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)

October 17, 2025

Version from October 17, 2025

Overview of Contents

(1) Programming language Python
  (a) Introduction and first steps
  (b) Basics
  (c) Advanced
(2) Markup languages
  (a) 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) Introduction
   (a) Organisation
   (b) Examination
(2) First steps with Python
   (a) Installation
   (b) The first script
   (c) Control flow
   (d) Data types
   (e) Operators

Organisation

- Lecture: Fridays, 10:15 - 11:45, room ESA K
- Seminar: Fridays, 12:15 - 13:45, room Phil A 12006
- Registration in STiNE is required

Organisation

- Lecture: Fridays, 10:15 - 11:45, room ESA K
- Seminar: Fridays, 12:15 - 13:45, room Phil A 12006
- Registration in STiNE is required
- Course material will be provided in Min-Moodle
  - https://lernen.min.uni-hamburg.de/course/view.php?id=5981

Note

You have to log in to Min-Moodle once before we can add you to the course!

Examination

- Written term paper
  - 12-15 pages
  - Deadline 31.03.2026
- Presentation of results as part of the seminar
  - 12 minutes presentation + 3 minutes discussion
- The submitted work must reflect your own effort
  - The use of any auxiliary tools is permitted but must be properly declared

Examination

Note

Prerequisite for registering for the module examination is regular participation in the seminar (defined as being present at least 85% of the sessions).

- Details are given in the examination and study regulations
  - Link to study regulations

Examination

- The task will consist of three parts:
  - Practical part
  - Written part (term paper)
  - Presentation
- Topics will be released during the semester
- We will provide a template for the written term paper that has to be used

NotebookLM

- Interactive environment by Google to interact with documents
  - https://notebooklm.google.com
- Upload course material and generate summaries and explanations
  - E.g., audio overview as an interactive podcast
  - You can jump in anytime and ask questions

NotebookLM

- Interactive environment by Google to interact with documents
  - https://notebooklm.google.com
- Upload course material and generate summaries and explanations
  - E.g., audio overview as an interactive podcast
  - You can jump in anytime and ask questions

Note

NotebookLM can make mistakes. We recommend using it after the lecture to consolidate your understanding.

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«

First Steps with Python – Installation

- Download Python: https://www.python.org/downloads/
- Installed on macOS and many Linux distributions
  - Terminal command: `python3`
  - The command `python` does not exist anymore (or starts the old Python 2)
- Additional ways to install Python later on in this course
  - Virtual environments to install different dependencies and package versions on the same computer

First Steps with Python – Code Editor

- You can use any text editor to write Python code
- A convenient choice is Visual Studio Code (VS Code)
  - https://code.visualstudio.com/
  - Offers syntax highlighting, code completion, debugging, ...
  - Marketplace with many extensions (e.g., Python extension)

First Steps with Python– Information Sources

- Official documentation: https://docs.python.org/3/
- Another documentation: https://devdocs.io/python 3.14/
- Tutorial with examples: https://www.w3schools.com/python/

Programming Language Python

- Interpreted language
  - No compilation
  - Interpreter reads code and executes it
- Interactive shell
  - Start via command `python3`
  - Exit via command `exit()`
- Object-oriented
- Multi-paradigm

- Indentation-based scoping
- 0-based indexing
- Easy to learn
- Quick start
- Many packages for data science
- Open source

The First Script

```
import time

timestamp = time.time()
print(timestamp, "Seconds since 1.1.1970")

timestamp = int(timestamp)
print(timestamp, "Seconds since 1.1.1970")

if timestamp % 60 == 0:
  print("A full minute")
else:
  print("Second {sec}".format(sec=timestamp % 60))
```

The First Script

```
1  import time
2
3  timestamp = time.time()
4  print(timestamp, "Seconds since 1.1.1970")
5
6  timestamp = int(timestamp)
7  print(timestamp, "Seconds since 1.1.1970")
8
9  if timestamp % 60 == 0:
10   print("A full minute")
11 else:
12   print("Second {sec}".format(sec=timestamp % 60))
```

Loading a (standard) library

Output in terminal (separated by ,)

Type conversion

If-else condition

Indentation defines code blocks

The First Script

```python
import time

timestamp = time.time()
print(timestamp, "Seconds since 1.1.1970")

timestamp = int(timestamp)
print(timestamp, "Seconds since 1.1.1970")

if timestamp % 60 == 0:
  print("A full minute")
else:
  print("Second {sec}".format(sec=timestamp % 60))
```

```
$> python3 script.py

1760347988.289336 Seconds since 1.1.1970
1760347988 Seconds since 1.1.1970
Second 8
```

The First Script

```python
import time

timestamp = time.time()
print(timestamp, "Seconds since 1.1.1970")

timestamp = int(timestamp)
print(timestamp, "Seconds since 1.1.1970")

if timestamp % 60 == 0:
  print("A full minute")
else:
  print("Second {sec}".format(sec=timestamp % 60))
```

```
$> python3 script.py

1760347988.289336 Seconds since 1.1.1970
1760347988 Seconds since 1.1.1970
Second 8

$> python3 script.py

1760349060.1601338 Seconds since 1.1.1970
1760349060 Seconds since 1.1.1970
A full minute
```

If-Else

```
duration = 60

if duration < 0:
  print("Time cannot be negative!")
elif duration == 60:
  print("Exactly one minute.")
elif duration > 60:
  print("More than one minute.")
else:
  print("It took " + str(duration) + " seconds.")

message = "That was " + ("fast" if duration < 30 else "slow") + "!"
print(message)
```

If-Else

```
duration = 60

if duration < 0:
  print("Time cannot be negative!")
elif duration == 60:
  print("Exactly one minute.")
elif duration > 60:
  print("More than one minute.")
else:
  print("It took " + str(duration) + " seconds.")

message = "That was " + ("fast" if duration < 30 else "slow") + "!"
print(message)
```

Why do we need `str()`?

If-Else

```
1  duration = 60
2
3  if duration < 0:
4    print("Time cannot be negative!")
5  elif duration == 60:
6    print("Exactly one minute.")
7  elif duration > 60:
8    print("More than one minute.")
9  else:
10   print("It took " + str(duration) + " seconds.")
11
12 message = "That was " + ("fast" if duration < 30 else "slow") + "!"
13 print(message)
```

Why do we need `str()`?

Inline-if: `<true-case> if <condition> else <false-case>`

If-Else

```
1  duration = 60
2
3  if duration < 0:
4    print("Time cannot be negative!")
5  elif duration == 60:
6    print("Exactly one minute.")
7  elif duration > 60:
8    print("More than one minute.")
9  else:
10   print("It took " + str(duration) + " seconds.")
11
12 message = "That was " + ("fast" if duration < 30 else "slow") + "!"
13 print(message)
```

```
$> python3 script.py

Exactly one minute.
That was slow!
```

Loops and Functions

```
1 import sys
2
3 def process_line(s):
4   print(type(s))
5   print(s)
6
7 for line in sys.stdin:
8   process_line(line)
```

Loops and Functions

```
1  import sys
2
3  def process_line(s):
4    print(type(s))
5    print(s)
6
7  for line in sys.stdin:
8    process_line(line)
```

Function definition via `def`

Type of a variable via `type()`

for-loop over standard input `stdin`

Function call

Loops and Functions

```
1 import sys
2
3 def process_line(s):
4   print(type(s))
5   print(s)
6
7 for line in sys.stdin:
8   process_line(line)
```

Fill the standard input stream

```
$> echo "a\nb" | python3 script.py

<class 'str'>
a

<class 'str'>
b
```

Loops in Detail

```
values = [1, 2, 3, 4]
print(values)

for v in values:
  print(v)

values2 = [v*2 for v in values]
print(values2)

while len(values2) > 0:
  print(values2.pop())

print(values2)
```

Loops in Detail

```
1  values = [1, 2, 3, 4]
2  print(values)
3
4  for v in values:
5    print(v)
6
7  values2 = [v*2 for v in values]
8  print(values2)
9
10 while len(values2) > 0:
11   print(values2.pop())
12
13 print(values2)
```

Create a list and output it

Inline-loop (*list comprehension*)

while-loop

`pop()` removes last element from the list

Loops in Detail

```
1  values = [1, 2, 3, 4]
2  print(values)
3
4  for v in values:
5    print(v)
6
7  values2 = [v*2 for v in values]
8  print(values2)
9
10 while len(values2) > 0:
11   print(values2.pop())
12
13 print(values2)
```

```
$> python3 script.py

[1, 2, 3, 4]
1
2
3
4
[2, 4, 6, 8]
8
6
4
2
[]
```

Functions in Detail

```
# print(add_or_multiply(1,2))

def add_or_multiply(x, y, add=True):
  if add:
    return x + y
  else:
    return x * y

print(add_or_multiply(1,2))
print(add_or_multiply(1,2, False))

print(add_or_multiply(x=5, y=6, add=True))
print(add_or_multiply(x=5, add=False, y=6))

add_or_multiply = "Hello"
add_or_multiply(1,2)
```

Functions in Detail

```
1  # print(add_or_multiply(1,2))
2
3  def add_or_multiply(x, y, add=True):
4    if add:
5      return x + y
6    else:
7      return x * y
8
9  print(add_or_multiply(1,2))
10 print(add_or_multiply(1,2, False))
11
12 print(add_or_multiply(x=5, y=6, add=True))
13 print(add_or_multiply(x=5, add=False, y=6))
14
15 add_or_multiply = "Hello"
16 add_or_multiply(1,2)
```

Definition of a function via `<name>(<parameter>):`

Default parameter via `<name>=<default-value>`

Call with values (in the order of parameters)

Call with names and values (in that case, the order of parameters does not matter)

Function names are variables as well

Functions in Detail

```python
# print(add_or_multiply(1,2))

def add_or_multiply(x, y, add=True):
  if add:
    return x + y
  else:
    return x * y

print(add_or_multiply(1,2))
print(add_or_multiply(1,2, False))

print(add_or_multiply(x=5, y=6, add=True))
print(add_or_multiply(x=5, add=False, y=6))

add_or_multiply = "Hello"
add_or_multiply(1,2)
```

```
$> python3 script.py

3
2
11
30
Traceback (most recent call last):
File "script.py", line 16, in <module>
      add_or_multiply(1,2)
TypeError:  'str' object is not callable
```

Functions in Detail

What happens when removing the #?

```
# print(add_or_multiply(1,2))

def add_or_multiply(x, y, add=True):
  if add:
    return x + y
  else:
    return x * y

print(add_or_multiply(1,2))
print(add_or_multiply(1,2, False))

print(add_or_multiply(x=5, y=6, add=True))
print(add_or_multiply(x=5, add=False, y=6))

add_or_multiply = "Hello"
add_or_multiply(1,2)
```

Functions in Detail

```
1  print(add_or_multiply(1,2))
2
3  def add_or_multiply(x, y, add=True):
4    if add:
5      return x + y
6    else:
7      return x * y
8
9  print(add_or_multiply(1,2))
10 print(add_or_multiply(1,2, False))
11
12 print(add_or_multiply(x=5, y=6, add=True))
13 print(add_or_multiply(x=5, add=False, y=6))
14
15 add_or_multiply = "Hello"
16 add_or_multiply(1,2)
```

```
$> python3 script.py

Traceback (most recent call last):
File "script.py", line 1, in <module>
      print(add_or_multiply(1,2))
NameError:  name 'add_or_multiply' is
      not defined
```

Functions in Detail

- It is also possible to define a function within a function
  - The inner function is then only accessible in the outer function

```
1 def multiply(x, y):
2   def inner_multiply(a, b):
3     return a * b
4   return inner_multiply(x, y)
5
6 print(multiply(1,2))
7 print(inner_multiply(3,4))
```

```
$> python3 script.py

2
Traceback (most recent call last):
File "script.py", line 7, in <module>
      print(inner_multiply(3,4))
NameError:  name 'inner_multiply' is not defined
```

Data Types

True, false, and null	`True`, `False`, `None`
Numbers	`12`, `12.5`, `12e3`, `-20`
Strings	`"Hello world!"`, `'Hello world!'`
Tuples	`(1, 2, 3, 4)`, `("A", 2, "C", None)`, `tuple("ABCD")`
Lists	`[1, 2, 3, 4]`, `["A", 2, "C", None]`, `list((1, 2, 3))`
Sets	`{ "a", "b"}`, `{ "a", "a", "b"}`, `set(("a", "b", "b"))`
Dictionaries	`{ "a": 1, "b": 2 }`, `dict((("a", 1 ), ("b", 2)))`, `dict(a=1, b=2)`

Classes (custom types) and further types later in this module

Strings in Detail

```
s = "Hello world "

print(s[0])
print(s[:-2])
print(s[1:3])

print(s.strip() + "!")
print(s.lower())
print(s.replace("world", "everyone"))

print(s == 'Hello world ')

s += "!"
print(s * 2)
print("world" in s)

print(s.split())
print('-'.join(["Hello", "world!"]))

print("Hello {you}, my name is {me}".format(you="A", me="M"))
```

Strings in Detail

```
1  s = "Hello world "
2
3  print(s[0])
4  print(s[:-2])
5  print(s[1:3])
6
7  print(s.strip() + "!")
8  print(s.lower())
9  print(s.replace("world", "everyone"))
10
11 print(s == 'Hello world ')
12
13 s += "!"
14 print(s * 2)
15 print("world" in s)
16
17 print(s.split())
18 print('-'.join(["Hello", "world!"]))
19
20 print("Hello {you}, my name is {me}".format(you="A", me="M"))
```

Access characters and substrings via indexing and slicing

String functions that return a new string

Single and double quotes are interchangeable

Strings in Detail

```python
s = "Hello world "

print(s[0])
print(s[:-2])
print(s[1:3])

print(s.strip() + "!")
print(s.lower())
print(s.replace("world", "everyone"))

print(s == 'Hello world ')

s += "!"
print(s * 2)
print("world" in s)

print(s.split())
print('-'.join(["Hello", "world!"]))

print("Hello {you}, my name is {me}".format(you="A", me="M"))
```

```
$> python3 script.py

H
Hello worl
el
Hello world!
hello world
Hello everyone
True
Hello world !Hello world !
True
['Hello', 'world', '!']
Hello-world!
Hello A, my name is M
```

Tuples in Detail

- Tuples are immutable (they cannot be changed or extended)

```
1  tup1 = (1, 2, 3)
2  tup2 = 5, 6, 7
3
4  print(tup1[0])
5  print(tup2)
6
7  a, b = "A", "B"
8  print(a, b)
9
10 for (i, j) in ((1,"a"), (2, "b"), (3, "c")):
11   print(i, j)
```

Tuples in Detail

- Tuples are immutable (they cannot be changed or extended)

```
1  tup1 = (1, 2, 3)
2  tup2 = 5, 6, 7
3
4  print(tup1[0])
5  print(tup2)
6
7  a, b = "A", "B"
8  print(a, b)
9
10 for (i, j) in ((1,"a"), (2, "b"), (3, "c")):
11   print(i, j)
```

Round parentheses can be omitted

Tuples can be used in assignments

Tuples can be unpacked in loops

Tuples in Detail

- Tuples are immutable (they cannot be changed or extended)

```
1  tup1 = (1, 2, 3)
2  tup2 = 5, 6, 7
3
4  print(tup1[0])
5  print(tup2)
6
7  a, b = "A", "B"
8  print(a, b)
9
10 for (i, j) in ((1,"a"), (2, "b"), (3, "c")):
11   print(i, j)
```

```
$> python3 script.py

1
(5, 6, 7)
A B
1 a
2 b
3 c
```

Lists in Detail

```
1  list1 = list((1, 2, 3))
2  list2 = [5, 6, 7]
3
4  print(list1[:-1])
5  print(list1 + list2)
6
7  list1.append(False)
8  list1.extend(list2)
9  print(list1)
10
11 print(sorted(list1), list1.sort(), list1)
12
13 for i, v in enumerate(list2):
14   print(i, v)
15
16 list3 = [i for i in range(10)]
17 print(list3)
```

Lists in Detail

```
1  list1 = list((1, 2, 3))
2  list2 = [5, 6, 7]
3
4  print(list1[:-1])
5  print(list1 + list2)
6
7  list1.append(False)
8  list1.extend(list2)
9  print(list1)
10
11 print(sorted(list1), list1.sort(), list1)
12
13 for i, v in enumerate(list2):
14   print(i, v)
15
16 list3 = [i for i in range(10)]
17 print(list3)
```

What is the difference between `append()` and `extend()`?

What is the difference between `sorted(l)` and `l.sort()`?

Lists in Detail

```
1  list1 = list((1, 2, 3))
2  list2 = [5, 6, 7]
3
4  print(list1[:-1])
5  print(list1 + list2)
6
7  list1.append(False)
8  list1.extend(list2)
9  print(list1)
10
11 print(sorted(list1), list1.sort(), list1)
12
13 for i, v in enumerate(list2):
14   print(i, v)
15
16 list3 = [i for i in range(10)]
17 print(list3)
```

```
$> python3 script.py

[1, 2]
[1, 2, 3, 5, 6, 7]
[1, 2, 3, False, 5, 6, 7]
[False, 1, 2, 3, 5, 6, 7] None [False,
      1, 2, 3, 5, 6, 7]
0 5
1 6
2 7
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dictionaries in Detail

```
dic = {"a" : 1, "b" : 2}

print(dic["a"])
dic["c"] = 3
print(dic)

del dic["b"]
print("b" in dic, "b" not in dic)

for k in dic: # dic.keys()
  print(k)

for v in dic.values():
  print(v)

for k, v in dic.items():
  print(k, v)
```

Dictionaries in Detail

```
1  dic = {"a" : 1, "b" : 2}
2
3  print(dic["a"])
4  dic["c"] = 3
5  print(dic)
6
7  del dic["b"]
8  print("b" in dic, "b" not in dic)
9
10 for k in dic: # dic.keys()
11   print(k)
12
13 for v in dic.values():
14   print(v)
15
16 for k, v in dic.items():
17   print(k, v)
```

Access value at key via `<dict-name>[<key>]`

Iteration and check whether a key exists

Dictionaries in Detail

```
1  dic = {"a" : 1, "b" : 2}
2
3  print(dic["a"])
4  dic["c"] = 3
5  print(dic)
6
7  del dic["b"]
8  print("b" in dic, "b" not in dic)
9
10 for k in dic: # dic.keys()
11   print(k)
12
13 for v in dic.values():
14   print(v)
15
16 for k, v in dic.items():
17   print(k, v)
```

```
$> python3 script.py

1
{'a': 1, 'b': 2, 'c': 3}
False True
a
c
1
3
a 1
c 3
```

Operators

Comparison	`==`	Equality
	`<` and `>`	Less and greater than
	`<=` and `>=`	Less and greater than or equal
Logical	`not`	Negation
	`and`	Conjunction
	`or`	Disjunction
Math	`*` and `**`	Multiplication and exponentiation
	`/` and `//`	Division and integer division
	`+` and `-`	Addition and subtraction
	`%`	Modulo (division with remainder)

`in`	Test for membership
`=`	Assignment
`+=`	Addition and assignment
`-=`	Subtraction and assignment
`*=`	Multiplication and assignment
`/=`	Division and assignment

Operators

Comparison	`==`	Equality
	`<` and `>`	Less and greater than
	`<=` and `>=`	Less and greater than or equal
Logical	`not`	Negation
	`and`	Conjunction
	`or`	Disjunction
Math	`*` and `**`	Multiplication and exponentiation
	`/` and `//`	Division and integer division
	`+` and `-`	Addition and subtraction
	`%`	Modulo (division with remainder)

`in`	Test for membership
`=`	Assignment
`+=`	Addition and assignment
`-=`	Subtraction and assignment
`*=`	Multiplication and assignment
`/=`	Division and assignment

Note

This is just a small selection of operators!

Operators

- Operators are defined for the respective data type
- Classes can define operators differently
  - For example, the + operator applied to strings concatenates two strings

Class-Specific Operators – Example

```
1 s1 = {1, 2, 3}
2 s2 = {2, 3, 4}
3
4 print(s1 - s2, s1.difference(s2))
5 print(s1 & s2, s1.intersection(s2))
6 print(s1 | s2, s1.union(s2))
7
8 s1 |= {4}
9 print(s1)
```

```
$> python3 script.py

{1} {1}
{2, 3} {2, 3}
{1, 2, 3, 4} {1, 2, 3, 4}
{1, 2, 3, 4}
```

Useful Functions

Strings	`s.strip()`	Removes any leading and trailing whitespaces in the string `s`
	`s.lower()`	Changes all characters in a string `s` to lower case
	`s.replace(x, y)`	Replaces all occurrences of `x` by `y` in the string `s`
	`s.split(x)`	Splits a string `s` at each occurrence of `x` into a list
	`s.join(x)`	Joins all elements in a list `x` using the separator `s` into a string
Lists	`l.append(x)`	Adds the element `x` to the end of the list `l`
Dictionaries	`d.items()`	Iterates over all elements in the dictionary `d` as `(key, value)` tuples
	`d.values()`	Iterates over all values in the dictionary `d`
Iteration	`enumerate(l)`	Enumerates all elements in a list `l` as `(index, element)` tuples
	`zip(l1, l2)`	Combines elements from lists `l1` and `l2` into pairs for each index
	`range(x)`	Generates a sequence of integers from `0` to `x - 1`
Types	`str(x)`	Converts `x` into a string
	`int(x)`	Converts `x` into an integer (rounds down)
	`float(x)`	Converts `x` into a floating point number
	`type(x)`	Returns the data type of `x`

Useful Functions

General	`print(x)`	Outputs `x`
	`eval(s)`	Evaluates the string `s` as a Python expression
	`open(f, r)`	Opens the file `f` in mode `r` (with `'r'` for reading and `'w'` for writing)
	`len(x)`	Returns the length (number of elements) of `x`

Useful Functions

General	`print(x)`	Outputs `x`
	`eval(s)`	Evaluates the string `s` as a Python expression
	`open(f, r)`	Opens the file `f` in mode `r` (with `'r'` for reading and `'w'` for writing)
	`len(x)`	Returns the length (number of elements) of `x`

»Eval is Evil!«

- `eval(s)` executes arbitrary code
- The input `s` must be fully trusted
- Otherwise, `eval(s)` is a major security risk

Example

```
def extract_numbers(l):
  l = l.strip()
  numbers = []
  for p in l.split(","):
    if p.strip().isnumeric():
      numbers.append(int(p))
  return numbers

def build_csv(nl):
  csv = ""
  for line in nl:
    csv += ','.join([
      str(n) for n in line
    ]) + "\n"
  return csv
```

```
f = open("01-Python-Introduction-14.csv", "r")
lines = f.readlines()
f.close()

new_lines = []
for line in lines:
  numbers = extract_numbers(line)
  new_lines.append([n ** 2 for n in numbers])

print(build_csv(new_lines))
```

Example

```python
def extract_numbers(l):
  l = l.strip()
  numbers = []
  for p in l.split(","):
    if p.strip().isnumeric():
      numbers.append(int(p))
  return numbers

def build_csv(nl):
  csv = ""
  for line in nl:
    csv += ','.join([
      str(n) for n in line
    ]) + "\n"
  return csv
```

```python
f = open("01-Python-Introduction-14.csv", "r")
lines = f.readlines()
f.close()

new_lines = []
for line in lines:
  numbers = extract_numbers(line)
  new_lines.append([n ** 2 for n in numbers])

print(build_csv(new_lines))
```

```
A, Carol, 12, 2045
B, Dave, 13, 5689
C, Alice, 89, 38594
D, Bob, 09, 2830
```

01-Python-Introduction-14.csv

```
$> python3 script.py

144,4182025
169,32364721
7921,1489496836
81,8008900
```

Summary

- Installation and first steps with Python
- Control flow (conditions, loops, and functions)
- Data types
- Operators
- Useful built-in functions

The slides cover only a small selection of available features!