



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG



CHAI

Humanities-Centered AI

# Understanding Data vs. Machine Training

**Malte Luttermann – Institute for Humanities-Centered  
Artificial Intelligence (CHAI)**

November 7, 2025

# Overview of Contents

- (1) Programming language Python
  - (a) Introduction and first steps
  - (b) Basics
  - (c) Advanced
- (2) Markup languages
  - (a)  $\text{\LaTeX}$
  - (b) Markdown
- (3) Development environments
  - (a) Jupyter notebooks
- (4) Version control
  - (a) Git and GitHub
- (5) Scientific computing
  - (a) NumPy and SciPy
- (6) Data processing and visualisation
  - (a) Pandas, matplotlib, and NLTK
- (7) Machine learning (scikit-learn)
  - (a) Basics (datasets, analysis)
  - (b) Simple methods (clustering, ...)
- (8) Deep learning
  - (a) PyTorch

# Topics for Today

## (1) Python advanced topics

- Magic Methods
- Generators
- Advanced loops
- Error handling
- Context managers
- Lambda functions
- Type annotations
- Variables and pointers
- Decorators



# Acknowledgement

- The upcoming slides are taken from the following lecture and have been translated and partially modified:
  - Dr. Magnus Bender: »[Python für Machine Learning und Data Science](#)« as part of the German course »[Werkzeuge für das wissenschaftliche Arbeiten](#)«

# Magic Methods

```
1 class Vector():
2     def __init__(self, a, b):
3         self.a = a
4         self.b = b
5
6     def __add__(self, o):
7         return Vector(self.a + o.a, self.b + o.b)
8
9     def __sub__(self, o):
10        return Vector(self.a - o.a, self.b - o.b)
11
12    def __iadd__(self, o):
13        self.a += o.a
14        self.b += o.b
15        return self
16
17    # def __isub__(self, o):
18
19    def __str__(self):
20        return "{}, {}".format(self.a, self.b)
```

```
1 x = Vector(1, 2)
2 print(x, str(x), x.__str__())
3 y = Vector(2, 3)
4 print(y)
5
6 print(x + y, x.__add__(y))
7 print(x - y)
8
9 x += y
10 print(x)
11
12 x -= y
13 print(x)
```

```
$> python3 script.py
```

```
(1, 2) (1, 2) (1, 2)
(2, 3)
(3, 5) (3, 5)
(-1, -1)
(3, 5)
(1, 2)
```

# Magic Methods

```
1 class Vector():
2     def __init__(self, a, b):
3         self.a = a
4         self.b = b
5
6     def __add__(self, o):
7         return Vector(self.a + o.a, self.b + o.b)
8
9     def __sub__(self, o):
10        return Vector(self.a - o.a, self.b - o.b)
11
12    def __iadd__(self, o):
13        self.a += o.a
14        self.b += o.b
15        return self
16
17    # def __isub__(self, o):
18
19    def __str__(self):
20        return "{}, {}".format(self.a, self.b)
```

```
1 x = Vector(1, 2)
2 print(x, str(x), x.__str__())
3 y = Vector(2, 3)
4 print(y)
5
6 print(x + y, x.__add__(y))
7 print(x - y)
8
9 x += y
10 print(x)
11
12 x -= y
13 print(x)
```

Why does `x -= y` work even though `__isub__` is not defined?

# Magic Methods

Math	<code>__add__</code>	<code>+</code>	Addition
	<code>__sub__</code>	<code>-</code>	Subtraction
	<code>__mul__</code>	<code>*</code>	Multiplication
	<code>__matmul__</code>	<code>@</code>	Matrix multiplication
	<code>__truediv__</code>	<code>/</code>	Division
	<code>__floordiv__</code>	<code>//</code>	Integer division
Logical	<code>__pow__</code>	<code>**</code>	Exponentiation
	<code>__and__</code>	<code>&amp;</code>	Bitwise AND
	<code>__xor__</code>	<code>^</code>	Bitwise XOR
	<code>__or__</code>	<code> </code>	Bitwise OR

Comparison	<code>__lt__</code>	<code>&lt;</code>	Less than
	<code>__le__</code>	<code>&lt;=</code>	Less than or equal
	<code>__eq__</code>	<code>==</code>	Equality
	<code>__ne__</code>	<code>!=</code>	Inequality
	<code>__gt__</code>	<code>&gt;</code>	Greater than
	<code>__ge__</code>	<code>&gt;=</code>	Greater than or equal
Classes	<code>__bool__</code>		Boolean of an object ( <code>if obj</code> )
	<code>__hash__</code>		Unique hash value of the object
	<code>__len__</code>		Length of the object
	<code>__str__</code>		String (output)
	<code>__del__</code>		Delete object ( <code>del obj</code> )

- Further information is available in the official Python documentation:
  - <https://docs.python.org/3/reference/datamodel.html>
  - <https://docs.python.org/3/library/operator.html#mapping-operators-to-functions>

## Magic Methods: dict and list

```
1 class DictContainer():
2     def __init__(self):
3         self.dict = {}
4
5     def __setitem__(self, key, value):
6         self.dict[key] = value
7
8     def __getitem__(self, key):
9         return self.dict[key]
10
11    def __delitem__(self, key):
12        del self.dict[key]
13
14    def __contains__(self, key):
15        return key in self.dict
16
17    def __str__(self):
18        return str(self.dict)
```

```
1 dc = DictContainer()
2
3 dc["a"] = "A"
4 dc.__setitem__("b", "B")
5 print(dc)
6
7 print(dc["a"])
8 print(dc.__getitem__("b"))
9
10 del dc["a"]
11 print(dc)
12
13 print("2" in dc, "b" in dc)
```

```
$> python3 script.py
```

```
{'a': 'A', 'b': 'B'}
A
B
{'b': 'B'}
False True
```

# Generators

```
1 import timeit
2
3 lc_a = """
4 [i for i in range(2**10)]
5 """
6 print(timeit.timeit(lc_a, number=20))
7
8 lc_b = """
9 [i for i in range(2**20)]
10 """
11 print(timeit.timeit(lc_b, number=20))
```

```
1 g_a = """
2 (i for i in range(2**10))
3 """
4 print(timeit.timeit(g_a, number=20))
5
6 g_b = """
7 (i for i in range(2**20))
8 """
9 print(timeit.timeit(g_b, number=20))
```

On the left, the run time heavily depends on the number of elements, on the right it does not. Why?

```
$> python3 script.py
```

```
0.0006132079999999988
0.8152710830000001
1.0999999999983245e-05
1.154199999997516e-05
```

# Generators

- Generator expressions do not compute the entire sequence immediately
  - Generator expression: `()`
- List comprehensions compute and store the entire list immediately
  - List comprehension: `[]`

# Generators: The `yield` Keyword

```
1 def list_range(until):  
2     l = []  
3     i = 0  
4     while i < until:  
5         l.append(i)  
6         i += 1  
7     return l  
  
9 print(list_range(20))  
10 for i in list_range(5):  
11     print(i)
```

```
$> python3 script.py
```

```
[0, 1, 2, 3, 4, 5, 6, ..., 15, 16, 17, 18, 19]
```

```
0  
1  
2  
3  
4
```

```
1 def generate_range(until):  
2     i = 0  
3     while i < until:  
4         yield i  
5         i += 1  
  
7 print(generate_range(20))  
8 for i in generate_range(5):  
9     print(i)
```

```
$> python3 script.py
```

```
<generator object generate_range at 0x7fa0e00b2a50>
```

```
0  
1  
2  
3  
4
```

## Generators: The `yield` Keyword

```
1 def list_range(until):  
2     l = []  
3     i = 0  
4     while i < until:  
5         l.append(i)  
6         i += 1  
7     return l  
  
9 print(list_range(20))  
10 for i in list_range(5):  
11     print(i)
```

The entire list with all elements is created and returned

```
1 def generate_range(until):  
2     i = 0  
3     while i < until:  
4         yield i  
5         i += 1  
  
7 print(generate_range(20))  
8 for i in generate_range(5):  
9     print(i)
```

Only the next value is generated, returned, processed, and then the next one is generated

# Iteration with Generators

```
1 class FileReader():
2     def __init__(self, filename):
3         self.filename = filename
4
5     def __iter__(self):
6         t = open(self.filename + ".titles.txt", "r")
7         c = open(self.filename + ".contents.txt", "r")
8         t_l = t.readline()
9         c_l = c.readline()
10
11        while t_l and c_l:
12            yield t_l.strip(), c_l.strip().split(",")
13            t_l = t.readline()
14            c_l = c.readline()
15
16        t.close()
17        c.close()
18
19 fr = FileReader("example")
20 for t, c in fr:
21     print(t, c)
```

Implicit call of magic method `__iter__`

```
1 Audi ,BMW ,Ford
2 Apple ,Banana ,Pear
3 Mouse ,Screen ,Keyboard
```

example.contents.txt

```
1 Car
2 Fruit
3 Computer
```

example.titles.txt

```
$> python3 script.py
```

```
Car ['Audi', 'BMW', 'Ford']
```

```
Fruit ['Apple', 'Banana', 'Pear']
```

```
Computer ['Mouse', 'Screen', 'Keyboard']
```

Even for large files the memory consumption remains constant

# Advanced Loops

- List comprehensions for dictionaries

```
1 d = {i : i**2 for i in range(5)}  
2 print(d)
```

```
$> python3 script.py
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

# Advanced Loops

## ■ `break` and `continue`

```
1 for i in range(20):  
2     if i < 2:  
3         continue  
4     elif i > 5:  
5         break  
6     print(i)
```

```
$> python3 script.py
```

```
2  
3  
4  
5
```

# Advanced Loops

## ■ else after loops

```
1 for i in range(4):  
2     if i == 5:  
3         print("Found")  
4         break  
5 else:  
6     print("Not found!")
```

```
$> python3 script.py
```

```
Not found!
```

```
1 for i in range(8):  
2     if i == 5:  
3         print("Found")  
4         break  
5 else:  
6     print("Not found!")
```

```
$> python3 script.py
```

```
Found
```

else after a loop is executed if the loop has finished without being prematurely terminated

# Error Handling

```
1 def divide(x, y):  
2     try:  
3         r = x / y  
4     except ZeroDivisionError:  
5         print("Division by zero!")  
6     except TypeError as e:  
7         print("TypeError:", e)  
8     else:  
9         print("{x} / {y} = {r}".format(x=x, y=y, r=r))  
10    finally:  
11        print("Finally done ;)")  
12  
13 divide(1, 2)  
14 divide(1, 0)  
15 divide("A", 2)
```

```
$> python3 script.py
```

```
1 / 2 = 0.5
```

```
Finally done ;)
```

```
Division by zero!
```

```
Finally done ;)
```

```
TypeError: unsupported
```

```
operand type(s) for /:
```

```
'str' and 'int'
```

```
Finally done ;)
```

<https://docs.python.org/3/tutorial/errors.html#defining-clean-up-actions>

# Error Handling

```
1 def divide(x, y):
2     try:
3         r = x / y
4     except ZeroDivisionError:
5         print("Division by zero!")
6     except TypeError as e:
7         print("TypeError:", e)
8     else:
9         print("{x} / {y} = {r}".format(x=x, y=y, r=r))
10    finally:
11        print("Finally done ;)")
12
13 divide(1, 2)
14 divide(1, 0)
15 divide("A", 2)
```

Distinction between different error types possible

else-block is executed if no error occurred

finally is always executed, regardless of whether an error occurred or not

# Error Handling

Why might such an error class without an implementation be useful?

- Errors can be raised via `raise`
- Errors are objects of the class (or a subclass of) `Exception`

```
1 class MyError(Exception):  
2     pass  
3  
4 raise MyError("Stopp")
```

- Pre-defined errors:  
<https://docs.python.org/3/library/exceptions.html#concrete-exceptions>
- Custom errors can be defined as a subclass of `Exception`

```
$> python3 script.py  
  
Traceback (most recent call last):  
  File "script.py", line 4, in <module>  
    raise MyError("Stopp")  
__main__.MyError: Stopp
```

# Context Managers

```
1 try:
2     t = open("example.titles.txt", "r")
3     t.write("hello")
4     t.close()
5 except BaseException as e:
6     print(e)
7
8 print(t.readline())
9
10 try:
11     with open("example.contents.txt", "r") as c:
12         c.write("hello")
13 except BaseException as e:
14     print(e)
15
16 print(c.readline())
```

What is the problem here?

# Context Managers

```
1 try:
2     t = open("example.titles.txt", "r")
3     t.write("hello")
4     t.close()
5 except BaseException as e:
6     print(e)
7
8 print(t.readline())
9
10 try:
11     with open("example.contents.txt", "r") as c:
12         c.write("hello")
13 except BaseException as e:
14     print(e)
15
16 print(c.readline())
```

```
$> python3 script.py
```

```
not writable
```

```
Car
```

```
not writable
```

```
Traceback (most recent call last):
```

```
File "script.py", line 16, in <module>
```

```
print(c.readline())
```

```
ValueError: I/O operation on closed file.
```

# Context Managers

```
1 try:
2     t = open("example.titles.txt", "r")
3     t.write("hello")
4     t.close()
5 except BaseException as e:
6     print(e)
7
8 print(t.readline())
9
10 try:
11     with open("example.contents.txt", "r") as c:
12         c.write("hello")
13 except BaseException as e:
14     print(e)
15
16 print(c.readline())
```

If `t.write()` raises an error, then `t.close()` is never called and the file remains open!

# Lambda Functions

Very useful, e.g., for sort (key=`lambda o: o.name`)

```
1 def pow2(number):  
2     return number ** 2  
3  
4 numbers = [1, 2, 3, 4]  
5 for i, n in enumerate(numbers):  
6     numbers[i] = pow2(n)  
7 print(numbers)  
8  
9 numbers = [1, 2, 3, 4]  
10 numbers = [ pow2(n) for n in numbers ]  
11 print(numbers)  
12  
13 numbers = [1, 2, 3, 4]  
14 numbers = map(pow2, numbers)  
15 print(numbers, list(numbers))  
16  
17 numbers = [1, 2, 3, 4]  
18 numbers = map(lambda n : n**2, numbers)  
19 print(numbers, list(numbers))
```

```
$> python3 script.py
```

```
[1, 4, 9, 16]
```

```
[1, 4, 9, 16]
```

```
<map object at 0x7f93701affa0> [1, 4, 9, 16]
```

```
<map object at 0x7f93701affd0> [1, 4, 9, 16]
```

`map()` uses an iterator internally and thus the result is not a list

```
def pow2_f(number):  
    return number ** 2
```

is equivalent to

```
pow2_l = lambda n: n ** 2
```

# Map, Filter, and Reduce

- `map()` applies a function to every item of a list and returns an iterator

```
1 numbers = [1, 2, 3, 4]
2
3 m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
4 print(list(m))
5
6 [n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
$> python3 script.py
```

```
[1, 8, 9, 16]
```

## Map, Filter, and Reduce

- `filter()` constructs an iterator from elements of a list for which a specified function returns `True`

```
1 numbers = [1, 2, 3, 4]
2
3 f = filter(lambda n: n % 2 == 0, numbers)
4 print(list(f))
5
6 [n for n in numbers if n % 2 == 0]
```

```
$> python3 script.py
```

```
[2, 4]
```

## Map, Filter, and Reduce

- `reduce()` cumulatively applies a function to the elements of a list, reducing the list to a single value

```
1 from functools import reduce
2
3 numbers = [1, 2, 3, 4]
4 r = reduce(lambda c, n: c + n, numbers, 0)
5 print(r)
6
7 c = 0
8 for n in numbers:
9     c = c + n
```

```
$> python3 script.py
```

```
10
```

# Type Annotations

```
1 def str_repeat(s:str, n:int, sep:str=None) -> str:  
2     s = s + sep if sep else s  
3     return s * n  
4  
5 print(str_repeat("Hello", 5))  
6 print(str_repeat("Hello", 2, ", "))  
7 print(str_repeat(5, 5))
```

What happens when you run this script?

# Type Annotations

```
1 def str_repeat(s:str, n:int, sep:str=None) -> str:  
2     s = s + sep if sep else s  
3     return s * n  
4  
5 print(str_repeat("Hello", 5))  
6 print(str_repeat("Hello", 2, ", "))  
7 print(str_repeat(5, 5))
```

```
$> python3 script.py  
HelloHelloHelloHelloHello  
Hello, Hello,  
25
```

- Type annotations are just *hints*
- Type hints are not enforced at runtime!

# Type Annotations

- With the external package `pydantic`, a type check is possible

```
1 from pydantic import validate_arguments
2
3 @validate_arguments
4 def str_repeat(s:str, n:int, sep:str=None) -> str:
5     s = s + sep if sep else s
6     return s * n
7
8 print(str_repeat("Hello", 5))
9 print(str_repeat("Hello", 2, ", "))
10 print(str_repeat(5, 5))
```

# Type Annotations

```
1 from typing import NoReturn, Union, Optional
2
3 def hello(name:Optional[str]="") -> NoReturn:
4     print("Hello " + name)
5
6 def divide(x:Union[int, float], y:int|float) -> float:
7     return x / y
```

```
1 from typing import List, Dict, Tuple
2
3 Tuple[int, str] # (1, "a")
4 Tuple[int, ...] # (1, 1), (1, 2, 3)
5
6 List[str] # ["a", "b"]
7 List[Union[str, int]] # [1, "b"], ["a", 2]
8
9 Dict[str, str] # {"a" : "A", "b" : "B"}
```

It is also possible to use a custom class in a type annotation

Type hints are extremely useful to document code

## Function Parameters `**` and `*`

```
1 def example(*args, **kwargs):  
2     print(type(args), args)  
3     print(type(kwargs), kwargs)  
4  
5 example("A", "B", "C", d="D", e="E")  
6  
7 p = "A", "B", "C"  
8 kp = {"d": "D", "e": "E"}  
9 example(*p, **kp)
```

```
$> python3 script.py  
  
<class 'tuple'> ('A', 'B', 'C')  
<class 'dict'> {'d': 'D', 'e': 'E'}  
  
<class 'tuple'> ('A', 'B', 'C')  
<class 'dict'> {'d': 'D', 'e': 'E'}
```

Extremely useful to pass parameters to a superclass

# Variables and Pointers

- Python uses *call by reference*
  - Call by reference: Function calls pass a reference (pointer) to a variable
  - Call by value: Function calls pass the value of the variable
  - Many types are immutable (e.g., `str`, `int`, `float`, `tuple`)
  - Changing an immutable type always creates a new object (call by value)

```
1 s = "Hello"  
2 print(id(s))  
3 s += "world"  
4 print(id(s))
```

```
$> python3 script.py
```

```
140366108394096  
140366108907568
```

```
1 l = ["Hello"]  
2 print(id(l))  
3 l.append("world")  
4 print(id(l))
```

```
$> python3 script.py
```

```
140366108394048  
140366108394048
```

# Variables and Pointers

```
1 def change(l):  
2     for i in range(len(l)//2):  
3         l[i], l[-(i+1)] = l[-(i+1)], l[i]  
4  
5 l = [1, 2, 3, 4, 5]  
6 change(l)  
7 print(l)
```

```
$> python3 script.py
```

```
[5, 4, 3, 2, 1]
```

`change(l)` changes the list `l` without returning a value (because a pointer is passed)

## Variables and Pointers

```
1 bools = [True] * 5
2 print(bools)
3 bools[2] = False
4 bools[3] = False
5 print(bools)
6
7 lists = [[]] * 5
8 print(lists)
9 lists[2].append(2)
10 lists[3].append(3)
11 print(lists)
```

```
$> python3 script.py
```

```
[True, True, True, True, True]
[True, True, False, False, True]
[[], [], [], [], []]
[[2, 3], [2, 3], [2, 3], [2, 3], [2, 3]]
```

Why do all values change?

It is also possible to change mutable types within an immutable type:

```
a = ([], [])
a[0].append(2)
```

# Decorators

```
1 def my_decorator(some_function):  
2     print("Load function")  
3  
4     def my_wrapper(*args, **kwargs):  
5         print("Before execution of the function")  
6         v = some_function(*args, **kwargs)  
7         print("After execution of the function")  
8         return v  
9  
10    return my_wrapper  
11  
12 @my_decorator  
13 def example(name):  
14     print("Hello " + name)  
15  
16 example("Malte")  
17 example("Alice")
```

```
$> python3 script.py
```

```
Load function  
Before execution of the function  
Hello Malte  
After execution of the function  
Before execution of the function  
Hello Alice  
After execution of the function
```

# Decorators

```
1 def my_decorator(some_function):  
2     print("Load function")  
3  
4     def my_wrapper(*args, **kwargs):  
5         print("Before execution of the function")  
6         v = some_function(*args, **kwargs)  
7         print("After execution of the function")  
8         return v  
9  
10    return my_wrapper  
11  
12 @my_decorator  
13 def example(name):  
14     print("Hello " + name)  
15  
16 example("Malte")  
17 example("Alice")
```

Decorators can also have parameters themselves

```
@my_decorator:  
Load function
```

```
example("Malte"):  
Before execution of the function  
Hello Malte  
After execution of the function
```

```
example("Alice"):  
Before execution of the function  
Hello Alice  
After execution of the function
```

## Abstract Classes with Decorators

- Are there abstract classes in Python?
  - Not really, but ...
  - ... abstract classes can be defined using decorators (the package `abc`)

```
1 from abc import ABC, abstractmethod
2
3 class AbstractExample(ABC):
4     def __init__(self, a):
5         self.a = a
6
7     @abstractmethod
8     def abstract(self, a):
9         pass
10
11 ae = AbstractExample()
```

```
$> python3 script.py
```

```
Traceback (most recent call last):
```

```
File "script.py", line 11, in <module>
    ae = AbstractExample()
```

```
TypeError: Can't instantiate abstract class
    AbstractExample with abstract method
    abstract
```

# Summary

- Magic Methods
- Generators
- Advanced loops
- Error handling
- Context managers
- Lambda functions
- Type annotations
- Variables and pointers
- Decorators

