

Werkzeuge für das wissenschaftliche Arbeiten

Python for Machine Learning and Data Science

Magnus Bender
bender@ifis.uni-luebeck.de
Wintersemester 2022/23

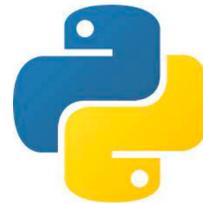
Inhaltsübersicht

1. Programmiersprache Python

a) *Einführung, Erste Schritte*

b) *Grundlagen*

c) **Fortgeschritten**



2. Auszeichnungssprachen

a) LaTeX, Markdown

L^AT_EX



3. Benutzeroberflächen und Entwicklungsumgebungen

a) Jupyter Notebooks lokal und in der Cloud (Google Colab)

4. Versionsverwaltung

a) Git, GitHub



5. Wissenschaftliches Rechnen

a) NumPy, SciPy



6. Datenverarbeitung und -visualisierung

a) Pandas, matplotlib, NLTK

7. Machine Learning (scikit-learn)

a) Grundlegende Ansätze (Datensätze, Auswertung)

b) Einfache Verfahren (Clustering, ...)



8. DeepLearning

a) TensorFlow, PyTorch, HuggingFace Transformers



Themen

- Magic Methods
- Generatoren
- Erweiterte Schleifen
- Fehlerbehandlung
- Kontextmanager
- Lambda-Funktionen
- Typannotationen
- Variablen und Zeiger
- Dekoratoren



Heute

```
class Vector():
```

```
def __init__(self, a, b):  
    self.a = a  
    self.b = b
```

```
def __add__(self, o):  
    return Vector(self.a + o.a, self.b + o.b)
```

```
def __sub__(self, o):  
    return Vector(self.a - o.a, self.b - o.b)
```

```
def __iadd__(self, o):  
    self.a += o.a  
    self.b += o.b  
    return self
```

```
# def __isub__(self, o):
```

```
def __str__(self):  
    return "({}, {})".format(self.a, self.b)
```

Magic Method

Falls kein String, versucht print() einen mit str() zu erzeugen und bei einer Klasse geht dies per __str__().

```
x = Vector(1, 2)  
print(x, str(x), x.__str__())
```

(1, 2) (1, 2) (1, 2)

```
y = Vector(2, 3)  
print(y)
```

(2, 3)

```
print(x + y, x.__add__(y))
```

(3, 5) (3, 5)

```
print(x - y)
```

(-1, -1)

```
x += y  
print(x)
```

(3, 5)

```
x -= y  
print(x)
```

(1, 2)

Warum geht es auch, obwohl
__isub__ nicht definiert ist?

Magic Methods

Mathe- matisch	<code>__add__</code>	<code>+</code>	Addition
	<code>__sub__</code>	<code>-</code>	Subtraktion
	<code>__mul__</code>	<code>*</code>	Multiplikation
	<code>__matmul__</code>	<code>@</code>	Matrixmultiplikation
	<code>__truediv__</code>	<code>/</code>	Division
	<code>__floordiv__</code>	<code>//</code>	Ganzzahldivision
	<code>__pow__</code>	<code>**</code>	Potenz
Logisch	<code>__and__</code>	<code>&</code>	Bitweises Und
	<code>__xor__</code>	<code>^</code>	Bitweises Entweder-Oder
	<code>__or__</code>	<code> </code>	Bitweises Oder

Vergleichend	<code>__lt__</code>	<code><</code>	Kleiner
	<code>__le__</code>	<code><=</code>	Kleiner gleich
	<code>__eq__</code>	<code>==</code>	Gleich
	<code>__ne__</code>	<code>!=</code>	Ungleich
	<code>__gt__</code>	<code>></code>	Größer
	<code>__ge__</code>	<code>>=</code>	Größer gleich
Klassen (-eigenschaften)	<code>__bool__</code>	Bool eines Objekts (if obj:)	
	<code>__hash__</code>	Eindeutiger Hash des Objekts	
	<code>__len__</code>	Länge des Objekts	
	<code>__str__</code>	String (Ausgabe)	
	<code>__del__</code>	Objekt löschen (del obj)	

Magic Methods: dict & list

```
class DictContainer():
```

```
    def __init__(self):  
        self.dict = {}
```

```
    def __setitem__(self, key, value):  
        self.dict[key] = value
```

```
    def __getitem__(self, key):  
        return self.dict[key]
```

```
    def __delitem__(self, key):  
        del self.dict[key]
```

```
    def __contains__(self, key):  
        return key in self.dict
```

```
    def __str__(self):  
        return str(self.dict)
```

```
dc = DictContainer()
```

```
dc["a"] = "A"  
dc.__setitem__("b", "B")  
print(dc)
```

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

```
print(dc["a"])  
print(dc.__getitem__("b"))
```

```
A
```

```
B
```

```
del dc["a"]  
print(dc)
```

```
{'b': 'B'}
```

```
print("a" in dc, "b" in dc)
```

```
False True
```

Generatoren

Generator Expression ()
List Comprehension []

```
import timeit
```

```
lc_a = """  
[i for i in range(2**10)]  
"""
```

```
print(timeit.timeit(lc_a, number=20))
```

0.0007693090010434389

```
lc_b = """  
[i for i in range(2**20)]  
"""
```

```
print(timeit.timeit(lc_b, number=20))
```

0.9470873650006979

```
import timeit
```

```
g_a = """  
(i for i in range(2**10))  
"""
```

```
print(timeit.timeit(g_a, number=20))
```

1.097899985325057e-05

```
g_b = """  
(i for i in range(2**20))  
"""
```

```
print(timeit.timeit(g_b, number=20))
```

2.2750000425730832e-05

Links hängt die Dauer von der Anzahl ab, rechts nicht. Warum?

Eine ganze Liste mit den einzelnen Zahlen wird erstellt und danach zurückgegeben.

Generatoren: yield

```
def list_range(until):  
    l = []  
    i = 0  
    while i < until:  
        l.append(i)  
        i += 1  
    return l
```

```
print(list_range(20))  
for i in list_range(5):  
    print(i)
```

```
[0, 1, 2, 3, 4, 5, ... 18, 19]
```

```
0  
1  
2  
3  
4
```

```
def generate_range(until):  
    i = 0  
    while i < until:  
        yield i  
        i += 1
```

```
print(generate_range(20))  
for i in generate_range(5):  
    print(i)
```

```
<generator object ... at 0x10a3bc040>
```

```
0  
1  
2  
3  
4
```

Es wird immer nur der nächste Wert erzeugt, dieser zurückgegeben, verarbeitet und dann der nächste erzeugt.

Iteration mit Generatoren

```
class FileReader():
```

```
def __init__(self, filename):  
    self.filename = filename
```

```
def __iter__(self):  
    t = open(self.filename + ".titles.txt", "r")  
    c = open(self.filename + ".contents.txt", "r")
```

```
    t_l = t.readline()  
    c_l = c.readline()
```

```
    while t_l and c_l:  
        yield t_l.strip(), c_l.strip().split(",")
```

```
        t_l = t.readline()  
        c_l = c.readline()
```

```
    t.close()  
    c.close()
```

Iteration über ein Objekt durch Nutzung von yield.

Auch bei großen Dateien bleibt der Speicherverbrauch konstant. Laufzeit erhöht sich aber leicht.

Implizierter Aufruf der *Magic Method* `__iter__()`.

e.contents.txt

```
Audi,BMW,Ford  
Apple,Banana,Pear  
Mouse,Screen,Keyboard
```

e.titles.txt

```
Car  
Fruit  
Computer
```

```
fr = FileReader("e")  
for t, c in fr:  
    print(t, c)
```

```
Car ['Audi', 'BMW', 'Ford']  
Fruit ['Apple', 'Banana', 'Pear']  
Computer ['Mouse', 'Screen', 'Keyboard']
```

Erweiterte Schleifen

- List Comprehensions für Wörterbücher

```
{i : i**2 for i in range(5)}  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

- „Break“ und „Continue“

- „Else“ nach Schleifen

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

Not found!

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

Found!

Else unter einer Schleife wird also genau dann ausgeführt, wenn die Schleife durchgelaufen ist und nicht abgebrochen wurde.

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

2
3
4
5

Fehlerbehandlung

```
def divide(x, y):  
    try:  
        r = x / y  
    except ZeroDivisionError:  
        print("Division by zero!")  
    except TypeError as e:  
        print("TypeError:", e)  
    else:  
        print("{x} / {y} = {r}".format(  
            x=x, y=y, r=r  
        ))  
    finally:  
        print("Finally done ;)")
```

divide(1, 2)

```
1 / 2 = 0.5  
Finally done ;)
```

divide(1, 0)

```
Division by zero!  
Finally done ;)
```

divide("A", 2)

```
TypeError: unsupported operand type(s) for  
Finally done ;)
```

Verschiedene Fehler werden unterschieden und könne als Variable abgegriffen werden.

Falls kein Fehler auftrat.

Immer, egal ob Fehler oder nicht.

Wozu könnte finally nützlich sein?

Fehlerbehandlung

- Fehler können mit `raise` ausgelöst werden
- Fehler sind Objekte der Klasse (einer Unterklasse von) `Exception`
- Vordefinierte Fehler: <https://docs.python.org/3/library/exceptions.html#builtin-exceptions>
- Eigene Fehler können als Unterklasse erstellt werden

Wofür könnte eine solcher Fehlerklasse ohne weitere Implementierung nützlich sein?

```
class MyError(Exception):  
    pass
```

```
raise MyError("Stopp")
```

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

Wo ist hier der Fehler?

Kontextmanager

```
try:
```

```
t = open("example.titles.txt", "r")  
t.write("huhu")  
t.close()
```

```
except BaseException as e:  
    print(e)
```

```
print(t.readline())
```

```
not writable
```

```
Car
```

Wenn `t.write()` einen Fehler auslöst, dann wird `t.close()` nicht ausgeführt und die Datei bleibt offen!

```
try:
```

```
with open("example.contents.txt", "r") as c:  
    c.write("huhu")
```

```
except BaseException as e:  
    print(e)
```

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

not writable

Intern durch die Magic Methods `__enter__()` und `__exit__()` realisiert.

```
Traceback (most recent call last):  
  File "name.py", line 18, in <module>  
    print(c.readline())  
ValueError: I/O operation on closed file.
```

```
def pow2(number):  
    return number ** 2
```

Lambda-Funktionen

```
numbers = [1, 2, 3, 4]  
for i, n in enumerate(numbers):  
    numbers[i] = pow2(n)  
print(numbers)           [1, 4, 9, 16]
```

Map nutzt intern einen Iterator und erzeugt somit nicht direkt eine Liste, sondern jeweils das nächste Element während der Iteration.

```
numbers = map(pow2, numbers)  
print(numbers, list(numbers))
```

```
<map object at 0x10e0a2830> [1, 4, 9, 16]
```

```
numbers = [1, 2, 3, 4]  
numbers = map(lambda n : n**2, numbers)  
print(numbers, list(numbers))
```

```
<map object at 0x10e0a3280> [1, 4, 9, 16]
```

```
def pow2_f(number):  
    return number ** 2  
  
pow2_l = lambda n : n**2
```

Zwei verschiedene Möglichkeiten eine Funktion zu erstellen!

Sehr praktisch für z.B.
`sort(key=lambda o: o.name)`

Map, Filter & Reduce

Alternative Schreibweisen mit den gleichen Ergebnis

```
numbers = [1, 2, 3, 4]
```

```
m = map(lambda n: n ** 3 if n < 3 else n ** 2, numbers)
print(list(m))
```

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

```
[n ** 3 if n < 3 else n ** 2 for n in numbers]
```

```
f = filter(lambda n: n % 2 == 0, numbers)
print(list(f))
```

```
[2, 4]
```

```
[n for n in numbers if n % 2 == 0]
```

```
from functools import reduce
```

```
r = reduce(lambda c, n: c + n, numbers, 0)
print(r)
```

```
10
```

Teilweise auch fold() genannt.

```
c = 0
for n in numbers:
    c = c + n
# c = (lambda c, n: c + n)(c, n)
```

Intern wir quasi das Folgenden ausgeführt.

Typannotationen

```
def str_repeat(s:str, n:int, sep:str=None) -> str:  
    s = s + sep if sep else s  
    return s * n
```

```
print(str_repeat("Hallo", 5))  
print(str_repeat("Hallo", 2, ", "))  
print(str_repeat(5, 5))
```

```
HalloHalloHalloHalloHallo  
Hallo, Hallo,  
25
```

Was passiert jeweils?

```
from pydantic import validate_arguments
```

```
@validate_arguments  
def str_repeat(s:str, n:int, sep:str=None):
```

Mit dem externen Paket pydantic ist tatsächlich eine Prüfung der Typen möglich.

Englisch „Type Hints“ und mehr als Hinweise sind es auch nicht, Python prüft Parameter nicht und führt die Funktion munter aus!

Typannotationen

```
from typing import NoReturn, Union, Optional

def hello(name:Optional[str]="") -> NoReturn:
    print("Hello " + name)

def divide(x:Union[int, float], y:int|float) -> float:
    return x / y
```

Hier könnte man auch jeweils eine eigene Klasse als Typ nutzen.

```
from typing import List, Dict, Tuple
```

```
Tuple[int, str] # (1, "a")
```

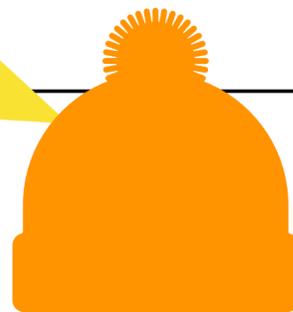
```
Tuple[int, ...] # (1, 1), (1, 2, 3)
```

```
List[str] # ["a", "b"]
```

```
List[Union[str, int]] # [1, "b"], ["a", 2]
```

```
Dict[str, str] # {"a" : "A", "b" : "B"}
```

„Type Hints“ sind sehr praktisch für die Dokumentation des Codes!



Funktionsparameter: ** und *

```
def example(*args, **kwargs):  
    print(type(args), args)  
    print(type(kwargs), kwargs)
```

```
example("A", "B", "C", d="D", e="E")
```

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

```
p = "A", "B", "C"  
kp = {"d": "D", "e": "E"}  
example(*p, **kp)
```

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

Sehr praktisch um Parameter
an eine Oberklasse
weiterzureichen.



Variablen und Zeiger (Pointer)

- Python nutzt *Call-by-Reference*

Call-by-Value: Funktionsaufrufe übertragen den Wert
Call-by-Reference: Funktionsaufrufe übertragen einen Zeiger auf ein Objekt/ den Wert

- Viele Typen sind jedoch unveränderlich (z.B. `str`, `int`, `float`, `tuple`)
- Bei „Änderungen“ eines unveränderlichen Typen wird somit immer ein neues Objekt erzeugt

```
s = "Hallo"  
print(id(s))    4517249328  
s += "Welt"  
print(id(s))    4517249200
```

```
l = ["Hallo"]  
print(id(l))    4516668480  
l.append("Welt")  
print(id(l))    4516668480
```

Variablen und Zeiger

```
def change(l):  
    for i in range(len(l)//2):  
        l[i], l[-(i+1)] = l[-(i+1)], l[i]
```

```
l = [1, 2, 3, 4, 5]  
change(l)  
print(l)
```

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

Tatsächlich kann man auch
änderbare Typen in einem
unveränderbaren Typen
verändern:

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

Liste wird geändert, ohne
zurückgegeben zu
werden, denn ein Zeiger
wurde übergeben!

Warum ändern sich alle?!

```
bools = [True] * 5  
print(bools)  
bools[2] = False  
bools[3] = False  
print(bools)
```

```
[True, True, True, True, True]
```

```
[True, True, False, False, True]
```

```
lists = [[]] * 5  
print(lists)  
lists[2].append(2)  
lists[3].append(3)  
print(lists)
```

```
[[], [], [], [], []]
```

```
[[2, 3], [2, 3], [2, 3], [2, 3], [2, 3]]
```

Dekoratoren

```
def my_decorator(some_function):  
    print("Funktion laden")
```

```
def my_wrapper(*args, **kwargs):  
    print("Vor Ausführung der Funktion")  
    v = some_function(*args, **kwargs)  
    print("Nach Ausführung der Funktion")  
    return v
```

```
return my_wrapper
```

```
@my_decorator  
def example(name):  
    print("Hallo " + name)
```

```
example("Magnus")  
example("Otto")
```

Drei Einstiegspunkte, beim Laden der Funktion durch den Interpreter sowie nach und vor jeder Ausführung.

Die Dekoratoren können auch selbst Parameter haben.

Achtung: Beim Erstellen von Dekoratoren gibt es noch ein paar Fallstricke!

Funktion laden

Vor Ausführung der Funktion
Hallo Magnus
Nach Ausführung der Funktion

Vor Ausführung der Funktion
Hallo Otto
Nach Ausführung der Funktion

Anwendung Dekoratoren: ABC

- Gibt es abstrakte Klassen in Python?
- Eigentlich nicht
 - Aber man kann abstrakte Klassen mit Dekoratoren erstellen
 - Bzw. das Paket abc nutzen

```
from abc import ABC, abstractmethod
```

```
class AbstractExample(ABC):
```

```
    def __init__(self, a):  
        self.a = a
```

```
    @abstractmethod  
    def abstract(self, a):  
        pass
```

```
ae = AbstractExample()
```

```
Traceback (most recent call last):  
  File "name.py", line 13, in <module>  
    ae = AbstractExample()  
TypeError: Can't instantiate abstract  
class AbstractExample with abstract  
method abstract
```

Zusammenfassung

- Magic Methods
- Generatoren
- Erweiterte Schleifen
- Fehlerbehandlung
- Kontextmanager
- Lambda-Funktionen
- Typannotationen
- Variablen und Zeiger
- Dekoratoren



Nächste Woche findet ein Übungstermin im PC Pool zu den Projektaufgaben 1 & 2 statt.



~~Heute~~

Inhaltsübersicht

1. Programmiersprache Python
 - a) *Einführung, Erste Schritte*
 - b) *Grundlagen*
 - c) *Fortgeschritten*
2. Auszeichnungssprachen
 - a) **LaTeX, Markdown**
3. Benutzeroberflächen und Entwicklungsumgebungen
 - a) Jupyter Notebooks lokal und in der Cloud (Google Colab)
4. Versionsverwaltung
 - a) Git, GitHub
5. Wissenschaftliches Rechnen
 - a) NumPy, SciPy
6. Datenverarbeitung und -visualisierung
 - a) Pandas, matplotlib, NLTK
7. Machine Learning (scikit-learn)
 - a) Grundlegende Ansätze (Datensätze, Auswertung)
 - b) Einfache Verfahren (Clustering, ...)
8. DeepLearning
 - a) TensorFlow, PyTorch, HuggingFace Transformers