Fortgeschrittene Programmierung

Kapitel 4: Python

Klassen und Objekte

Prof. Dr. Thomas Wieland

WS 2022/2023

Klassen in Python

- In Python ist eine Klasse eine **Gruppe von Datenelementen und Funktionen**. Klassen haben folgende Merkmale:
  - Konstruktoren und Destruktoren
  - Datenelemente (Attribute)
  - Elementfunktionen (Methoden)
  - Getter und Setter
- Deklaration mit Schlüsselwort `class`

```
class MyClass:
    def __init__(self): pass  # Konstruktor
    def __del__(self): pass   # Destruktor
```

- **Methoden** der Klassen haben als ersten Parameter stets `self`, um die Zugehörigkeit zu kennzeichnen
  - Wird beim Aufruf weggelassen

Objekte von Klassen

- Zugriff auf die Elemente **wie bei Modulen** mit Punkt '.'

```
class Krapfen :
    # ...

dein_krapfen = Krapfen()
dein_krapfen.fuellung = "senf";
```

- Zugriff auf Elemente des eigenen Objekts immer über self
  - `self.variable` oder `self.methode()`

Konstruktor und Destruktor

Konstruktor

- fester Name `__init__(self, ...)`
- Selbst optional, weitere Parameter optional
- Kann Speicher allokieren, Variablen initialisieren usw.
- Wird automatisch beim Erzeugen eines Objekts aufgerufen

Destruktor

- fester Name `__del__(self, ...)`
- Wird seltener implementiert
- kann Speicher freigeben, externe Verbindungen beenden, aufräumen usw.
- Zeitpunkt des Aufrufs nicht garantiert!

Zugriffsbeschränkungen

- **public:** Das Element unterliegt **keiner Zugriffsbeschränkung**, sie können daher beliebig aufgerufen bzw. modifiziert werden. Gilt standardmäßig für alle Attribute und Methoden.
- **private:** Das Element ist **ausschließlich innerhalb der Klasse** selbst zugreifbar, aber nicht in einer davon abgeleiteten Klasse. Dazu wird das Element mit zwei vorangestellten Unterstrichen gekennzeichnet.
- **intern:** Das Element **wird nicht exportiert**, ist bei Modulimporten also nicht sichtbar. Dazu wird das Element mit einem vorangestellten Unterstrich gekennzeichnet.

Statische Klassenelemente

- Attribute, die außerhalb einer Methoden deklariert sind, existieren **unabhängig von einer Instanz** der Klasse
  - Für alle Objekte einer Klasse nur einmal vorhanden
- **Zugriff**: *Klassenname.Elementname*
  - Z.B. `Kind.eisAmStiel = 0;`
- Es kann aber **auch über ein Objekt** zugegriffen werden!
- **Statische Methoden** müssen mit dem Dekorator `@staticmethod` gekennzeichnet sein, haben dann aber keinen „`self`“-Parameter, z.B.

```
@staticmethod
def gibEis(): Kind.eisAmStiel += 1
```

Getter und Setter

- **Zugriffe auf private Attribute** erfolgen über Methoden zum Lesen („Getter“) und Schreiben („Setter“)

```
def getAttribute1(self): return self.__attribute1
def setAttribute1(self, v): self.__attribute1 = v
```

- Der Zugriff kann mit **Dekorator** `@property` vereinfacht werden:

```
@property
def attribute1(self): return self.__attribute1
```

- Im Programm dann wie ein Attribut nutzbar!
- Für **kontrollierte Schreibzugriffe** mit Setter-Dekorator:

```
@attribute1.setter
def attribute1(self, v) : self.__attribute1 = v
```

Methoden zur Operatorüberladung (Hooks)

- `__getitem__(key)`: Erlaubt Lesezugriff mit []
- `__setitem__(key, value)`: Erlaubt Schreibzugriff mit []
- `__add__(value)`: Erlaubt Addition mit +
- `__str__()`: Wird von print verwendet
- `__gt__()`, `__lt__()`: Vergleich mit >,<
- ... und viele, viele mehr!

Vererbung

- Kennzeichnung als Unterklasse: bei Klassendeklaration hinter dem Klassennamen die **Oberklasse in Klammern**:
  `class Klassenname(Oberklassenname)`
- Beispiel:

```
class Chirurg(Arzt)          #Vererbung
  def __init__(self) :
      super().__init__()
      zulage = 1000.0;
      dienstjahre = 0;

  def patientBehandeln():  #Überschreiben
      #...
  def schneiden():         # Autsch!
      #...
```

- Alle Klassen erben von der Basisklasse `object`!
- Auch Mehrfachvererbung möglich, aber i.d.R. nicht ratsam

Ausnahmebehandlung

- Zur Behandlung von **vorhersehbaren Fehlern** (ähnlich wie in Java)

Ablauf:

- Eine Funktion **ruft** eine andere auf (`try`)
- Gibt es während der Abarbeitung des Aufrufs einen Fehler, wirft sie eine **Ausnahme** aus (`raise`)
- Die aufrufende Funktion kennt die Möglichkeit, dass eine solche Ausnahme auftreten könnte, und ist bereit, diese **abzufangen** (`except`).
- Optional ist ein `else`, das nur ausgeführt wird, wenn kein Fehler auftritt
- Zuletzt optional `finally`, das immer ausgeführt wird

- Wenn eine Funktion eine Exception auswirft, ist dies ein **alternativer Rücksprung**
  - Dort erzeugte Objekte werden gelöscht; es findet aber kein normaler Rücksprung mehr statt.

Ausnahmen

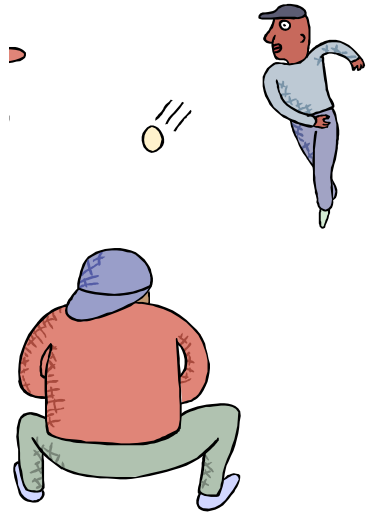
```
try :
    # beliebiger Code
except ValueError as e:
    # behandle ValueError
except Exception as e:
    # behandle andere Fehler
else:
    # ausführen, wenn kein Fehler auftrat
finally:
    # wird immer ausgeführt
```

Ausnahmen fangen

- Die aufrufende Funktion **umschließt den Aufruf** mit `try` und fügt anschließend ein oder mehrere `except`-Kommandos an, um die verschiedenen **Ausnahmen abzufangen**, die die Funktion auswerfen kann:

```
try :
    run_func();

except (Exc1, Exc2) :
    # Fehlerbehandlung
```



- Der jeweilige except-Block wird aufgerufen, wenn
  - `run_func()` **genau die Ausnahme** Exc1 (oder Exc2) geworfen hat
  - `run_func()` eine Ausnahme geworfen hat, die Exc1 (oder Exc2) als **Basisklasse** hat
- Eigene Exception-Klassen sollten von `Exception` ableiten

Zusammenfassung

- **Klassen** gruppieren Datenelemente und Funktionen
- Von ihnen werden **Objekte** erzeugt
- Im **Konstruktor** können die Datenelemente initialisiert werden
- Alle Elemente der Klasse sind **öffentlich**, außer sie werden entsprechend gekennzeichnet
- **Statische Klassenelemente** kommen für alle Objekte nur einmal vor
- Der Zugriff als Klassenelemente kann über **Getter-** und **Setter-Methoden** erfolgen
- **Operatoren** zwischen Objekten eigener Klassen können überladen werden
- Bei **Vererbung** übernimmt eine spezialisierte Klasse alle Elemente der Oberklasse und fügt ggf. eigene hinzu
- Zur Behandlung von vorhersehbaren Fehlern können **Ausnahmen** abgefangen werden