Python Grundlagen

Julian Huber & Matthias Panny

Objektorientierung I

🎯 Lernziele

- Klassen und Objekte im Sinne der Objektorientierten Programmierung zu beschreiben
- Studierenden können mittels Implementierung von "Dunder-Methoden" Operatoren in Klassen überladen

Klassen und Objekte

Motivation

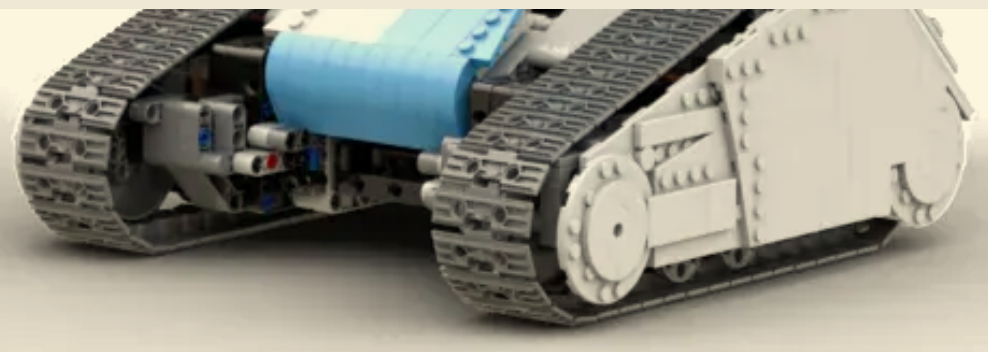
- Komplexe Software besteht auf hunderten Modulen, welche die Realität abbilden und miteinader über Schnittstellen interagieren müssen
- Gleichzeitig müssen mehrere Personen im Team an der Code-Basis entwickeln

Klassen und Objekte

Objektorientierte Programmierung

- Programmierparadigma mit Fokus auf Objekte, deren Eigenschaften und Fähigkeiten
- Am Beispiel eines autonomen Roboters
  - Attribute (Eigenschaften):
    - Identifikationsnummer
    - Maximale Geschwindigkeit
    - Abstand zur Wand
  - Methoden:
    - Abstand zur Wand ermitteln
    - Geschwindigkeit setzen
    - Kommunizieren

Bildquelle



Klassen und Objekte

Klassen

- Die Klasse Roboter ist eine abstrakte Beschreibung aller denkbaren Objekte vom Typ Roboter
- Eine Klasse beschreibt die Eigenschaften (Attribute) und die Tätigkeiten (Methoden), die ein Objekt der Klasse später besitzen kann
- *Analogie: Definition einer Funktion*

Objekte

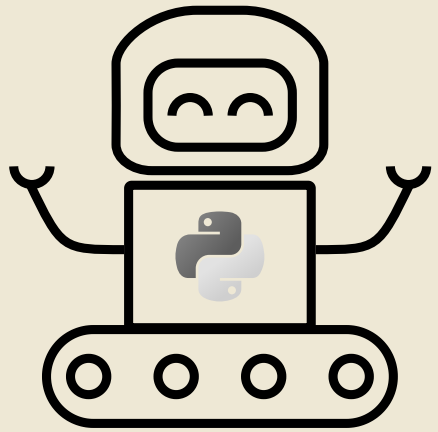
- Ein konkretes Objekt z.B. Nummer 5 ist eine Instanz (Ausprägung) der Klasse Roboter
- Objekte sind Instanzen (Ausprägungen) einer Klasse und hat bestimmte Werte für die Attribute
  - Identifikationsnummer: 5
  - Maximale Geschwindigkeit: $20\ \mathrm{km/h}$
- *Analogie: Aufruf einer Funktion mit bestimmten Parametern. Allerdings verbleibt das Objekt statt einer Rückgabe im Speicher*

Klassen und Objekte

Attribute und Methoden

Klasse
Robot

Methoden
- *communicate()*
- *calc_time_till_impact()*
- *communicate_with_time()*



Attribute
- *id_number*
- *max_speed_kmh*
- *dist_to_wall_cm*

Instanzen
number5, r2d2, etc.

- Attribute: Variablen mit elementaren Datentypen (`int`, `str`, `float`, etc.) oder Klassen → z.B.: `type(id_number) == int`
- Methoden: (kleine) Algorithmen implementiert im Sinne von Prozeduren und Funktionen
- Instanzen: Objekte, vom Typ der Klasse

Klassen und Objekte - Deklaration

- Eine Klasse ist die allgemeine Beschreibung einer Klasse von Objekten:

Deklaration einer Klasse

```python
# Python
class Robot():
  # Definition des Konstruktors = Beschreibung der Klasse
  def __init__(self, id_number, max_speed_kmh):
    # Genau diesem erstellen Objekt selbst
    # Weisen wir als Größe die übergebene Größe zu
    self.id_number = id_number
    self.max_speed_kmh = max_speed_kmh
```

- Die Methode `__init__()` ist der Konstruktor und wird aufgerufen, sobald ein Objekt einer Klasse instanziiert wird
- Jede Methode hat den Parameter `self` → referenziert auf das Objekt selbst und wird immer genutzt, wenn wir ein Attribut oder Methode genau dieses Objektes ansprechen wollen → wie der Zeiger `this` in C++
- Objekt im Beispiel hat die Attribute `id_number` und `max_speed_kmh`

Klassen und Objekte - Instanzierung

- Ein Objekt ist eine spezielle Ausprägung einer Klasse
- Um mit einer Klasse zu arbeiten werden Objekte der Klasse instanziiert

Instanzieren von Objekte

```
# Aufruf des Konstruktors = Instanziierung des Objekts
number5 = Robot(5, 20)
r2d2 = Robot(147, 32)
```

- Die Klasse `Robot` beschreibt allgemein das Verhalten jedes Roboters
- Hier werden zwei Objekte der Klasse `Robot` angelegt
- Die beiden Roboter sind durch die Menge ihrer Attribute vollständig beschrieben

Klassen und Objekte - Methoden

- Prozeduren oder Funktionen einer Klasse
- Syntax ist analog zur Definition von Funktionen → immer mit `self` als erstem Parameter

Methoden

```
class Robot():
# [...]

  def communicate(self):
    print(f"Hi, my ID is {self.id_number}")
    # [...]

# Instanziierung eines Roboters
number5 = number5 = Robot(5, 20)
# Aufruf einer Methode des Roboters number5
number5.communicate()
#> "Hi, my ID is 5"
```

Klassen und Objekte - Zugriff auf Methoden & Attribute

Attribute

- Objekte haben einen beliebigen Datentyp
- `<objekt_name>.<attribut_name>`

```
print(r2d2.max_speed_kmh)
# > 32
```

Methoden

- Aufruf: `<objekt_name>.<methoden_name>()`
- Methoden sind Funktionen, die zu einer Klasse gehören → über `self` Zugriff auf alle Attribute
- Können einen Rückgabewert ausgeben → oft keine Rückgabe sondern direktes ändern des Objektes selbst

```
class Robot():
# [...]
  def set_max_speed_kmh(self, max_speed_kmh):
    self.max_speed_kmh = max_speed_kmh
```

✍️ Aufgabe 1/2

- Gegeben ist die Folgenden Klassenbeschreibung für Roboter
- Leben Sie zwei unterschiedliche Objekte von Typ Roboter an

```python
# Python
class Robot():
  # Definition des Konstruktors = Beschreibung der Klasse
  def __init__(self, id_number, max_speed_kmh, dist_to_wall_cm):
    self.id_number = id_number
    self.max_speed_kmh = max_speed_kmh
    self.dist_to_wall_cm = dist_to_wall_cm
```

✍️ Aufgabe 2/2

- Legen Sie eine Methode `communicate()` an, in der sich der Roboter mittels `print()` Statement identifiziert (seine ID nennt)
- Legen Sie einer weitere Methode `calc_time_till_impact()` an, die die aktuelle Geschwindigkeit als Parameter übernimmt und die Zeit bis zum Impact errechnet. Wird keine Geschwindigkeit übergeben, so wird mit der Maximalgeschwindigkeit gerechnet
- Legen Sie eine neue Methode `communicate_with_time()` an, die auch die Zeit bis zum Einschlag berechnet. Rufen Sie hierin die `calc_time_till_impact()` Methode auf
- Legen Sie ein Objekt `nummer5` an und testen Sie die Methoden

Variablen

Attribute

- Eigenschaften des Objektes, welche wird während der Laufzeit speichern möchten, da wir später wieder darauf zugreifen möchten
- sind mit dem Objekt verknüpft
- z.B. `self.dist_to_wall_cm`

Hilfsvariablen

- Variablen, die wir einmalig während der Ausführung einer Methode brauchen
- Speicher wird nach Ausführung der Methode wieder freigegeben
- z.B. `max_speed_ms = self.max_speed_kmh / 3.6`

Versteckte Attribute

- Attribute, die dauerhaft erhalten bleiben, aber nicht von außen sichtbar sein sollen
  `self.__ip_address = "192.23.31.223"`

Kapselung - Beispiel

- Die IP-Adresse des Roboters soll von außen nicht sichtbar oder veränderbar sein → Attribut `__ip_address` und Methode `__print_ip_address()` nach außen versteckt
- in Python geschieht dies über zwei vorangestellte `_`
- innerhalb der Klasse selbst ist das Attribut sichtbar

Beispiel - `oop_robot.py`

- Erster Abschnitt im Beispiel

```
class Robot():
  def __init__(self, name, ip):
    self.name = name
    self.__ip = ip

  def print_robot(self):
    print(f"Created new robot {self.name} with ip {self.__ip}!")


robot_1 = Robot("Bob", "192.23.31.223")
robot_1.print_robot()
#> Created new robot Bob with ip 192.23.31.223!
robot_1.__ip
#> 'Robot' object has no attribute '__ip'
```

Kapselung - Getter & Setter

- sind Methoden, die den Zugriff auf gekapselte Attribute ermöglichen → wie in C++
- ermöglicht Aufteilung zwischen read-only (nur getter) and write-only (nur setter)
- in anderen Programmiersprachen sehr üblich

Beispiel - `oop_robot.py`

- Zweiter Abschnitt im Beispiel

```
class Robot():

  def __init__(self, name, ip):
    self.name = name
    self.__ip = ip
  # Getter
  def get_ip(self):
    return self.__ip
  # Setter
  def set_ip(self, new_ip):
    self.__ip = new_ip
    print("IP changed!")

robot_1 = Robot("Bob", "192.23.31.224")
robot_1.set_ip("192.1.1.1")
```

"Dunder Magic"

- "Dunder" steht für "Double Under" und bezeichnet spezielle Methoden/Variablen in Python → z.B.: `__name__`
- Dienen der Steuerung von Python selbst als Programmiersprache

Overloading

- Selbes Sprachelement (z.B.: `+`) wird kontext-abhängig unterschiedlich interpretiert
- Wir nutzen die gleiche Syntax für verschiedene Operationen mit verschiedenen Datentypen *unter der Haube*
  - `print("foo"+"ba")`: Strings werden verknüpft
  - `print(2+2)`: Integers werden addiert
- Kann mit Dunder-Methoden implementiert werden → z.B.: `__add__`

"Dunder Magic"

- Beispiel automatisches Umwandeln einer Klasse in einen String
- Standardmäßig wird ein Objekt als Speicheradresse dargestellt
  → siehe unten

Beispiel - `oop_robot.py`

- Dritter Abschnitt im Beispiel

```
class Robot():
  def __init__(self, name, ip):
    self.name = name
    self.__ip = ip
  # Getter
  def get_ip(self):
    return self.__ip
  # Setter
  def set_ip(self, new_ip):
    self.__ip = new_ip
    print("IP changed!")

robot_1 = Robot("Bob", "192.023.31.224")

# Per Default wird ein Object wenn es geprinted wird
# Zunächst als String dargestellt - z.B. mit Adresse
print(robot_1)
#> <__main__.Robot object at 0x000002015D0A6D40>
```

"Dunder Magic"

- Wir können die dabei aufgerufene Methode `__str__()` aber auch überschreiben:

Beispiel - `oop_robot.py`

- Vierter Abschnitt im Beispiel

```
class Robot():
  def __init__(self, name, ip):
    self.name = name
    self.__ip = ip

  def get_ip(self):
    return self.__ip

  def set_ip(self, new_ip):
    self.__ip = new_ip
    print("IP changed!")

  #Dunder-Methode
  def __str__(self):
    return F"I am {self.name} the robot!"

robot_1 = Robot("Bob", "192.023.31.224")

print(robot_1)
#> I am Bob the robot!
```

- Weitere Dunder-Methoden finden sich unter `dir(<class_name>)`

🏆 Hausaufgabe

- Schreiben Sie die aufgeführten Klassen
- Instanziieren Sie entsprechende Objekte und testen Sie die Datenverarbeitung an einem Beispiel
- Beantworten Sie die gestellten Fragen