



Guido van Rossum - Autor der  
Programmiersprache Python

Von Doc Searls - 2006oscon\_203.JPG, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=4974869>

# Grundlagen des Maschinellen Lernens

## Python Einführung

21.04.2021  
Prof. Dr. Volker Gruhn  
Nils Schwenzfeier

1. Einleitung
2. Datentypen
3. Kontrollfluss
4. Funktionen
5. Exceptions
6. Objektorientierte Programmierung (OOP)
7. NumPy



Lehrstuhl für  
Software Engineering

UNIVERSITÄT  
DUISBURG  
ESSEN

*Offen im Denken*

# Einleitung

- Dynamische Programmiersprache
- Erlaubt verschiedene Programmierparadigmen
  - Objektorientiert
  - Funktional
  - Prozedural
- Code wird im Python-Interpreter ausgeführt
  - Code ist Plattformunabhängig

- Einfache, leicht zu lesende und lernende Syntax
- Intuitive Objektorientierte Programmierung
- Sehr verbreitete Sprache
  - Große Community
  - Viele bestehende Libraries
- Standard Library umfasst bereits viele Anwendungsbereiche
- Leicht zu erweitern um schnelleren C/C++ Code

- Machine Learning Algorithmen sind rechenintensiv
- Interpretersprachen (z.B. Java, Python) sind langsamer als kompilierte Sprachen
- Große Teile der Python-Bibliotheken sind in C/C++ geschrieben und daher schnell
- Rechenintensive Algorithmen können in C/C++ geschrieben werden, falls notwendig

# Datentypen

- Starke Typisierung
  - Eine Variable hat immer genau einen Typ
  - Bei starker Typisierung ist eine implizite Konvertierung von einem Typ in einen anderen Typ nicht erlaubt
- Dynamische Typisierung
  - Variablen haben keine Deklaration
  - Dieselbe Variable kann unterschiedliche Typen im Programm annehmen
  - Der Typ einer Variable wird zur Laufzeit ermittelt, wenn die Variable genutzt wird

```
number = 42
print(number, type(number))
print(number + 42)
```

```
number = "42"
print(number, type(number))
print(number + 42)
```

```
42 <class 'int'>
84
42 <class 'str'>
Traceback (most recent call last):
  File "starke_typisierung.py", line 7, in <module>
    print(number + 42)
TypeError: can only concatenate str (not "int") to str
```



*„When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck“*

— James Whitcomb Riley

- Typen werden durch Klassen realisiert
  - Python hat einige Klassen standardmäßig
- Text: `str`
- Numerisch: `int`, `float`, `complex`
- Sequenz: `list`, `tuple`, `range`
- Zuordnung: `dict`
- Set: `set`, `frozenset`
- Boolean: `bool`

- Datentyp
  - int: Ganze Zahlen (unendlich)
  - float: Gleitkommazahlen
  - complex: Komplexe Zahlen (j ist der imaginäre Teil)
- Operationen (weitere Operationen im math-Modul)
  - Standard: +, -, \*, /
  - Modulo: %, divmod(a, b)
  - Exponent: a\*\*b, pow(a, b)
  - Runden: round(a)
  - Betrag: abs(a)

```
x = 1      # int
x = 1.5    # float
x = 1 + 5j # complex

# Alternative Schreibweise für float
x = 1e1
# e steht für eine Potenz von 10
# x == 1 * 10^1 == 10
print(x, type(x))

x = 1e-1
# x == 1 * 10^-1 == 0.1
print(x, type(x))

10.0 <class 'float'>
0.1 <class 'float'>
```



- Datentyp: str
- Strings werden durch einfache oder doppelte Anführungszeichen umgeben
- Strings umgeben von drei Anführungszeichen können mehrere Zeilen umfassen
- Strings mit Prefix r ignorieren Escape-Sequenzen
- Strings mit Prefix f können dynamische Werte in geschweiften Klammern enthalten
  - Ab Version 3.6
  - Für ältere Versionen: str.format()

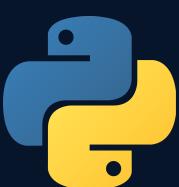
```
s = "Hello World!"  
s = 'Hello World!'  
print(s)  
  
s = """Dieser String  
geht über  
mehrere Zeilen"""  
print(s)  
  
s = r"Hello \nWorld"  
print(s)  
  
a = 42  
s = f"Die Antwort lautet {a}"  
print(s)  
  
Hello World!  
Dieser String  
geht über  
mehrere Zeilen  
Hello \nWorld  
Die Antwort lautet 42
```



- Datentyp: list
- Speichert mehrere Werte in einer Variable
  - Sind Pythons Arrays
  - Jeder Wert kann von einem anderen Typ sein
- Element hinzufügen: `list.append(x)`
- Element löschen: `list.remove(x)`
- Element löschen und zurückgeben an Position:  
`list.pop(i)`

```
# empty list
s = []

# list with different types
s = [1, "2", 3, "4"]
```

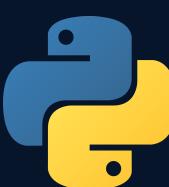


- Datentyp: tuple
- Konstante Liste, kann nach der Erstellung nicht mehr geändert werden
- Listen werden durch [] gekennzeichnet, Tuple werden durch () gekennzeichnet
- Tuple und Listen können in einander geschachtelt werden

```
# simple tuple
s = (1, 2)
# () are optional
s = 1, 2

# tuple can have elements with different types
s = (1, "2", 3, "4")

# nested, multidimensional list
s = [[1, 2, 3], (4, 5, 6), "42"]
```



- Strings, Listen und Tuple haben viele Gemeinsamkeiten; Sie sind sogenannten Sequenzen
- Strings und Tuple sind konstante Sequenzen, die nicht mehr geändert werden können
- String-Funktionen geben immer einen neuen String zurück

- Zugriff auf ein Element an Position i:  $s[i]$
- Zugriff auf ein Element an Position i vom Ende aus gesehen:  $s[-i]$
- Letztes Element ist daher immer:  $s[-1]$
- Teilsequenz einer Sequenz:  $s[i : j]$ , mit Schrittgröße k:  $s[i : j : k]$ 
  - Teilsequenz startet an Position i und endet an Position j
  - i und j können weggelassen werden und es wird der Anfang, bzw. das Ende der Sequenz ergänzt
- Länge der Sequenz:  $\text{len}(s)$
- Sequenz enthält Element x:  $x \text{ in } s$ ,  $x \text{ not in } s$
- Sequenz konkatenieren:  $s + t$
- Sequenz multiplizieren:  $s * n$ 
  - n ist ein Integer
  - Äquivalent mit n-facher Konkatenation mit sich selbst
- Kleinstes/Größtes Element:  $\text{min}(s)$ ,  $\text{max}(s)$

- Datentyp: **set, frozenset** (Konstantes Set)
  - **s = set([sequence])**
  - **s = frozenset([sequence])**
- Sets sind Mengen, d.h. sie sind ungeordnet und enthalten keine Duplikate
  - Füge Element hinzu: **s.add(x)**
  - Entferne Element: **s.remove(x)**
  - Länge des Sets: **len(s)**
  - Set enthält Element: **x in s**
- Mengenfunktionen sind für Sets vorhanden
  - Ist Teilmenge: **s.issubset(t)**,  $s \leq t$ , Echte Teilmenge: **s < t**
  - Ist Obermenge: **s.issuperset(t)**,  $s \geq t$ ,  $s > t$
  - Vereinigung: **s.union(t)**,  $s \mid t$
  - Schnittmenge: **s.intersection(t)**,  $s \& t$
  - Differenz: **s.difference(t)**,  $s - t$

- Datentyp: dict
- Dictionary bestehen aus Schlüssel-Wert-Paaren
  - Jedem Schlüssel wird ein Wert zugeordnet
- Schlüssel sind ungeordnet
- Sichten auf Dictionaries
  - Liste aller Schlüssel: `dictionary.keys()`
  - Liste aller Werte: `dictionary.values()`
  - Liste aller (Schlüssel, Wert)-Tuples:  
`dictionary.items()`

```
# simple dictionary
# key, value mapping via :
dictionary = { "key1": "value1", "key2": 2}

# add a new key
dictionary["new_key"] = 42

# access value with key
dictionary["key1"]
```



- Datentyp: bool
- Kann nur die Werte **True**, **False** annehmen
- Bestimmte Werte werden immer als **False** evaluiert
  - **None** (Datentyp `NoneType`)
  - **False**
  - **0** (Für jeden numerischen Datentyp)
  - Leere Sequenzen: „“, `[]`, `()`
  - Leeres Dictionary: `{}`
  - Leeres Set: `set()`
- Alle anderen Werte für Standard Datentypen werden zu **True** evaluiert

- Ein Typ kann zu einem anderen Typ konvertiert werden
  - Nicht jeder Typ kann in jeden anderen Typ konvertiert werden
  - Manche Konvertierungen sind nur für bestimmte Werte möglich
    - String -> Numerisch ist nur möglich wenn der String eine Zahl ist
- Konvertierung erfolgt durch den Ziel-Datentyp gefolgt von dem Objekt in runden Klammern
  - `int(„42“)` -> 42
  - `str(42)` -> „42“
  - `float(42)` -> 42.0

# Kontrollfluss

- In Python werden Blöcke durch Einrückung erzeugt
- In Java werden Blöcke durch geschweifte Klammern erzeugt
- Standard für die Einrückung: 4 Leerzeichen
- Für gewöhnlich wird einfach Tab genutzt
- Python benötigt keine Klammern um Kontroll-Anweisungen
- Python kennt keine switch-Anweisung

```
if a == 1:  
    print("a is 1")  
elif a == 2:  
    print("a is 2")  
elif a == 3:  
    print("a is 3")  
else:  
    print("a is not 1, 2, 3")
```



- Vergleich von Inhalt: `==`, `<`, `>`, `<=`, `>=`, `!=`
- Vergleich von Identität: `a is b`, `a is not b`
- Logische Verknüpfungen: `a and b`, `a or b`
- Negation: `not a`

- In Python beziehen sich For-Schleifen nicht auf eine hochzählende Variable
- For-Schleifen beziehen sich immer auf die Elemente in einem Iterator
  - Sequenzen sind immer Iteratoren
- Die Funktion `range(start, end, [step])` kann genutzt werden um einen Iterator von start bis end mit Schrittgröße step zu erzeugen
- Schleifen frühzeitig abbrechen: `break`
- Nächste Iteration: `continue`

```
seq = ["Hello", "World", "!"]

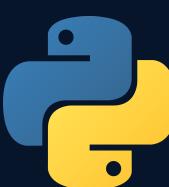
# range function creates a sequence to iterate over
for i in range(10):
    print(i) # 0, 1, 2, 3, ...

# the range function can be used to emulate java behaviour
for i in range(len(seq)):
    print(seq[i]) # "Hello", "World", "!"

# we can iterate directly over the sequence
for value in seq:
    print(value) # "Hello", "World", "!"

# if the index is necessary use enumerate
for i, value in enumerate(seq):
    print(i, value) # 1 "Hello", 2 "World", 3 "!"

# reversing order
for value in reversed(seq):
    print(value) # "!", "World", "Hello"
```



- While-Schleife wird solange ausgeführt bis die Bedingung **False** wird
- **break** und **continue** können auch auf While-Schleifen angewandt werden
- Do-while Schleifen gibt es in Python nicht, können aber emuliert werden

```
# simple while loop
i = 0
while i < 3:
    i += 1

# emulate do-while
i = 0
while True:
    # do something
    i += 1
    # stop loop if condition
    if i >= 3:
        break
```



# List Comprehension

► Kontrollfluss



- Python bietet eine Kompakte Schreibweise um aus einer Sequenz eine andere Sequenz zu erzeugen

- Normaler Ansatz:

- Erzeuge leere Liste

- Durchlaufe gegeben Liste

- Füge Elemente der leeren Liste hinzu

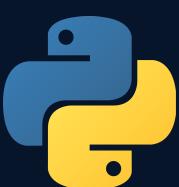
- List Comprehension erzeugt eine Liste und füllt diese direkt

```
# standard for-loop
a = []
for i in range(10):
    a.append(i**2)

# list comprehension
a = [i**2 for i in range(10)]

# list comprehension with condition
a = [i**2 for i in range(10) if i % 2 == 0]

# list comprehension for dictionaries
# creates (key, value)-pairs from sequence
a = {i: i**2 for i in range(10)}
```

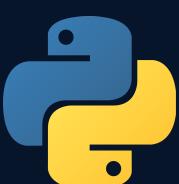


# Funktionen

- Funktionen werden mit dem Keyword **def** eingeleitet
- Es folgt der Name, alle Parameter in runden Klammern und :
  - Parameter haben keinen Typ
- Rückgabe erfolgt über **return**
  - Funktionen in Python haben keinen spezifischen Typ für die Rückgabe
  - Falls eine Funktion keinen Rückgabewert hat, so wird **None** zurückgegeben

```
def add(a, b):  
    return a + b  
  
def hello_world():  
    print("hello world")  
  
result = add(1, 2)  
print(result)  
  
result = hello_world()  
print(result)
```

```
3  
hello world  
None
```



- Funktionen können mehrere Rückgabewerte haben
- Dies wird durch Listen oder Tuple als Rückgabewert realisiert

```
def foo():
    # return tuple
    return (1, 2, 3)

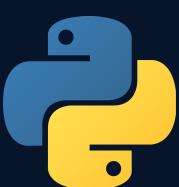
def bar():
    # return list
    return [1, 2, 3]

def foobar():
    # automatically groups return values as tuple
    return 1, 2, 3

result = foobar()
print(result)

# automatically extract values from tuple/list
a, b, c = foobar()
print(a, b, c)
```

```
(1, 2, 3)
1 2 3
```



# Optionale und Positionelle Parameter

▸ Funktionen



- Parameter können Default-Werte erhalten; gekennzeichnet durch =<value>
- Parameter ohne Default-Wert können nicht rechts von Parametern mit Default-Wert stehen
- Parameter können einer Funktion in beliebiger Reihenfolge übergeben werden
- Der Name des Parameters muss beim Funktionsaufruf angegeben werden

```
def foobar(a, b=2, c=100):  
    print(a, b, c)  
  
foobar(1, 2)  
  
foobar(1, 2, 3)  
  
foobar(c=1, b=3, a=5)  
  
foobar(1, c=7)
```

```
1 2 100  
1 2 3  
5 3 1  
1 2 7
```



# Funktionen sind Objekte

► Funktionen



- Funktionen sind Objekte
- Funktionen können daher einer Variable zugewiesen werden
- Funktionen können als Parameter für andere Funktionen genutzt werden

```
def add(a, b):  
    return a + b  
  
def sub(a, b):  
    return a - b  
  
def apply_thrice(a, b, func):  
    a = func(a, b)  
    a = func(a, b)  
    a = func(a, b)  
    return a  
  
func = add  
print(apply_thrice(0, 1, func))  
  
func = sub  
print(apply_thrice(0, 1, func))
```

3  
-3



- Funktionen die thematisch zusammengehören können in einer Python-Datei ausgelagert werden
- Diese Datei nennt man Modul und sie kann in jede andere Python-Datei importiert werden um die Funktionen nutzen zu können
- Viele solcher Module sind Teil der Standard Python Installation - genannt Python Standard Library
- Module können geladen werden mittels: **import <filename>**
  - Die Endung .py wird dabei immer weggelassen

```
# load all functions and variables from the math-module
import math

# to access functions the module-name needs to be used
result = math.sin(math.pi)

# only a subset of functions can be imported
from math import sin, pi

# in this case the module-name can be omitted
result = sin(pi)

# the module-name can be changed on import
import math as m

result = m.sin(m.pi)
```



- Anonyme Funktionen ohne Namen
- Werden auch Lambda-Funktion genannt
- Werden Inline als Parameter übergeben oder in einer Variable gespeichert
- Können nur eine Aktion ausführen!
  - Geben immer einen Wert zurück
  - Nur für simple Aufgaben geeignet für die nicht extra eine Funktion definiert werden soll

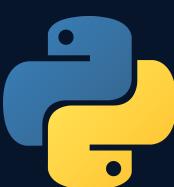
```
f = lambda x, y: x + y
print(f(1, 1))

print((lambda x: x**2)(3))

def foo(bar):
    return bar(1, 2)

print(foo(lambda x, y : (x - y)**2))
```

2  
9  
1



# Exceptions

# Behandeln von Exceptions (1)

▶ Exceptions



- Exceptions sind Fehlermeldungen die während der Laufzeit eines Programms auftreten
- Wird eine Exception nicht zur Laufzeit behandelt wird das Programm beendet
- Exception können mit **try** und **except** Blöcken behandelt werden
- Der Code im **try** Block wird ausgeführt, falls eine Exception auftritt wird der **except** Block ausgeführt
- Nach der Behandlung der Exception wird das Programm normal weiter ausgeführt

```
try:  
    s = "Hello World"  
    i = int(s)  
except ValueError:  
    print("That is not a number")
```



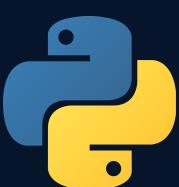
# Behandeln von Exceptions (2)

▶ Exceptions



- Ein **except** Block kann mehrere Exceptions abfangen:  
**except (ValueError, TypeError, NameError):**
- Es können mehrere **except** Blöcke an ein **try** Block gehangen werden
- Der letzte **except** Block kann ohne spezifizierte Exception alle möglichen Exceptions abfangen
- **else** Block wird ausgeführt wenn kein **except** Block ausgeführt wurde
- **finally** Block wird immer ausgeführt

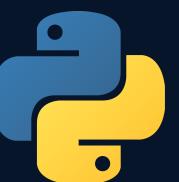
```
try:  
    s = input("Enter a number: ")  
    number = 1 / float(s)  
except ValueError:  
    print("Not a number")  
except ZeroDivisionError:  
    print("Cannot divide by zero")  
except:  
    print("unspecified error occurred")  
else:  
    print("No Error occurred")  
finally:  
    print("End of try")
```



- Mit dem Keyword **as** kann man auf ein Exception Objekt zugreifen
- Für **EnvironmentError** hat das Objekt drei Attribute (int, str, str)
  - Fehler die bei der Öffnung von Dateien auftreten
- Für alle anderen Exceptions ist das Objekt ein String

```
try:  
    f = open("file")  
except IOError as e:  
    print(e.errno)  
    print(e.filename)  
    print(e.strerror)  
    # basic error string which all exception have  
    print(e)
```

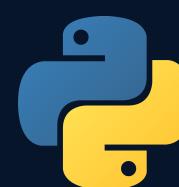
```
2  
file  
No such file or directory  
[Errno 2] No such file or directory: 'file'
```



- Funktion ruft andere Funktion auf
- Die zweite Funktion bekommt eine Exception
- Falls die Exception nicht behandelt wird, wird diese Exception an die aufrufende Funktion weitergegeben
- Aufrufende Funktion kann die Exception selber behandeln oder ebenfalls weitergeben
- Funktion kann selber eine Exception werfen mit dem Keyword **raise**

```
# passing on exception
try:
    s = "Hello"
    i = int(s)
except ValueError as e:
    print(e)
    raise

# raise exception
def division(a, b):
    if b == 0:
        raise ZeroDivisionError()
    return a / b
```



- Funktionsparameter haben keinen Typ
- Auf den richtigen Typ überprüfen mit `type(x)` bevor der Code ausgeführt wird und Exceptions vermeiden
- Oder Code ausführen und dann Exceptions behandeln
- Es ist besser Exceptions zu behandeln
  - Duck-Typing ermöglicht das die Funktion auch für Typen funktioniert die vorher nicht bedacht wurden

# Objektorientierte Programmierung

- Soweit wurde prozedurales programmieren besprochen
  - Daten (Standard-Typen, Variablen, Parameter, ...)
  - Funktionen die Daten als Parameter nehmen und neue Werte zurückgeben
  - Daten und Funktionen sind voneinander unabhängig
- Alternative: Verbinde Daten und Funktionen die zusammen gehören in selbst definierten Datentypen
  - Klassen

- Eine Klasse wird mit dem Keyword **class** eingeleitet
- Die Klasse ist ein selbst definierter Datentyp
- Ein Objekt ist eine Instanz dieses Datentyps
- Attribute sind Daten die zu dieser Klasse gehören
  - Attribute können dynamisch zur Laufzeit hinzugefügt werden
  - Eine Klasse kann so als eine Sammlung von bestimmten Datentypen fungieren

```
class Point:  
    # pass is a No Operation function  
    pass  
  
    # create an instance of class Point  
p = Point()  
print(type(p))  
  
    # add attributes dynamically  
p.x = 2  
p.y = 3
```



- Attribute können zur Laufzeit hinzugefügt werden
  - Zwei Objekte des selben Typs können unterschiedliche Attribute haben
- Attribute können einer Klasse fest zugewiesen werden
  - Jedes Objekt des selben Typs muss diese Attribute haben
- Attribute werden im Konstruktor definiert
  - Funktion: \_\_init\_\_
  - Kann wie jede Funktion Parameter haben die bei der Objekterzeugung angegeben werden müssen
  - **self** ist eine Referenz auf das Objekt selber und muss immer angegeben werden

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
p = Point(2, 3)  
  
print(p.x, p.y)
```

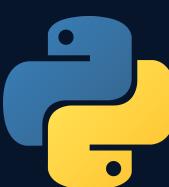
2 3



- Methoden sind Funktionen innerhalb der Klasse
- Beim Funktionsaufruf bekommen diese Funktionen das Objekt selber automatisch als ersten Parameter übergeben
  - Die Funktion muss daher immer **self** als ersten Parameter haben
- Überladen von Funktionen ist nicht möglich
  - Es kann immer nur eine Funktion mit dem selben Namen geben

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def print(self):  
        print(self.x)  
        print(self.y)  
  
p = Point(2, 3)  
p.print()
```

2  
3



- Wenn ein Objekt zu einem String konvertiert wird, wird standardmäßig der Klassenname und die Speicheradresse der Instanz ausgegeben

- Dieses Verhalten kann überschrieben werden

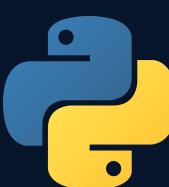
- Funktion: \_\_str\_\_

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        return f"Point at location ({self.x}, {self.y})"  
  
p = Point(2, 3)  
  
print(p) # implicit conversion to str -> str(p)  
  
Point at location (2, 3)
```



- Standardmäßig vergleicht `==` die Identität zweier Objekte
  - Zwei Objekte sind nur gleich wenn sie dasselbe Objekt sind
- Verhalten kann überschrieben werden
  - Funktion: `__eq__`

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
p1 = Point(2, 3)  
p2 = Point(2, 3)  
print(p1 == p2)  
  
class EqPoint:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y  
  
p1 = EqPoint(2, 3)  
p2 = EqPoint(2, 3)  
print(p1 == p2)  
# check for identity  
print(p1 is p2)  
  
False  
True  
False
```

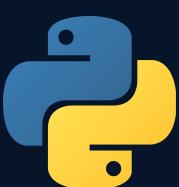


- Viele Standard-Operatoren können auf diese Weise überladen werden
- < : `__lt__(self, other)`
- <= : `__le__(self, other)`
- != : `__ne__(self, other)`
- > : `__gt__(self, other)`
- >= : `__ge__(self, other)`
- + : `__add__(self, other)`
- - : `__sub__(self, other)`
- ...

- Klassen können auch Standard Datentypen emulieren
  - Ermöglicht die Nutzung von Operatoren von diesen Datentypen
    - z.B. `len(myobj)`, `myobj[...]`, `x in myobj`
  - Iteratoren; ermöglicht die Klasse in einer For-Schleife zu durchlaufen
  - Siehe Dokumentation:  
<http://docs.python.org/3/reference/datamodel.html>  
für Funktionen die zu überladen sind

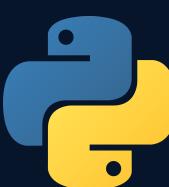
- Attribute die im Konstruktor deklariert werden sind in allen Objekten dieser Klasse vorhanden
- Jedes Objekt hat seine eigene Instanz dieser Attribute
- Attribute die außerhalb des Konstruktors deklariert werden sind für alle Objekte dieser Klasse gleich
- Jedes Objekt greift auf dieselbe Instanz dieses Attributs zu

```
class Point:  
    count = 0  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
        Point.count += 1  
  
p1 = Point(2, 3)  
p2 = Point(2, 3)  
  
print(p1.count)  
print(p2.count)  
print(Point.count)  
  
2  
2  
2
```



- Oft gibt es Klassen die sich sehr ähnlich sind
- Eine Klasse ist nur ein Spezialfall einer anderen Klasse
- Vererbung erlaubt eine hierarchische Struktur für Klassen
- Unterklassen erben den Code der Oberklasse
- Beispiel: Person und Student
  - Jeder Student ist auch eine Person
  - Jeder Student sollte alle Attribute haben die eine Person hat

```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
    def get_name(self):  
        return self.name  
  
class Student(Person):  
    def __init__(self, name, nr):  
        super().__init__(name)  
        self.nr = nr  
  
student = Student("Max", 42)  
print(student.get_name()) # inherited from Person
```



# Überschreiben von Methoden

► OOP



- Unterklasse kann Methoden der Oberklasse überschreiben
- Methoden der Oberklasse können über **super** oder den Klassennamen der Oberklasse aufgerufen werden

- Aufruf über Klassenname:  
**Person.get\_name(self)**

- Methode wird in der Klasse Person gesucht

- self muss als Parameter übergeben werden

- Keyword **super** sucht die Methode in einer beliebigen Oberklasse

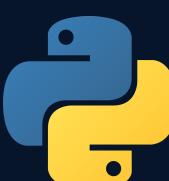
```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
    def get_name(self):  
        return self.name  
  
class Student(Person):  
    def __init__(self, name, nr):  
        super().__init__(name)  
        self.nr = nr  
  
    def get_name(self):  
        return f"Student {super().get_name()}"  
  
student = Student("Max", 42)  
print(student.get_name()) # inherited from Person
```



- Es ist möglich dass eine Unterklasse mehrere Oberklassen hat
- Zwei Oberklassen können Methoden oder Attribute mit demselben Namen haben
  - Von welcher Oberklasse wird das Attribut bzw. Methode übernommen?
  - Gesucht wird von links nach rechts in der Liste der Oberklassen
    - Erster Treffer wird vererbt
    - Dabei werden auch alle Oberklassen der Oberklasse durchsucht

```
class ParentA:  
    def foo(self):  
        print("ParentA")  
  
    def bar(self):  
        print("ParentA")  
  
class ParentB():  
    def foo(self):  
        print("ParentB")  
  
    def bar(self):  
        print("ParentB")  
  
class Child(ParentA, ParentB):  
    def bar(self):  
        super().bar()  
  
child = Child()  
child.foo()  
child.bar()
```

ParentA  
ParentA



- In Python gibt es keine privaten Methoden oder Attribute
- Stattdessen gibt es eine Konvention: Variablen die nicht außerhalb der Klasse genutzt werden sollten beginnen mit einem Unterstrich
  - z.B. `_foo`
- Um Namenskonflikte von „privaten“ Variablen bei der Vererbung zu vermeiden kann eine Variable mit zwei Unterstrichen beginnen
  - `__foo` wird ersetzt durch `__classname__foo`

- Wenn bestimmte Aktionen beim Zugriff auf Variablen durchgeführt werden sollen, können getter und setter Methoden verwendet werden
- Beim Zugriff auf die Variable wird automatisch die entsprechende Methode aufgerufen
- Aufzurufende Methode wird durch Keyword **property** bestimmt
  - Hier: Beim Zugriff auf die Variable `x` werden die Methoden `get_x` und `set_x` aufgerufen
  - Die Methoden `get_x` und `set_x` greifen dann auf die Variable `_x` zu
  - Die Variable `_x` ist nach wie vor nach außen sichtbar

```
class Point:  
    def __init__(self, x, y):  
        self._x = x  
        self._y = y  
  
    def set_x(self, x):  
        self._x = x  
        print(f"New Value for x: {x}")  
  
    def get_x(self):  
        print(f"x is {self._x}")  
        return self._x  
  
x = property(get_x, set_x)  
  
p = Point(3, 2)  
p.x  
p.x = 5  
  
x is 3  
New Value for x: 5
```



# NumPy

- NumPy ist eine Python-Bibliothek und steht für „Numerical Python“
- NumPy verbessert die Funktionsweise von Arrays und bietet viele mathematische Funktionen für Arrays
- NumPy ersetzt dabei Pythons Arrays (**list**) durch einen eigenen Datentyp (**ndarray**)
- **ndarray** sind bis zu 50x schneller als Python **list**

- Voraussetzung: Python und der Paketmanager PIP sind installiert
- NumPy kann über PIP installiert werden:  
`pip install numpy`
- NumPy kann nun in jedem Python-Script importiert werden
  - `import numpy`
- Normalerweise wird NumPy mit dem alias np importiert  
`import numpy as np`

- Erzeugung von **ndarray** aus einer Sequenz mit der Funktion **array**

- Arrays können geschachtelt sein

- Jede Schachtelung ist eine Dimension

- Array mit 0 Dimensionen ist ein Skalar

- Array mit 1 Dimension ist ein Vektor

- Array mit 2 Dimensionen ist eine Matrix

- ... etc

- Für weitere Funktionen siehe:  
<https://numpy.org/doc/stable/reference/routines.array-creation.html>

```
import numpy as np

foo = np.array([1, 2, 5, 6, 7])

bar = np.array([[1,2,3], [4,5,6], [7,8,9]])

foobar = np.arange(5)

print(foo)
print(bar)
print(foobar)
```

[ 1 2 5 6 7]  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]  
[0 1 2 3 4]



- **WICHTIG:** Im Gegensatz zu Python List muss jedes Item in einer Dimension die gleiche Anzahl an Dimensionen haben!
- i.e. eine Matrix muss für jede Reihe (Dimension 1) dieselbe Anzahl an Spalten haben (Dimension 2)
- Daher kann für jedes NumPy-Array immer genau bestimmt werden wie es aufgebaut ist
  - Darstellung als Tupel wo jeder Wert angibt wie viele Items in der entsprechenden Dimension vorhanden sind
  - Z.B. eine NxM-Matrix kann als Tupel (N, M) beschrieben werden mit N Reihen und M Spalten und jeder Wert ist ein Skalar
  - Z.B. eine NxM-Matrix kann als Tupel (N, M, V) beschrieben werden mit N Reihen und M Spalten und jeder Wert ist ein Vektor der Länge V

- Die zuvor beschriebene Darstellung nennt man **Shape**
  - Shape ist ein Attribut eines jeden **ndarray**
- Shape bestimmt grundlegend wie zwei NumPy-Arrays miteinander verknüpft werden können
  - Z.B. Matrixmultiplikation von A, B ist nur möglich, wenn Anzahl der Spalten (Dimension 2) von A gleich Anzahl der Zeilen (Dimension 1) von B
- Die Anzahl der gesamt Items kann aus dem Shape ermittelt werden, indem alle Dimensionen multipliziert werden
  - Shape (2, 5) ->  $2 \cdot 5 = 10$  Items
- Umgekehrt schränkt die Anzahl der Items die Menge der möglichen shapes ein
  - 10 Items -> eine Dimension mit 3 Items ist nicht möglich!

```
import numpy as np

foo = np.array([1, 2, 5, 6, 7])
bar = np.array([[1,2,3], [4,5,6], [7,8,9]])
foobar = np.array([[[1]], [[2]], [[3]]])

print(foo.shape)
print(bar.shape)
print(foobar.shape)
```

```
(5,)
(3, 3)
(3, 1, 1)
```



- Im Rahmen der zuvor beschriebenen Einschränkungen kann der Shape eines **ndarray** geändert werden
  - Z.B. Shape (2, 5) -> 10 Items -> Shape (5, 2)
- Shaping ändert nicht wie der Array gespeichert wird sondern nur wie auf das Array zugegriffen wird!
- Funktion um den Shape zu ändern:  
**ndarray.reshape(shape)**
  - Eine Dimension kann -1 sein, in dem Fall wird die korrekte Dimension berechnet

```
import numpy as np
foo = np.arange(10)
foo = foo.reshape(5, 2)

print(foo)

print(foo[2][1])

foo = foo.reshape(2, 5)

print(foo)

# we switched dimension 1 with dimension 2
# one could assume that switching the indexes
# when accessing the array
# would yield the same result
print(foo[1][2])

# this is not the case because the array is just
# a list of values and the shape describes which position
# of the list should be accessed

# index_1 * shape[1] + index_2
# 2 * 2 + 1 = 5 -> 6th item -> 5 in range(0, 10)
# 1 * 5 + 2 = 7 -> 8th item -> 7 in range(0, 10)
```



# Shaping (2)

► NumPy



- Dimensionen mit Länge 1 können beliebig hinzugefügt werden

$$\bullet 2 \cdot 5 = 10 = 1 \cdot 2 \cdot 1 \cdot 5 \cdot 1$$

- Kann genutzt werden um einen Vektor in eine Matrix mit einer Zeile/Spalte umzuändern

- Entferne Dimension mit Länge 1: `numpy.squeeze()`

- Füge Dimension mit Länge 1 hinzu:  
`a[None]` oder `a[newaxis]` oder  
`numpy.expand_dims()`

- Oder `numpy.reshape()`

- Tausche Dimensionen sodass dasselbe Element mit denselben Indexen angesprochen wird:  
`numpy.swapaxes()`

```
import numpy as np
foo = np.arange(10)
foo = foo.reshape(5, 2)

print(foo[2][1])
print(foo.shape)
print(foo[None].shape)
print(foo[:, np.newaxis].shape)

foo = foo.reshape(-1, 2, 1)
print(foo.shape)

foo = foo.squeeze()
print(foo.shape)

foo = foo.swapaxes(0, 1)
print(foo.shape)
print(foo[1][2])
```

5  
(5, 2)  
(1, 5, 2)  
(5, 1, 2)  
(5, 2, 1)  
(5, 2)  
(2, 5)  
5



- Transpose Array
  - Bei Matrix werden Zeilen und Spalten vertauscht
    - `ndarray.T` oder `numpy.transpose()`
  - Entferne alle Dimensionen
    - Endergebnis ist ein Vektor mit shape (N)
    - `ndarray.flatten()` oder `ndarray.ravel()`
  - Für mehr Funktionen siehe:  
<https://numpy.org/doc/stable/reference/routines.array-manipulation.html>

- NumPy-Arrays haben dieselben Möglichkeiten zur Indexierung wie Python-Lists
  - Slicing ( `a[start:end:step]` )
  - Negativer Index ( `a[-1]` )
- NumPy-Arrays erweitern die Indexierung um die Möglichkeit jede Dimension direkt anzusprechen
  - Indizes werden durch Kommata in derselben eckigen Klammer getrennt

```
import numpy as np

foo = np.arange(10).reshape(2, 5)
foo_list = list(foo)

# foo_list is a list which contains other lists
# if we want to access an item of the nested lists
# we first need to retrieve the list the item is contained in
foo_nested_list = foo_list[0]
foo_nested_list[0]
# or directly
foo_list[0][0]

# however, since we need to retrieve the nested list first
# we cannot access the same item for EVERY nested list
# contained in foo_list
# i.e. we cannot access the nested list directly
# instead we need to retrieve each nested list
# and access the desired item
print([nested_list[0] for nested_list in foo_list])

# in numpy we can directly access each dimension
# i.e. we do not need to retrieve the nested dimensions
# the indices for nested dimensions are separated by commas
# in this case we want all items in the first dimension (:)
# and the first item in the second dimension (0)
print(foo[:, 0])
```



# Indexing (2)

▶ NumPy



- NumPy ermöglicht es Listen bzw. Arrays als Index zu benutzen
- NumPy ermöglicht es eine boolean-Liste (sogenannte boolesche Maske) als Index zu benutzen
  - Für jedes Item wird der Eintrag in der booleschen Maske an derselben Stelle überprüft
  - Ist die Maske an der Stelle True, wird das Item ausgewählt
  - Ist die Maske an der Stelle False, wird das Item nicht ausgewählt

```
import numpy as np

foo = np.arange(6)
print(foo)

# lists can be used to specify which items
# should be selected and in which order
print(foo[[5, 4, 3]])

boolean_mask = [True, False] * 3
print(boolean_mask)

# access each item of foo where the mask is True
print(foo[boolean_mask])

# boolean masks can be used to filter arrays
boolean_mask = foo > 3
print(boolean_mask)
print(foo[boolean_mask])

[0 1 2 3 4 5]
[5 4 3]
[True, False, True, False, True, False]
[0 2 4]
[False False False False  True  True]
[4 5]
```



# Numerische Operationen auf Arrays

▶ NumPy



- Numerische Operatoren  $+, -, *, /$  können direkt auf zwei NumPy-Arrays mit demselben Shape angewandt werden
- Dabei wird jedes Item aus dem linken Array mit dem Item aus dem rechten Array an derselben Stelle verknüpft
- Wichtig: Es handelt sich hierbei nicht um mathematische Funktionen wie Matrixmultiplikation oder Skalarprodukt!

```
import numpy as np

foo = np.arange(1, 7)
bar = np.arange(7, 13)

print(foo + bar)
print(foo - bar)
print(foo * bar)

[ 8 10 12 14 16 18]
[-6 -6 -6 -6 -6 -6]
[ 7 16 27 40 55 72]

foo = foo.reshape((2, 3))
bar = bar.reshape((2, 3))

print(foo + bar)
print(foo - bar)
print(foo * bar)

[[ 8 10 12]
 [14 16 18]]
[[-6 -6 -6]
 [-6 -6 -6]]
[[ 7 16 27]
 [40 55 72]]
```



- Zwei Arrays mit unterschiedlichen Shapes können ebenfalls verknüpft werden
- NumPy versucht einen Array so zu erweitern das beide Arrays denselben Shape haben
  - Wird Broadcasting genannt
  - Broadcasting ist nicht immer möglich
  - NumPy zeigt in keiner Form an wann Broadcasting angewandt wurde!

```
import numpy as np [ 5 6 7 8 9 10]

foo = np.arange(6) [[0 1]
bar = 5 [2 3]]

# bar has a different shape than foo [[ 0 11]
# bar is stretched to the same shape as foo [ 2 13]]
# missing values are filled with bar
# thus we get a vector filled with bar
print(foo + bar) [[ 0  1]
[12 13]]
```

```
bar = np.array([0, 10])
# bar has a different shape than foo
# however, we cannot extend bar in such a way
# that bar and foo have the same shape and
# missing values are filled with bar!
```

```
# foo + bar -> not possible
```

```
foo = np.arange(4).reshape(2, 2)
print(foo)
```

```
# foo is a 2x2 matrix and bar is a vector with length 2
# broadcasting could use bar as a 2 rows or 2 columns!
# in this case bar is used as rows
print(foo + bar)
# use bar as columns by reshaping it to a column vector
print(foo + bar[:, None])
```



- NumPy stellt eine gewaltige Sammlung von Funktionen für den Umgang mit Arrays zur Verfügung
  - Konkatenation von Arrays (vstack, hstack, concatenate, ...)
  - linearen Algreba
    - Länge eines Vektors, Skalarprodukt, Matrixmultiplikation, ...
  - Mathematische Routinen (Trigonometrie, Logarithmus, ...)
  - Statistics
  - ...
- Für eine Übersicht siehe die NumPy Dokumentation  
<https://numpy.org/doc/stable/reference/index.html>

- Installation
  - Conda (<https://docs.conda.io/en/latest/miniconda.html>)
  - Python (<https://www.python.org/downloads/>)
- API Reference
  - Python (<https://docs.python.org/3.8/library/index.html>)
  - NumPy (<https://numpy.org/doc/stable/contents.html>)