Grundkonzepte der Objektorientierten Programmierung mit Python

Veit Berger

T³ DEUTSCHLAND

Teachers Teaching with Technology™

Einleitung

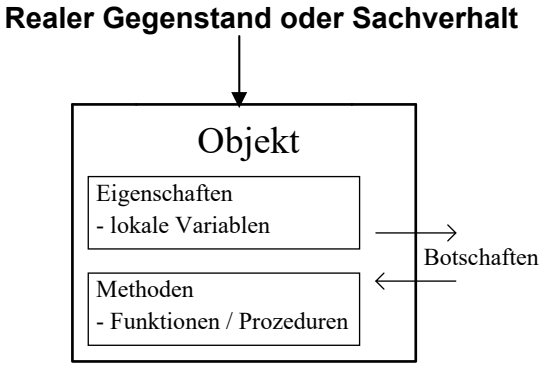
Mit der Einführung der neuen TI-Nspire-Taschenrechnergeneration steht ein sehr leistungsstarker Python-Interpreter zur Verfügung. Seine Implementierung entspricht dem MicroPython 1.11.0. Insbesondere für den anspruchsvollen Informatikunterricht ergibt sich damit die interessante Möglichkeit, die Grundkonzepte der Objektorientierten Programmierung (OOP)

- Kapselung
- Vererbung
- Polymorphie

einerseits zu vermitteln, andererseits aber auch in Form leistungsstarker Bibliotheken Schülerinnen und Schülern zu weiteren Problemlösungen anzubieten.
Der nachfolgende Beitrag soll diese beiden Aspekte in knapper Form thematisieren.

Objektorientierte Programmierung mit Python

Objekte sind in einer geeigneten Programmierumgebung modellierte Abbilder realer Gegenstände oder Sachverhalte:



- *Eigenschaften* repräsentieren den *Zustand* eines Objektes.
- *Methoden* beschreiben die Fähigkeit, empfangene Botschaften zu verarbeiten. Sie werden durch Funktionen und Prozeduren repräsentiert.
- Methoden und Botschaften haben im Allgemeinen die gleichen Bezeichner.

Mit Methoden kann das Objekt nach außen auf Botschaften reagieren oder seinen Zustand verändern. Dann werden die Werte der lokalen Variablen verändert.

Klassen und Instanzen:
Zu einem konkreten objektorientierten Modell gehören im Allgemeinen viele Objekte. Gleichgeartete Objekte werden zu einer *Klasse* zusammengefasst.
Ein einzelnes Objekt dieser Klasse heißt *Instanz*.

Beispiel:

Definition einer Klasse:

```
class person:
  def __init__(self, firstname, name):
    self.firstname = firstname
    self.name = name

  def set_firstname(self, firstname):
    self.firstname = firstname

  def get_firstname(self):
    return self.firstname

  def set_name(self, name):
    self.name = name

  def get_name(self):
    return self.name

  def say_name(self):
    print(self.name,",",self.firstname)

  def __del__(self):
    print('Entry deleted')
```

Konstrukor-Methode

Eigenschaften

Methoden

Destruktor-Methode

Instanz dieser Klasse:

```
from classes import *

alica = person('Alica', 'Newman')
print('Name:',alica.get_name())
alica.set_name('Miller')
alica.say_name()
del alica
```

Import der Klassendefintionen

Konstruktor: Instanziierung eines Objektes

Senden von Botschaften

Destruktor: Löschen des Objektes

Anmerkung: In MicroPython wird die Destruktor-Methode **del** nicht ausgeführt.

Kapselung:

- Die Eigenschaften sind streng *gekapselt*. Ihre Werte können ausschließlich durch das Objekt selbst, also durch seine Methoden verändert werden.
- Gleiche Botschaften können unterschiedliche Reaktionen auslösen, da das Objekt seinen Zustand verändert haben kann.

Anmerkung: In MicroPython kann nicht zwischen den Attributen **private** und **public** unterschieden werden. Dennoch sollte dieses Grundkonzept konsequent eingehalten werden.

Vererbung (inheritence):
Ableitung von Unterklassen, deren Instanzen spezialisiertere Eigenschaften und Methoden besitzen. Dabei unterscheidet man:

- **overtake:** SämtlicheEigenschaften und -methoden werden aus der Superklasse übernommen.
- **add:** Neue Elemente (Eigenschaften / Methoden) werden hinzugefügt.
- **modify:** Übernommene Elemente werden verändert.

Vererbungen erfolgen immer in der Klassenhierarchie, nicht von einem Objekt auf ein anderes!

Beispiel:

Die Tochterkasse **student** erbt alle Methoden und Eigenschaften aus der Elternklasse **person**. Die Klassen **person** und **student** bilden eine *Klassenhierarchie*.

```
class student(person):
  def __init__(self, firstname, name, grade):
    super().__init__(firstname, name)
    self.gade = grade

  def say_name(self):
    print('My name is', self.firstname, self.name,'.')

  def set_grade(self, grade):
    self.grade = grade

  def get_grade(self):
    return self.grade

  def say_grade(self):
    print('My grade of school is',self.grade,'.')
```

Vererbung aus der Klasse **person**

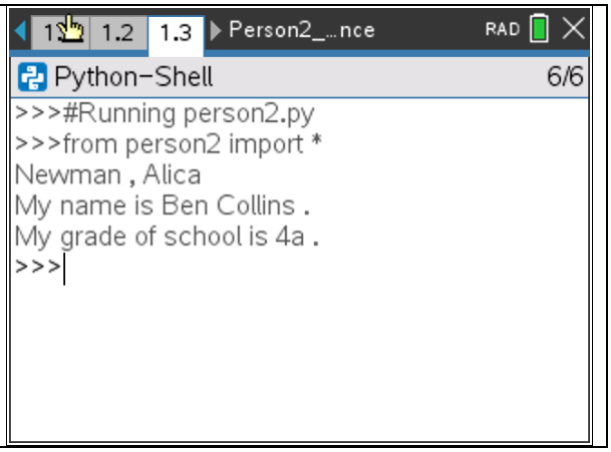
weitere Eigenschaft

Überschreiben einer Methode

weitere Methoden

In der Python-Shell werden nun die Klassendefinitionen importiert und getestet.

```
from classes import *

alica = person('Alica','Newman')
alica.say_name()
ben = student('Ben','Collins','3a')
ben.say_name()
ben.set_grade('4a')
ben.say_grade()
del alica
del ben
```

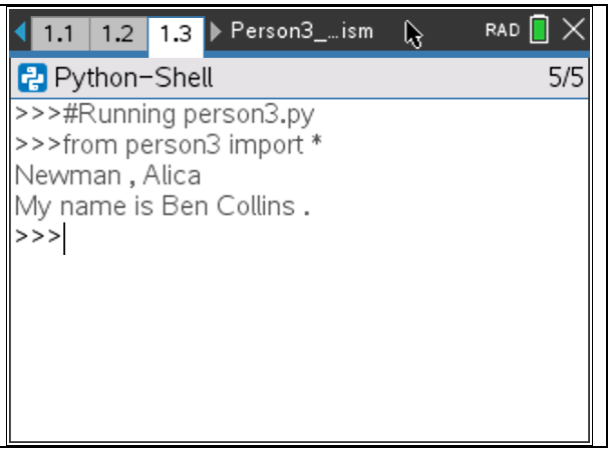


Polymorphie: „viele Formen besitzend“ (griech.)

- Innerhalb einer Klassenhierarchie können *Methoden mit gleichen Bezeichnern und gleicher Signatur bestehen, die auf gleiche Botschaften unterschiedlich reagieren.* Diese Methoden heißen *polymorph*.

Beispiel:
In der Klassenhierarchie **person** ist die Methode **say_name** polymorph, d.h. sie reagiert in jeder Klasse unterschiedlich. Damit können Prozeduren und Funktionen entwickelt werden, bei denen erst zur Laufzeit entschieden wird, welche Reaktionen ausgelöst werden (*späte Bindung*).
Im nachfolgenden Beispiel führt der Methodenaufruf **obj.say_name()** in der Definition **say_name(obj)** in Abhängigkeit des jeweiligen Instanz zu unterschiedlichen Bildschirmausgaben.

```
from classes import *

def say_name(obj):
  if isinstance(obj, person):
    obj.say_name()

alica = person('Alica','Newman')
ben = student('Ben','Collins','3a')
say_name(alica)
say_name(ben)
del alica
del ben
```



Ein geeignetes Unterrichtsbeispiel

Die Grundkonzepte der OOP lassen sich sehr anschaulich mit dem TI-RGB Array demonstrieren, das an dem TI-Innovator Hub angeschlossen ist.
Im nachfolgenden Beispiel werden zunächst alle Eigenschaften und Methoden der Standard-Klasse **rgb_array** auf die Unterklasse **my_rgb_array** vererbt.

```
from ti_hub import rgb_array
from time import sleep

class my_rgb_array(rgb_array):
  def __init__(self, color, *arg):
    super().__init__(*arg)
    self.state = 0
    self.color = color

  def on(self, t):
    if self.state == 0:
      if self.color == 'red':
        super().set_all(255, 0, 0)
      elif self.color =='green':
        super().set_all(0, 255, 0)
      elif self.color =='blue':
        super().set_all(0, 0, 255)
      else:
        super().set_all(255, 255, 255)
    sleep(t)
    self.state = 1

  def off(self):
    if self.state == 1:
      super().all_off()
      self.state = 0

  def get_state(self):
    return self.state

  def exit(self):
    self.off()
```

Übernahme eines optionalen Parameters

Aufruf einer Methode aus der Oberklasse

Ergänzung weiterer Methoden

Diese Klassenhierarchie wird nun durch eine weitere Unterklasse **my_alarm_array** erweitert, die ihrerseits Methoden und Eigenschaften aus der Oberklasse **my_rgb_array** erbt.

`class my_alarm_array(my_rgb_array):`
`  def __init__(self, *arg):`
`  super().__init__('red', *arg)`
`  self.dt = 0.2`
 	
`  def on(self, t):`
`  n = int(t / self.dt)`
`  for i in range(1, n):`
`  super().on(self.dt)`
`  super().off()`
`  sleep(self.dt)`
	Überschreibung der Methode aus der Oberklasse

Im nachfolgenden Programm kann man sich in sehr anschaulicher Weise von den unterschiedlichen Wirkungen auf dem TI-RGB Array überzeugen, die durch die polymorphen Methoden **on(2)** ausgelöst werden.

`from classes import *`
 	
`def switch(led):`
`  if isinstance(led, my_rgb_array):`
`  led.on(2)`
`  led.off()`
 	
`def main():`
`  led1 = my_rgb_array('green')`
`  switch(led1)`
`  led1.exit()`
`  c = input('Power on? (y/n) ')`
`  if c == 'y' or c =='Y':`
`  led2 = my_alarm_array('lamp')`
`  else:`
`  led2 = my_alarm_array()`
`  switch(led2)`
`  led2.exit()`
 	
`main()`
	Definition des Hauptprogramms
	Instanziierung des Objektes **led1**
	
	Instanziierung des Objektes **led2** unter der wahlweisen Einbeziehung des optionalen Parameters **'lamp'**
	
	Aufruf des Hauptprogramms

Nutzung vorgegebener Klassenbibliotheken

Die Entwicklung eigener Programme fällt Schülerinnen und Schülern oft schwer. Häufig beschränken sich deshalb Aufgabenstellungen auf recht einfache Problemlösungen, die wenig motivierend sind.
Eine bereitgestellte, leistungsstarke Klassenbibliothek könnte die Attraktivität des Problemlösens erhöhen. Die Idee besteht darin, Eigenschaften und Methoden einer Klasse genau zu beschreiben, damit deren Instanzen in schülereigenen, einfach strukturierten Programmen genutzt und zu anspruchsvollen Lösungen geführt werden können.

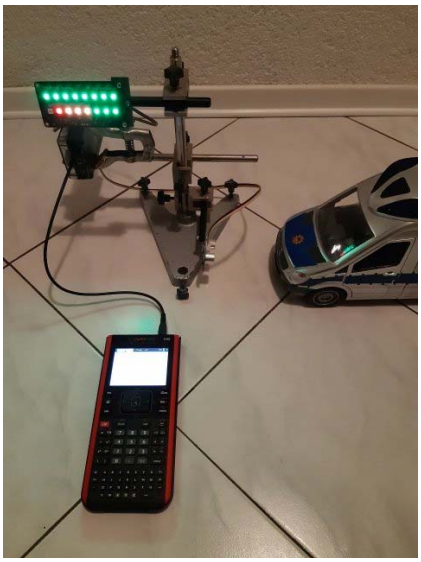
Im nachfolgenden Beispiel soll ein optischer Abstandswarner mit dem TI-RGB Array erstellt werden. Dazu wird den Schülerinnen und Schülern die abgeleitete Klasse **my-rgb-array** vorgegeben, die eine Quasi-Analoganzeige einer beliebigen Messgröße in einem beliebigen Intervall ermöglicht:

```
from ti_hub import rgb_array

class my_rgb_array(rgb_array):
  def __init__(self, min, max, *arg):
```

```
    try:
      super().__init__(*arg)
      self.device = True
    except:
      self.device = False
    self.val = 0
    self.min = min
    self.max = max

  def set_value(self, val):
    self.val = round((val - self.min) / (self.max - self.min) * 16)
    if self.val < 0: self.val = 0
    if self.val > 16: self.val = 16
    if self.device:
      for i in range(0, self.val):
        if i > 13: super().set(i, 255, 0, 0)
        elif i > 10: super().set(i, 255, 125, 0)
        else: super().set(i, 0, 255, 0)
      for i in range(self.val, 16):
        super().set(i, 0, 0, 0)

  def get_value(self):
    return self.val

  def get_info(self):
    if self.device:
      return('RGB-Array exists!')
    else:
      return('RGB-Array is not ready for use!')

  def exit(self):
    if self.device: super().all_off()
```

Nun ist die Problemlösung zum Abstandswarner in wenigen Programmzeilen möglich. Dabei kann die zu erfassende Distanz in weiten Grenzen verändert werden:

```
from classes import *
from ti_system import get_key
from ti_hub import ranger

def main():
  min = 0
  max = int(input("max. distance in cm: "))
  rgb = my_rgb_array(min, max)
  print (rgb.get_info())
  ran = ranger("IN 1")
  while get_key() != "esc":
    x = 100 * ran.measurement()
    #print("distance in cm: ",x)
    x_rgb = max - x
    rgb.set_value(x_rgb)
  rgb.exit()

main()
```



Zusammenfassung

Auf dem TI-Nspire CX II gestattet MicroPython die Vermittlung einfacher Programmstrukturen ebenso wie die Einführung der Grundlagen einer anspruchsvolle Objektorientierte Programmierung.
In diesem Spektrum sind vielfältige didaktische Einsatzmöglichkeiten denkbar:

- Einführung in die imperative Programmierung
- Vermittlung der Grundkonzepte der OOP
- Problemlösungen unter Einbeziehung leistungsfähiger Programmmodule
- Programmentwicklung in verschiedenen Schülerteams
- Entwicklung einer umfassenden Klassenbibliothek zur Realisierung anspruchsvoller Softwareprojekte

Nicht zuletzt bietet die Kommunikationsfähigkeit mit dem TI-Innovator Hub weitere umfassende Einsatzmöglichkeiten im MINT-Bereich.

Quellen:

[1] https://www.python-kurs.eu/python3_vererbung.php, zugegriffen am 18.10.2020
[2] Horn, Kerner: Lehr- und Übungsbuch INFORMATIK, Band 3: Praktische Informatik, Fachbuchverlag Leipzig im Carl Hanser Verlag, München, Wien 1997
[3] Wagenknecht: Programmierparadigmen - Eine Einführung auf der Grundlage mit Scheme, B. G. Teubner Verlag, Wiesbaden 2004

T³ DEUTSCHLAND

Teachers Teaching with Technology™