

Grundlagen

Felix Döring, Felix Wittwer

Python-Kurs

8. November 2021



Gliederung

1. Wiederholung

- Gemeinsamkeiten und Unterschiede
- Fehler

2. Scopes

- Das Scoping Problem

3. Objekte

- Klassen, Objekte und Attribute
- Methoden

 - Übung: Vektor

- Super- und Subklassen

 - Übung: Planet

- Spezielle Methoden

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
-------------	---------------	----------------

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
Strings?	f"{var}" Format-String	+ -Operator

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
Strings?	f"{var}" Format-String	+ -Operator
Argumente	greeting hat Standard-Argument "Hi"	Beide Argumente sind notwendig

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
Strings?	f"{var}" Format-String	+ -Operator
Argumente	greeting hat Standard-Argument "Hi"	Beide Argumente sind notwendig
Fehler?	keine	Missing Argument

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
Strings?	f"{var}" Format-String	+ -Operator
Argumente	greeting hat Standard-Argument "Hi"	Beide Argumente sind notwendig
Fehler?	keine	Missing Argument
Effekt	Funktion gibt aus	Funktion gibt zurück → in Datei geschrieben

Unterschiede und Gemeinsamkeiten

```
1 def say_hi(name, greet="Hi"):
2     print(f"{greet}, {name}")
3
4 if __name__ == '__main__':
5     say_hi("Kurs", "Hallo")
6     say_hi("Menschen")
```

```
1 def say_hi(name, greet):
2     return greet+", "+name
3
4 file = open("Datei.txt", "w")
5 file.write(say_hi("Kurs"))
6 file.close()
```

Eigenschaft	print-Version	return-Version
Strings?	f"{var}" Format-String	+ -Operator
Argumente	greeting hat Standard-Argument "Hi"	Beide Argumente sind notwendig
Fehler?	keine	Missing Argument
Effekt	Funktion gibt aus	Funktion gibt zurück → in Datei geschrieben
Bedingung	Nur wenn Skript direkt ausgeführt wird (__name__ == "__main__")	wird immer gemacht

Boilerplate

```
1 class ManyClasses:
2     ...
3
4 def a_lot_of_functions():
5     ...
6
7 def main():
8     do_a_lot_of_things_when_executed_directly()
9
10 if __name__ == '__main__':
11     main()
```

Ausführbare Python-Datei vs. Bibliothek → **main**-Funktion.
Boilerplate unterscheidet: Main oder Bibliothek?

Auftreten von Fehlern

- ▶ Wann bemerkt man Fehler?

Auftreten von Fehlern

- ▶ Wann bemerkt man Fehler?
- ▶ Syntax-Fehler am Anfang

Auftreten von Fehlern

- ▶ Wann bemerkt man Fehler?
- ▶ Syntax-Fehler am Anfang
- ▶ Alle anderen bei der Ausführung!

Auftreten von Fehlern

- ▶ Wann bemerkt man Fehler?
- ▶ Syntax-Fehler am Anfang
- ▶ Alle anderen bei der Ausführung!
- ▶ Deswegen komplizierten Code Testen! (später)

Das Problem mit dem Scope

```
1 var = 12
2 def foo():
3     var = 9
4     print(f"    in foo: {var}")
5
6 #def main...
7 print(f" vor foo: {var}")
8 foo()
9 print(f"nach foo: {var}")
```

Das Problem mit dem Scope

```
1 var = 12
2 def foo():
3     var = 9
4     print(f"    in foo: {var}")
5
6 #def main...
7 print(f" vor foo: {var}")
8 foo()
9 print(f"nach foo: {var}")
```

Das Problem:

Variablen sind zwar nach innen sichtbar, werden aber innerhalb der Funktion neu angelegt.

global

```
1 var = 12
2 def foo():
3     global var
4     var = 9
5
6 #def main...
7 print(f"vor foo: {var}")
8 foo()
9 print(f"nach foo: {var}")
```

Das ist meist Gaffa, keine dauerhafte Lösung!
⇒ Variablen als Argumente übergeben

Parameter?

```
1 var = 12
2 def foo(variable):
3     variable = 9
4     print(f"    in foo: {variable}")
5
6 #def main...
7 print(f" vor foo: {var}")
8 foo(var)
9 print(f"nach foo: {var}")
```

„passing by value“ statt „by reference“ :(
zurückgeben \Rightarrow Funktionaler Ansatz

Funktionaler Ansatz

```
1 var = 12
2 def foo(variable):
3     variable = 9
4     print(f"  in foo: {variable}")
5     return variable
6
7 print(f" vor foo: {var}")
8 var = foo(var)
9 print(f"nach foo: {var}")
```

Alles übergeben und zurückgeben... Anstrengend
sinnvoll strukturieren \Rightarrow Objekte

Klassen und Attribute

- ▶ Klassen sind wie Schablonen

Klassen und Attribute

- ▶ Klassen sind wie Schablonen
- ▶ Objekte sind konkrete Instanzen

Klassen und Attribute

- ▶ Klassen sind wie Schablonen
- ▶ Objekte sind konkrete Instanzen

```
1 class Klasse:
2     def __init__(self, wert):
3         self.attribut = wert
4
5     def print_attribut(self):
6         print(self.attribut)
7
8 def main():
9     a = Klasse(3)
10    b = Klasse(12)
11    a.attribut = -a.attribut
12    b.print_attribut()
```

auf Deutsch: selbst festgelegt

Attribute

Attribute setzen: meist im **Initialisierer** `__init__`

```
1 class Mensch:
2     def __init__(self, vorname, nachname):
3         self.vorname = vorname
4         self.nachname = nachname
5     def vorstellen(self):
6         return f"Hi, ich bin {self.vorname} {self.nachname}"
7
8 def main():
9     # instanziiert zwei Objekte vom Typ 'Mensch'
10    maria = Mensch("Maria", "Stuhlbein")
11    john = Mensch("John", "Doe")
12    print(maria.vorstellen())
```

auf Deutsch: selbst festgelegt

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind
- ▶ erstes Argument `self` wird übergeben

Methoden

- ▶ Funktionen, die Teil von Klassen/Objekten sind
- ▶ erstes Argument `self` wird übergeben
- ▶ `def method(self, arg2): ...`
- ▶ `obj.method(arg2)`

Übung: Vektor-Klasse

```
1 class Vector:
2     """
3     Vector(x, y) -> an object with attributes x and y
4     vec.add(vec2) -> vec + vec2
5     vec.mul(factor) -> vec * factor
6     vec.abs() -> (x**2 + y**2) ** .5
7     """
8
9     def __init__(self, x, y):
10         self.x = x
11         ...
12
13     def add(self, vec2):
14         return ...
15
16     def mul(self, factor):
17         return ...
18
19     def abs(self):
20         return ...
```

Übung: Vektor-Klasse

```
1 class Vector:
2     """
3     Vector(x, y) -> an object with attributes x and y
4     vec.add(vec2) -> vec + vec2
5     vec.mul(factor) -> vec * factor
6     vec.abs() -> (x**2 + y**2) ** .5
7     """
8
9     def __init__(self, x, y):
10         self.x = x
11         self.y = y
12
13     def add(self, vec2):
14         return Vector(self.x + vec2.x, self.y + vec2.y)
15     def mul(self, factor):
16         return Vector(self.x * factor, self.y * factor)
17     def abs(self):
18         return (self.x ** 2 + self.y ** 2) ** .5
```

Beispiel: Mein Vektor

```
1 class Vector:
2     def __init__(self, x, y): <...>
3     def __add__( self, other):          #self + other
4         return Vector( <...> )
5     def __iadd__(self, other):          #self += other
6         self.x += other.x
7         self.y += other.y
8         return self
9     def __sub__( self, other): <...> #self - other
10    def __isub__(self, other): <...> #self -= other
11    def __mul__( self, factor): <...> #self * factor
12    def __imul__(self, factor): <...> #self *= factor
13    def __abs__( self): <...>         #abs(self)
14    def __iter__(self): <...>         #<...>
```

Super- und Subklassen

Erben / Erweitern: mit `class subclass(superclass):` definieren

- ▶ neue Variablen und Methoden hinzufügen, auch alte überschreiben
- ▶ Superklasse mit `super()` aufrufen

```
1 class Mensch():
2     def __init__(self, vorname, nachname):
3         self.vorname = vorname
4         self.nachname = nachname
5
6 class Kind(Mensch):
7     def __init__(self, vorname, nachname, eltern):
8         super(Kind, self).__init__(vorname, nachname)
9         self.eltern = eltern
```

Übung: Planet-Klasse

```
1 class Planet(...):
2     """
3     Planet(mass, Vector(x, y)) -> a mass at a position
4         vel: velocity (initially (0, 0))
5     planet.update(dt=.1) -> move self by vel * dt
6     planet.accel(acc, dt=.1) -> accelerate (dt: timestep)
7     planet.attract(other, dt=.1) -> accelerate planet
        towards other
8     some random bits of physics:
9         vel += acc * dt
10        pos += vel * dt
11        acc = force / mass
12        force = G * mass1 * mass2 / dist ** 2
13    """
14
15    def __init__(self, mass, pos):
16        super(Planet, self).__init__(pos.x, pos.y)
17        ...
18    def update(self): ...
19    def accel(self, acc): ...
20    def attract(self, other): ...
```

Übung: Planet-Klasse

```
1 from Vector import Vector
2 dt = .1 #time step per update
3 class Planet(Vector):
4     """
5     Planet(mass, Vector(x, y)) -> a mass at a position
6         vel: velocity (initially (0, 0))
7     planet.update() -> move (timestep: global dt)
8     planet.accel(acc) -> accelerate (change vel by acc)
9     planet.attract(other) -> accelerate planet towards other
10    """
11
12    def __init__(self, mass, pos):
13        super(Planet, self).__init__(pos.x, pos.y)
14        self.mass = mass
15
16    def update(self):
17        self += self.vel * dt #__iadd__, __mul__
18
19    def accel(self, acc):
20        self.vel += acc * dt #__iadd__, __mul__
```

Richtige Implementierung im Ordner ressourcen/planets/

Übung: Planet-Klasse (Fortsetzung)

```
1  def attract(self, other):  
2      diff = self - other #__sub__  
3      dist = diff.abs()  
4  
5      if dist > 0:  
6          force = self.mass * other.mass / dist ** 2  
7          factor = force / self.mass / dist  
8          directed_force = diff * factor #__mul__  
9          self.accel(directed_force)
```

Richtige Implementierung im Ordner ressourcen/planets/

Spezielle Methoden

von Python intern verwendet

`--init--` Bei Instanziierung aufgerufen

Spezielle Methoden

von Python intern verwendet

`--init--` Bei Instanziierung aufgerufen

`--del--` Bei Löschung aufgerufen (selten)

Spezielle Methoden

von Python intern verwendet

`--init--` Bei Instanziierung aufgerufen

`--del--` Bei Löschung aufgerufen (selten)

`--str--` Für Ausgabe als String

Spezielle Methoden

von Python intern verwendet

`--init--` Bei Instanziierung aufgerufen

`--del--` Bei Löschung aufgerufen (selten)

`--str--` Für Ausgabe als String

`--add--` Funktion für **+**-Operator

Spezielle Methoden

von Python intern verwendet

- `__init__` Bei Instanziierung aufgerufen
- `__del__` Bei Löschung aufgerufen (selten)
- `__str__` Für Ausgabe als String
- `__add__` Funktion für `+`-Operator
- `__iadd__` Funktion für `+=`-Operator (muss `self` zurückgeben)