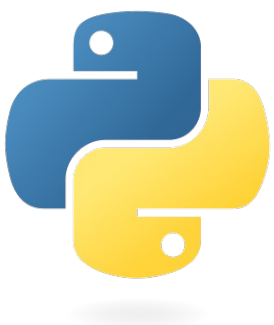
HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Programmierung mit



python™

Listen - Rückblick

Listen sind eine **sequentielle Folge** von Objekten und veränderbar.

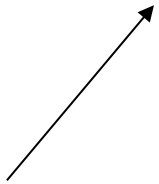
```
list = ["python", "refugeeks", "I am a string!"]
```


```
list[1] = 'Apple'
print(list) # ["python", "Apple", "I am a string!"]
```

Listen in Listen

Listen können auch selbst Listen enthalten

```
list = [[True, False, True], [25, 63, 23]]
```



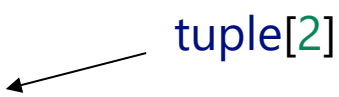
list[0][2]

Tupel

Auch Tupel sind eine Folge von Objekten

```
tuple = ("python", "refugeeks", "I am a string!")
```

Zugriff auf Index genau wie bei Listen




```
tuple = ("python", "refugeeks", "I am a string!")
```

Unterschied zu Listen:

Tupel sind nicht veränderbar (eng: *immutable*)

```
tuple[2] = "Test" # Fehler: TypeError
```

Dictionary

Dictionaries speichern Werte in key:value-Paaren

```
dict = {
    "Name": "Julia",
    "Job": "Software Engineer",
    "degrees": ["B. Sc.", "M. Sc."]
}

print("Name: ", dict[“Name”]) # Julia
print("Job: ", dict[“Job”]) # Software Engineer
```

Dictionaries sind veränderbar und geordnet*, lassen aber **keine** Duplikate zu

```
dict[“Name”] = "Kelvin" # Eintrag ändern
del dict[“Name”]; # Eintrag 'name' löschen
dict.clear(); # Alle Einträge löschen
```

*ab Python Version 3.7

Dictionary – Iterieren - Keys

Um mit einem for loop über einen Dictionary zu iterieren, gibt es mehrere Möglichkeiten:
Die Keys können auf zwei Wegen ausgegeben werden.

```
my_dict = {
    "Name": "Julia",
    "Job": "Software Engineer",
    "degrees": ["B. Sc.", "M. Sc."]
}


for key in my_dict:
    print(key) # Name, Job, degrees

# alternativ
for key in my_dict.keys():
    print(key) # Name, Job, degrees
```

Dictionary – Iterieren - Values

Um mit einem for loop über einen Dictionary zu iterieren, gibt es mehrere Möglichkeiten:
Auch die Values können über zwei Wege ausgegeben werden.

```
my_dict = {
    "Name": "Julia",
    "Job": "Software Engineer",
    "degrees": ["B. Sc.", "M. Sc."]
}

for key in my_dict:
    print(my_dict[key]) # Julia, Software Engineer, ['B. Sc.', 'M. Sc.']

# alternativ
for value in my_dict.values():
    print(value) # Julia, Software Engineer, ['B. Sc.', 'M. Sc.']
```

Dictionary – Iterieren – Keys und Values

Um mit einem for loop über einen Dictionary zu iterieren, gibt es mehrere Möglichkeiten:
Das folgende Beispiel zeigt, wie man sowohl den Key als auch das Value ausgeben kann

```
my_dict = {
    "Name": "Julia",
    "Job": "Software Engineer",
    "degrees": ["B. Sc.", "M. Sc."]
}

# Kombination
for key, value in my_dict.items():
    print(key, value)
```

Aufgaben 1

1. Schreibt eine Funktion, welche für einen gegebenen Monat, die Anzahl der Tage in diesem Monat zurückgibt. (Für den Februar soll immer 28 zurückgegeben werden.)

Nutzt einen Dictionary um diese Aufgabe zu lösen.

Aufgaben 1

2. Schreibt ein Programm, welches einen String zu einen Dictionary konvertiert.

Die Keys sollen dabei die Buchstaben sein und die Values die Positionen wo sich der Buchstabe im Alphabet befindet. Es reicht wenn das Programm nur Wörter mit Kleinbuchstaben bearbeiten kann.

```
Welcher String soll konvertiert werden?: refugeeks
{'r': 18, 'e': 5, 'f': 6, 'u': 21, 'g': 7, 'k': 11, 's': 19}
```

Klassen & Objekte - Motivation

Funktionen bieten kein Konzept für die Verwaltung strukturierter Daten

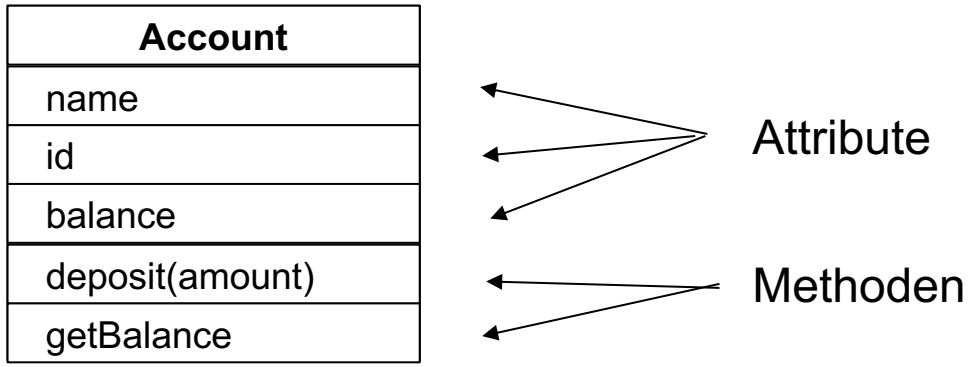
Beispiel: Wie würde man ein Konto verwalten?

```
# Falsch!
["Max", 22424, 52.23] # name, id, kontostand
```

-> Ungeeignet für komplexe Strukturen

Klassen - Objekte

Objekte besitzen Daten und Funktionen und sind Instanzen einer Klasse



Klassen - Definition

self verweist auf das Objekt

```
class Account:
    def __init__(self, name, id, balance):
        self.name = name
        self.id = id
        self.balance = 0.0

    def __str__(self):
        return str(self.name) + ' ' + str(self.id) + ': ' + str(self.balance)

    def deposit(self, amount):
        self.balance = self.balance + amount

    def getBalance(self):
        return self.balance
```

Konstruktor:

- Spezielle Funktion
- Initialisiert Attribute
- Wird für die Erzeugung der Objekte genutzt (und dann ausgeführt)

__str__:

- Spezielle Funktion
- Beschreibt das Objekt als String

Fachliche Methoden

Klassen - Objekte erzeugen

Instanzen von Objekten werden mit dem Aufruf der Klasse erstellt

```
# Objekte erzeugen
acc1 = Account("Max", 22424, 200.0) # def __init__(self, name, id, balance):
acc2 = Account("Julia", 22425) # def __init__(self, name, id):

# Methoden aufrufen
acc1.deposit(500)
print(acc1.getBalance()) # 500
acc2.deposit(250)
acc2.deposit(105)
print(acc2) # Julia 22425: 355.0
```

Klassen - Objekte verändern

Objekte sind veränderbar (mutable)

```
# direkter aufruf von balance
acc1 = Account("Max", 22424)
acc1.balance = 2500
# besser: acc1.setBalance(2500)
print(acc1.getBalance())
```

acc1.kontonummer

Attribute können privat sein

- Schützt Variablen vor ungewollter Veränderung

```
class Account:
    def __init__(self, name, id):
        self.__name = name
        self.__id = id
        self.__balance = 0.0


acc_private = PrivateAccount("Max", 22426)
print(acc_private.__name) # AttributeError
```

Aufgaben 2

2.

- Schreibt eine Klasse **Person** mit den Attributen *name, gewicht* und *groesse*
- Außerdem soll eine Funktion getBMI in der Klasse implementiert werden, welche den BMI der Person zurückgibt.
- Erstellt abschließend ein Objekt mit der Klasse **Person**

Person
name
gewicht
groesse
getBMI()

Aufgaben 2

Schreibt die Klassen **Kreis, Rechteck und Quadrat** wie unten beschrieben.

Sie haben zwei Methoden, welche die Fläche oder den Umfang des Objekts berechnen.
Außerdem soll über die Klasse **GeometrieRechner** entschieden werden, welche Figur berechnet werden soll.

Die Attribute wie radius oder a, b sollen über input in der Funktion *setValues()* eingegeben werden.

GeometrieRechner
figur
setFigur()
getFlaeche()
getUmfang()

Kreis
radius
getFlaeche()
getUmfang()
setValues()

Rechteck
a
b
getFlaeche()
getUmfang()
setValues()

Quadrat
a
getFlaeche()
getUmfang()
setValues()

Aufgaben 2

3. Schreibt die Klasse **Auto** wie unten beschrieben. Implementiert für alle Attribute die getter- und setter-Methoden. (z.B. *setMarke* oder *getMarke*)

Erstellt anschließend verschiedene Instanzen dieser Klasse von Autos verschiedener Automarken und probiert die getter- und setter-Methoden aus.

Auto
marke
preis
model
baujahr

Aufgaben 2

4. Schreibt die Klasse **Hund** wie unten beschrieben.

Es müssen **keine** getter und setter Methoden geschrieben werden. Erstellt anschließend wieder ein Objekt der Klasse **Hund**

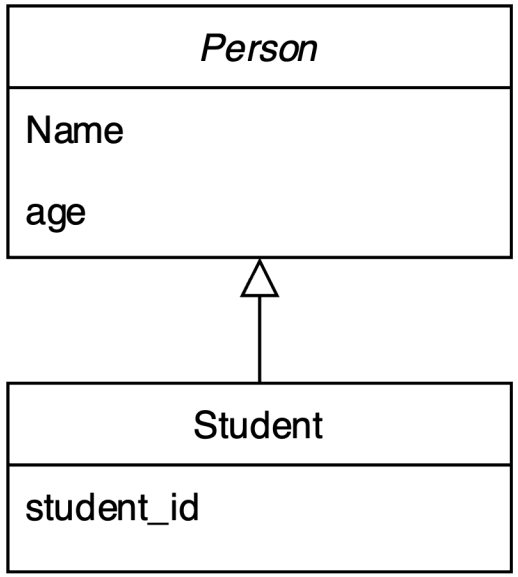
Beim Aufruf der Funktion bellen() soll auf der Konsole der Hund bellen ☺

Hund
name
gewicht
bellen()

Klassen - Vererbung

Problem: Ab einer gewissen Komplexität muss abstrahiert werden

Vererbung erlaubt einer Klasse alle Methoden und Variablen einer anderen Klasse zu übernehmen.

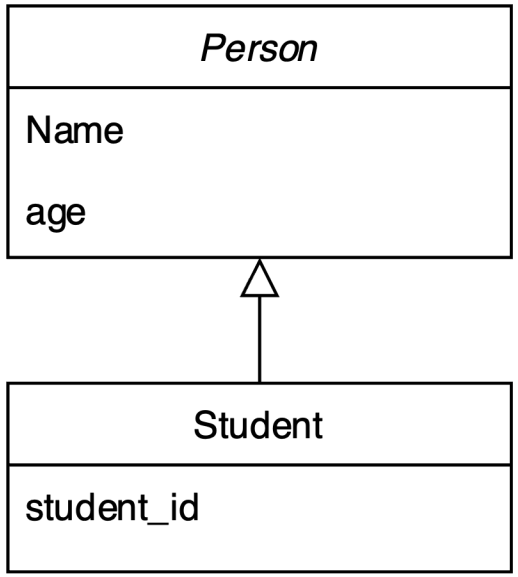



```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def printinfo(self):
        print(self.name, self.age)

class Student(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.student_id = id

    def printinfo(self):
        print(self.name, self.age, self.student_id)
```

Klassen - Vererbung

Vererbung ist ein Konzept aller Objekt-Orientierten Programmiersprachen

Vererbung eignet sich zur Wiederverwendung von Code




```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def printinfo(self):
        print(self.name, self.age)

class Student(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.student_id = id

    def printinfo(self):
        print(self.name, self.age, self.student_id)
```

Klassen – Vererbung - __init__()

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def printinfo(self):
        print(self.name, self.age)

class Student(Person):
    def __init__(self, id):
        self.student_id = id

    def printinfo(self):
        print(self.student_id)
```

Wenn __init__() in der Unterklasse eingefügt wird, erbt sie nicht mehr das__init__() der Oberklasse

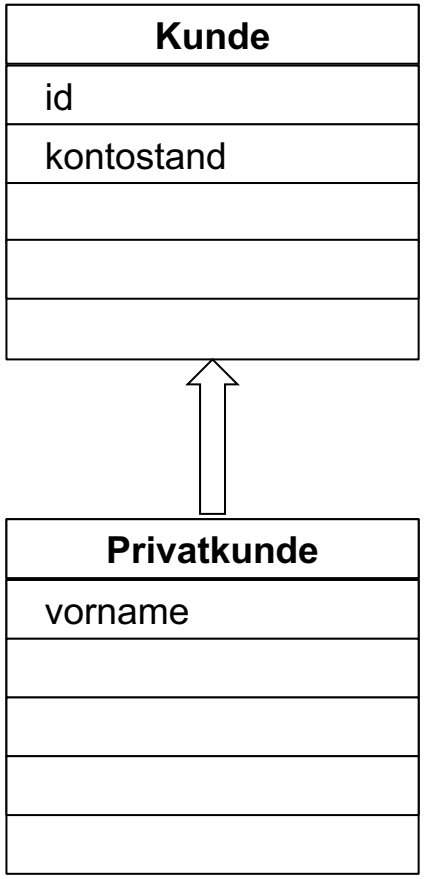
Klassen – Vererbung - super()

super() erlaubt den Aufruf des **__init__()** der Oberklasse (ohne self)

```
class Student(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.student_id = id

    def printinfo(self):
        print(self.name, self.age, self.student_id)
```

Aufgaben 2

4.



Aufgaben 2

5. Schreibt ein Programm mit den folgenden zwei Klassen wie im Diagramm beschrieben.

   Die Klasse **Liga** bietet die folgenden Methoden an:
   meldeAn(team): Meldet ein neues Team in der Liga an
   spiele(team1, …): Aktualisiert die Tore nach einem Spiel der Teams
   gibTabelle(): gibt einen Dictionary mit einer Tabelle der Teams aus

Liga
name: string
teams: [ ]
meldeAn(team)
spiele(team1, teams2, toreTeam1, toreTeam2)
printTabelle(): Dictionary

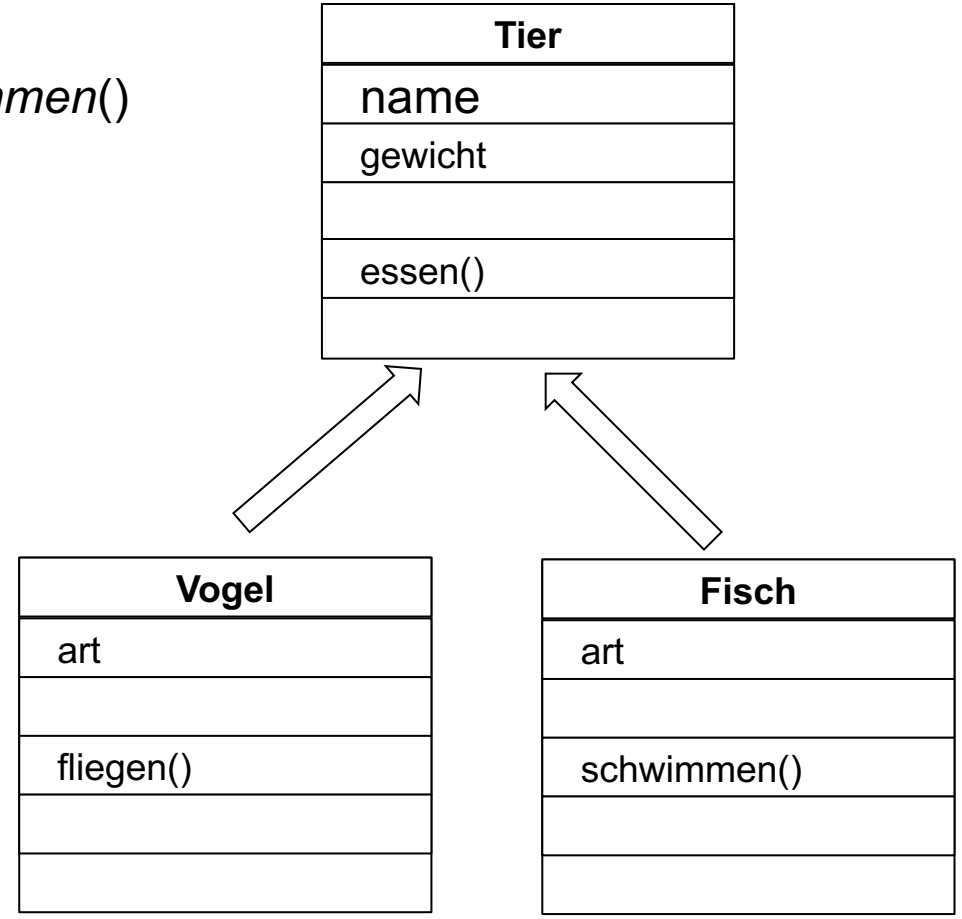
Team
name: string
tore: int
spieler: [ ]
printTeam()

Spieler
name: string
alter: int
karriereStart: int
gehalt: float
position: string

Aufgaben 2

6. Schreibt die folgenden 3 Klassen wie im Diagramm beschrieben.
   Dabei sollen Vogel und Fisch von Tier erben.

   Die Methoden *essen(), fliegen()* und *schwimmen*()
   sollen nur print() ausführen



Fehlerbehandlung

Anwendungen müssen auf Fehler reagieren können ohne abzustürzen

Beispiel: Zugriff auf Indizes außerhalb von Listenrange

```
>>> list = ["python", "refugeeks", "I am a string!"]
>>> print(list[3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Fehlerbehandlung - Exceptions

Exceptions behandeln fehlerhafte Situationen

- Unbehandelte Exceptions beenden das Programm

Wie auf Fehler reagieren?

try/catch um auf Fehler zu reagieren

```
# Auf Exception reagieren
try:
    input = int(input("Eine Zahl eingeben größer 0 eingeben: "))
    print(5 / input)
except ZeroDivisionError:
    print("Fehler: die Eingabe darf nicht 0 sein")
finally: # Optional: wird immer ausgeführt
    print("Fertig")
```

Aufgaben 2

7. Schreibt die Funktion *divide(n)*. Sie soll 1/n zurückgeben. Die Eingabe von *n* soll über *input* stattfinden.

Dabei soll das Programm auf den **ZeroDivisionError** reagieren können und den Fehler auf der Konsole ausgeben.

Beispielausführung vom Programm:

```
Bitte gib eine Zahl ein: 0
Fehler: Division durch Null !!
```

Fehlerbehandlung – Eigene Exceptions

Python definiert viele Exceptions vor die automatisch geworfen werden: ValueError, AttributeError, TypeError, IndexError usw.

Es ist aber oft nötig eigene Exceptions zu schreiben:

```
# Exception abfangen
try:
    acc = Account("Max", 0)
    acc.deposit(500)
    acc.withdraw(1000) # Fehler wird hier geworfen
except BalanceLowException as exception:
    print(exception)
```

```
class BalanceLowException(Exception):
    def __str__(self):
        return "Balance of Account to low to withdraw balance!"

# Methode von Account
def withdraw(self, amount):
    if amount > self.balance:
        raise BalanceLowException()
    else:
        self.balance = self.balance - amount
```

Aufgaben 2

8. Schreibt die Klasse **Kugel** wie unten beschrieben.

Außerdem soll eine eigene Exception **BelowZeroError** geschrieben werden. Diese Exception wird geworfen wenn der übergebene Radius an die Klasse Kugel < 0 ist.

Formel zur Berechnung vom Volumen einer Kugel: $4/3 \cdot \pi \cdot r^3$

Kugel
radius
getVolumen()

BelowZeroError

Aufgaben 2

9. Schreibt eine Funktion insertIntoList(index, item, liste), welche ein Element in eine Liste an einem bestimmten Index einfügt. Diese Funktion soll auf den Fehler **IndexError** mit einer Fehlermeldung reagieren können, ohne das das Programm abstürzt.

Dateien – open()

Mit Python können mit der *open()*-Funktion Dateien gelesen und bearbeitet werden

```
file = open("text.txt", "r")
```

open() benötigt zwei Parameter: Dateiname und Modus.

Es gibt 4-Standard Modi und 2 Optionale

r	**Lesen**	**Standardwert - Öffnet eine Datei zu Lesen**
a	**Append**	**Hängt Daten an eine Datei ein**
w	**Write**	**Schreibt in eine geöffnete Datei**
x	**Create**	**Erzeugt eine Datei**
t	Text	Standardwert - Text-Modus
b	Binary	Binary-Modus

Dateien – read()

Angenommen eine Datei wurde mit open() schon geöffnet, kann der Inhalt mit read() gelesen werden

```
file = open("text.txt", "r")
print(file.read())
```

read(n) liest *n* Charaktere in der Datei

```
print(file.read(3))
```

readline() liest *eine Zeile der Datei*

```
print(file.readline())
```

Nach dem Lesen/Schreiben sollte die Datei wieder geschlossen werden

```
file.close()
```

Dateien – write()

Angenommen eine Datei wurde mit open() schon geöffnet, können Daten mit write() in die Datei geschrieben werden.
Dafür muss die Datei mit dem Append oder Write-Modus geöffnet werden.

Append-Modus fügt Daten an

```
file = open("text.txt", "a")
file.write("\nJetzt steht hier mehr drin!")
file.close()
```

Achtung: Write-Modus überschreibt alle Daten in der Datei

```
file = open("text.txt", "w")
file.write("Ich habe alles überschrieben!")
file.close()
```

* \n ist ein Zeilenumbruch

Aufgaben 3

10. Schreibt eine Funktion *writeIntoFile(string),* welche einen übergebenen String als Parameter in die Datei “strings.txt“ schreibt.

Die Funktion soll auf den Fehler **FileNotFoundError** mit einer Fehlermeldung reagieren können.

Der String der in die Datei geschrieben wird soll über *input* vom Benutzer eingelesen werden.

Aufgaben 3

11. Schreibe eine Funktion countWordOccurence(fileName), welche die Häufigkeiten von Worten in einer .txt-Datei liest und in einem Dictionary zurückgibt.

Dieser Dictionary nutzt das Wort als **key** und die Häufigkeit im Text als **value**.

Aufgaben 3

12. Schreibt ein Python Programm mit der Klasse **Diagramm** wie unten beschrieben. Als Attribute hat diese Klasse titel und data. Titel ist der Titel des Diagramms und data sind die “Balken“ des Diagramms.

printDiagramm() gibt das Diagramm wie folgt auf der Konsole aus:

```
Noten Python 1
Kelvin: ####
Peter: ##
Max: ##
Ali: ###
Gustav: ######
```

Diagramm
titel: string
data: Dictionary
printDiagramm()

Beispiel für data:

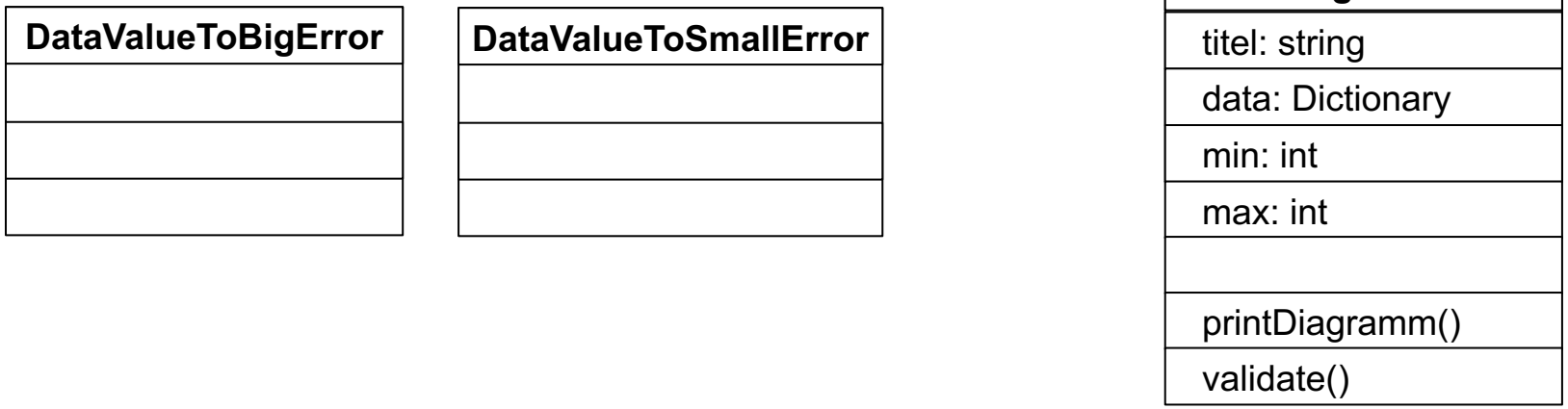
```
data = {"Kelvin": 4, "Peter": 2, "Max": 2, "Ali": 3, "Gustav": 6}
```

Gemeinsame Aufgabe: Wie schafft man es, das die Balken alle auf der gleichen Höhe anfangen?

Aufgaben 3

13. Erweitert die Diagramm-Klasse mit einem min und einem max Attribut.
Außerdem soll die Methode *validate()* implementiert werden.

In dieser Methode soll geprüft werden ob die Values in data nicht größer als max, bzw. kleiner als min sind. Wenn ein Value aus data größer als max ist, wird der Fehler **DataValueToBigError** geworfen, wenn ein Value aus data kleiner als min ist wird der Fehler **DataValueToSmallError** geworfen. Die validate()-Methode soll beim erstellen des Objektes aufgerufen werden.



Aufgaben 3

12. Implementiert die folgenden Klassen wie im Diagramm beschrieben. Die Methoden ausleihen() bei der Klasse Buecherei ruft die Methode ausleihen() in der Klasse Kunde auf. Diese Methode fügt ein Buch zu der ausgeliehen Liste von Kunde hinzu.

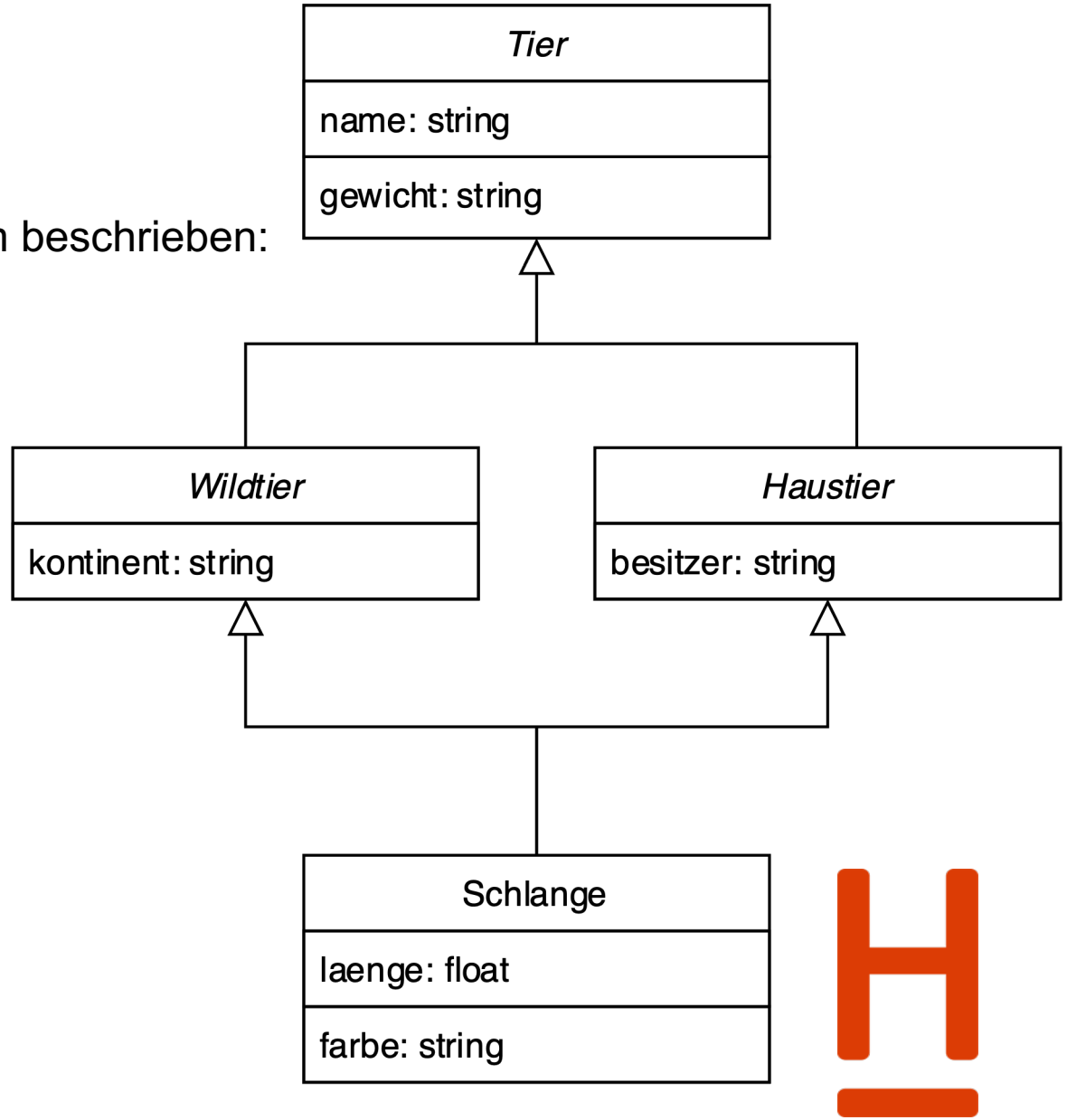
Buecherei
ort: string
buecher: Buch[ ]
ausleihen(kunde, buch)

Buch
titel: string
autor: string
seiten: int

Kunde
name: string
ausgeliehen: Buch[ ]
ausleihen(buch)

Aufgaben 3

13. Implementiert die folgenden Klassen wie im Diagramm beschrieben:



Aufgaben 3

13. Implementiert die folgenden 3 Klassen **Song**, **Artist** und **WrongFormatError**.

In der Song Klasse soll beim erstellen des Objektes geprüft werden ob das file_format korrekt ist.
Wenn das file_format **nicht** „mp3“ oder „wma“ ist, soll der **WrongFormatError** geworfen werden.