with labeled fields. Every time you fill one out, you create an **instance**: one specific reading with concrete values.

```
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` -- define the form
- `__init__` -- runs when you fill it out
- `self` -- "this particular copy"

Data That Belongs Together Now Lives Together

```
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
```

And the messy five-parameter function? **One parameter now:**

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

report_weather(aalen)  # clean, no chance of swapping values
```

That is a real improvement. But we are not done yet...

The Wall: Functions Still Live Outside

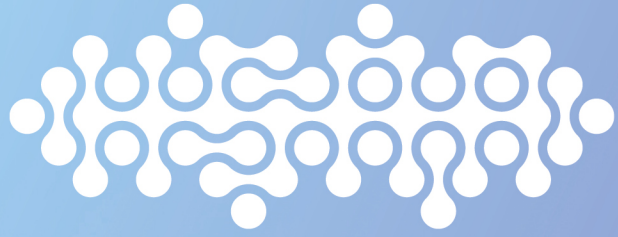
We have a `WeatherReading` class. And we have standalone functions:

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

But `celsius_to_fahrenheit` **only makes sense** for a weather reading. And `feels_like` pulls three fields out of the object we just carefully grouped together.

Why keep the functions **separate** from the data they operate on?

Would it not make more sense to put them **inside** the class?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Methods: Behavior Meets Data

6

Move the Functions Inside

A **method** is a function that lives inside a class. It already has access to the data through `self` -- no need to pass anything in.

```
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)
```

Look How Clean This Is Now

Before (standalone):

```
# You must remember the order  
feels_like(23.5, 65.0, 12.3)
```

Error-prone, clumsy.

After (method):

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

The object **knows its own data**.

Process a whole list -- elegantly:

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

Making Objects Printable: `__str__`

What happens when you try `print(aalen)` ?

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

Not helpful. Python does not know how to display your object. You can teach it by defining `__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__` -- these are **dunder methods** (double underscore). They customize how Python handles your objects.

Where We Are Now

WeatherReading

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

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

Data and behavior, **together in one place**. Beautiful. But there is one more wall...

The Wall: A New Kind of Station

Your city installs **advanced** weather stations. They also measure UV index and air pressure. You need a new class -- but it should still do **everything** `WeatherReading` already does.

Do you copy-paste the entire class and add two fields?

That would work. But you would have **two copies** of the same code. Fix a bug in one, forget the other. We have seen this movie before.

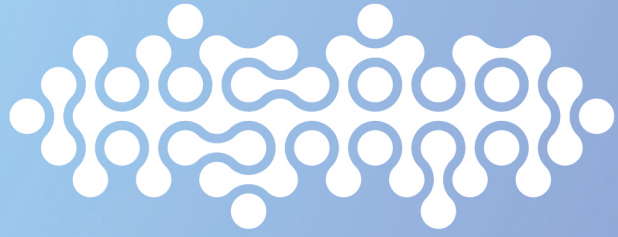
There must be a way to say: "do everything the old class does, plus more."

Hands-On: Build a WeatherReading

Open the [Guided Hands-On Notebook](#) and go to **Sections 4-5**:

1. Define `WeatherReading` with `city`, `temperature`, `humidity`, `wind_speed`
2. Add `to_fahrenheit(self)` and `feels_like(self)` as methods
3. Add `is_comfortable(self)`: `True` if $18 \leq \text{temp} \leq 26$ and `humidity < 70`
4. Create three instances, put them in a list, loop and print

You have 5 minutes. This is the core pattern of the whole course.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Inheritance: Extend Without Rewriting

7

Build On What Already Works

```
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)` -- "inherits everything from the parent"
- `super().__init__(...)` -- let the parent set up its own fields
- We only add what is **new**: `uv_index` and `pressure`

Zero copy-pasting. Zero duplicated code.

It Just Works

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

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

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

You wrote **zero** conversion code. `AdvancedReading` got it all for free.

But what if the advanced version needs to calculate `feels_like` differently?

Override What Needs to Change

The advanced station should factor in UV index. Just redefine the method:

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

Same method name, **different behavior** in the child. The parent's version is replaced.

`to_fahrenheit()` stays inherited unchanged -- **only change what needs changing.**

The Magic: One Loop, Many Types

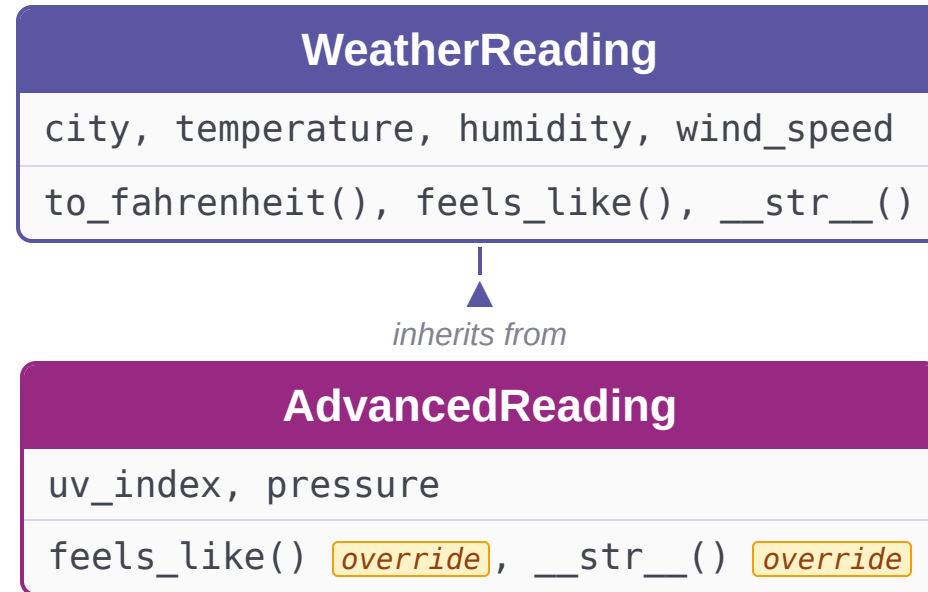
Mix basic and advanced readings in the same list:

```
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 calls the **right version** of `feels_like()` automatically. Basic stations get wind chill. Advanced stations get UV-adjusted. Same code, different behavior.

This is **polymorphism** -- Greek for "many forms."

Inheritance Diagram



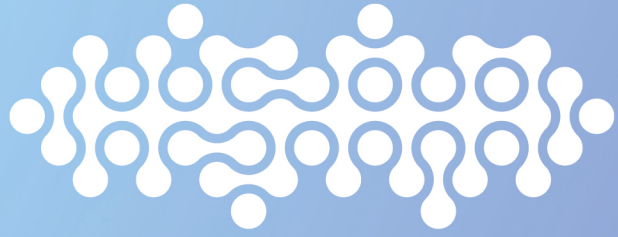
Discussion: Where Else Does This Happen?

Polymorphism is everywhere:

- A **media player** calls `.play()` on audio, video, streams -- same interface
- A **payment system** calls `.charge()` on credit cards, PayPal, crypto
- A **game engine** calls `.render()` on sprites, 3D models, particles

Same method name. Different behavior. The caller does not need to care.

Can you think of other examples from your daily life?



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

Libraries: Don't Reinvent the Wheel

8

A Secret Every Experienced Programmer Knows

Most of the code you need has already been written by someone else.

Remember the loop we wrote to compute the average? There is a better way:

```
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
```

One line instead of five. And it handles edge cases our loop did not.

Standing on the Shoulders of Giants

Beyond the standard library -- hundreds of thousands of community packages:

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

You will encounter these throughout the course:

- **NumPy** -- fast numerical computing
- **Pandas** -- data analysis and manipulation
- **Pytest** -- testing your code (you will use this **a lot**)

Programming is not about writing everything from scratch. It is about combining your code with existing tools to solve problems efficiently.

Your Own Code Becomes a Library Too

When your file gets long, split it. Same idea -- define once, reuse everywhere:

weather.py:

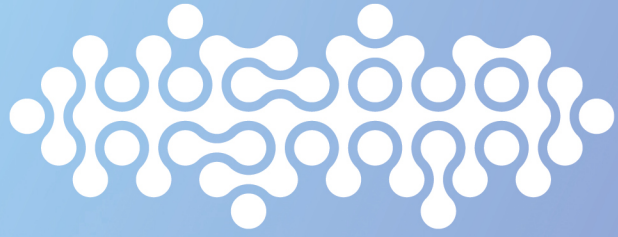
```
class WeatherReading:
    def __init__(self, city, temp,
                  hum, wind):
        self.city = city
        # ...
    def to_fahrenheit(self):
        return self.temperature * 9/5 + 32
```

main.py:

```
from weather import WeatherReading

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

Same `import` you used for `statistics` -- except the library is **your own code**.



Hochschule Aalen
UNIVERSITY OF APPLIED SCIENCES

The Big Picture

9

Every Concept Was Forced by a Wall

 **Raw calculations**
variables + operators

Too much repetition ▼

 **Functions**
reusable blocks of logic

Too many loose parameters ▼

 **Classes**
group related data

Functions separate from data ▼

 **Methods**
behavior inside classes

Want to extend without rewriting ▼

 **Inheritance & Polymorphism**
build on what exists

Don't reinvent the wheel ▼

 **Libraries & Imports**
reuse others' work

This Is Not a Teaching Trick

This progression reflects how **the field itself** evolved over decades:

- Early programs were just sequences of calculations
- Functions were invented to manage complexity
- Classes emerged when programs grew too large for loose data
- Libraries arose because no one wants to solve the same problem twice

You just walked the same path -- in one afternoon.

And that is genuinely all you need to know about programming for this course. The syntax will become second nature as you practice.

Practice on Your Own

Ready to apply everything you learned today? Open the independent practice notebook and build a complete weather dashboard from scratch:

Weather Dashboard Notebook

- Runs in your browser -- no installation needed
- Analyze one week of weather data from three cities
- Fill in the code, run the tests, get instant feedback
- Estimated time: **20--30 minutes**

What Comes Next

You do **not** need to memorize everything. This gave you the **conceptual map**.

- **Lecture 1** -- set up Python, Git, VS Code. Real code from day one.
- **Appendix 2** -- deeper dive into imports, modules, packages
- **Every lecture** builds on today's concepts

The most important takeaway:

Programming is **structured problem-solving**. Every concept you learned today exists because someone hit a wall -- and found a better way. Keep that mindset, and the syntax will follow.

Quick Reference Cheat Sheet

Variable:

```
name = "Aalen"
```

List:

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

Function:

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

Class + Method:

```
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

Questions?

THANK YOU

Your feedback is welcome.