Gültigkeitsbreich von Variablen (Scope)

Scope

- Eine Variable kann an verschiedenen Stellen in einem Python-Skript deklariert werden
  - auf der globalen Ebene (außerhalb einer Funktion oder Schleife)
  - in einer `while`-Schleife
  - als Schleifenvariable einer `for`-Schleife
  - in einer Funktion
- Die Scoping-Regeln legen fest, von wo aus eine Variable gelesen und/oder verändert werden kann

Beispiel: Variablendeklaration an verschiedenen Stellen

```
def f():
    k=5 # in einer Funktion


for i in range(10): # i als Schleifenvariable
    print(i)

a=1 # Globale Variable
while a<5:
    b = a+1 # in einer while-Schleife deklariert
    a += 1

if a > 5:
    c = 0 # innerhalb eines ifs
else:
    c = 1 # innerhalb eines ifs
```

Scoping-Regeln in Python (1)

- Variablen, die innerhalb von `if`, `while` und `for` definiert werden, sind auch außerhalb lesbar

  ```
  for i in range(10):
      print(i)
  print(i) # 9
  ```

- Globale Variablen können innerhalb von Funktionen gelesen werden

  ```
  a=0
  def f():
  ```

```
    print(a)
f() # 0
```

Scoping-Regeln in Python (2)

- Variablen, die innerhalb von Funktionen definiert werden, sind außerhalb der Funktion nicht sichtbar

```
def f():
    a=1
print(a) # wirft eine Exception vom Typ NameError
```

- Wenn eine Funktion eine lokale Variable definiert, so dass es schon eine globale Variable mit dem selben Namen gibt, wird die globale Variable nicht überschrieben

```
def f():
    a=1
    print(a) # 1
a=0
f()
print(a) # immernoch 0
```

- Eine Funktion kann nicht eine Variable mit demselben Namen lokal und global nutzen

```
def f():
    print(a) # wirft eine Exception vom Typ UnboundLocalError
    a=1
a=0
f()
```

Globale Variablen aus einer Funktion heraus ändern oder definieren

- Warum ist das keine so gute Idee?
- In einer Funktion deklariert man mit dem Keyword `global` eine Variable als global

```
a = 0
def f():
    global a
    global b
    a = 1
    b = 2
print(a,b) # 1 2
```

Übung

Was geben folgende Skripte aus (oder wo wird ein Fehler geworfen)

1. ```
   def f():
       print(a)
   f()
   a=1
   ```
2. ```
   def f():
       a = 1
       global b
       b = 1

   a=0
   b=0
   f()
   print(a,b)
   ```

Objektorientierte Programmierung in Python

Objektorientierung als Programmierparadigma

> Die Grundidee besteht darin, die Architektur einer Software an den Grundstrukturen desjenigen Bereichs der Wirklichkeit auszurichten, der die gegebene Anwendung betrifft. Ein Modell dieser Strukturen wird in der Entwurfsphase aufgestellt. Es enthält Informationen über die auftretenden Objekte und deren Abstraktionen, ihre Typen.

(Wikipedia)

- Die Welt wird als eine Sammlung aus Objekten dargestellt
- Ein Objekt hat Eigenschaften (Attribute), und kann Handlungen ausführen (Methoden)
- Objekte, die sich nur in Ihren Daten unterscheiden, sind Instanzen derselben Klasse
- Klassen mit ähnlichen Eigenschaften oder Methoden können von der selben Basisklasse abgeleitet werden

Objekte: Beispiele

- Katze *Fauch*
  - Attribute: Name=Fauch, Farbe=getigert, Gewicht=4kg
  - Methoden: futtern, schlafen, schnurren, Mäuse fangen
- Plot von x^2

  - Attribute: Titel=Parabel, Größe=10cm, Funktion=x^2, ...
  - Methoden: anzeigen, speichern
- In Python ist alles ein Objekt

Klassen

- Klassen sind ein Bauplan für ein Objekt.
- Objekte, die auf Basis einer Klasse erzeugt werden, nennt man Instanzen der Klasse
- Instanzen einer Klasse haben die selben Attribute und Methoden, die Werte der Attribute unterscheiden sich aber

Beispiel:

- Klasse *Katze*
  - Attribute: Name, Farbe, Gewicht
  - Methoden: futtern, schlafen, schnurren, Mäuse fangen
- Instanzen von *Katze*:
  - Fauch, getigert, 4 kg
  - Stubentiger, weiß, 5 kg
  - Merlin, schwarz, 4 kg

Vererbung

- Eine Klasse kann von einer anderen Klasse abgeleitet werden
- Die abgeleitete Klasse
  - erbt alle Attribute und Methoden der Elternklasse
  - kann aber weitere Methoden und Attribute definieren
  - soll das selbe Verhalten wie die Elternklasse haben (die selben Invarianten)
- Ist ein Quadrat ein Spezialfall eines Rechtecks?
  - aus Sicht der Mathematik
  - aus Sicht der objektorientierten Programmierung

Liskov'sches Substitutionsprinzip

> Es besagt, dass ein Programm, das Objekte einer Basisklasse T verwendet, auch mit Objekten der davon abgeleiteten Klasse S korrekt funktionieren muss, ohne dabei das Programm zu verändern.

Wikipedia

- Benannt nach Barbara Liskov (Turing Award 2009)

- Beispiele:
  - Erfüllt: T=Tasteninstrument, S=Klavier, S=Keyboard, S=Cembalo (erfüllt)
  - Nicht erfüllt: T=Rechteck, S=Quadrat (Warum?)
  - Übung: T=Hai, S=Goldfisch

Klassen und Objekte in Python: Beispiel

```
class A: # definiert die Klasse A
    def __init__(self, value): # Konstructor
        self.value = value # setzt das Attribut value der Klasse

    def do_something(self): # andere Methode der Klasse
        print("Doing something. BTW, the value attribtue is", self.value)

# Wir erstellen zwei Instanzen der Klasse
a1 = A(2) # das Argument wird an die __init__ Methode gegeben
a2 = A("hallo")

a1.do_something()
a2.do_something()
print(a1.value, a2.value) # direkter Zugriff auf ein Attribut
```

Klassen in Python: Regeln

- Definition mit dem Keyword `class` gefolgt vom Klassenname
- Code der Klasse eingerückt
- Innerhalb der Klasse werden Methoden definiert
- Das erste Argument jeder Methode ist `self`. Es zeigt auf die jeweilige Instanz der Klasse
- Die Methode `__init__()` wird aufgerufen, wenn ein Objekt basierend auf der Klasse erstellt wird (konstruiert/instanziiert wird)
- Attribute und Methoden, deren Name mit `_` beginnt sind per Konvention privat. Aufruf sollte nur von innerhalb der Klasse erfolgen
- Empfehlung aus PEP8:
  - KlassenNamen in CamelCase
  - `methoden()` und `attribut_namen` klein und mit Unterstrich

Vererbung

```
class Kindklasse(Elternklasse):
    ...
```

- Methoden der Elternklasse können ersetzt werden.

- Aus einer Ersetzten Methode wird die Methode der Elternklasse mit `super().methoden_name()` aufgerufen

“‘python class A: # definiert die Klasse A def **init**(self, value): # Konstructor self.value = value # setzt das Attribut value der Klasse

class B(A): # definiert die Klasse B, die von A erbt def **init**(self, value): # ersetzt ___init aus A super().**init**(value) # ruft die **init**-Methode von A auf self.second_value = “whatever” # setzt weiteres Atttribut

- Instanziierung

```
b = B(17)
print(b.value, b.second_value) # value ist aus A geerbt,
                               # second_value in B definiert
```

Python: gute Praxis

Python: gute Praxis

- alle Konstanten am Anfang definieren (Werte, die für das ganze Programm gelten)
- eine einheitliche Konvention für Namen verwenden
- Finger weg von Wildcard-Imports wie `from numpy import *`
- Abhängigkeiten klar darstellen
  - Funktionen sollten nur ihre Argumente als Eingabe nutzen, möglichst keine globale Variablen
  - die Nutzung des `global` Keywords ist erst nach schriftlicher Begründung zulässig

Stärken und Schwächen von Python

Stärken und Schwächen von Python

- Stärken
  - leicht zu lernen
  - recht gut lesbarer Code
  - (unbürokratische) dynamische Typisierung
  - viele nützliche Module (NumPy, MatPlotLib, argparse), die leicht zu nutzen sind
- Schwächen
  - langsame Ausführung (im Vergleich zu JavaScript, C++)
  - viel Potenzial für Überraschung durch dynamische Typisierung
  - wenige Möglichkeiten, im Code Garantien zu geben oder einzufordern