706.088 INFORMATIK 1

WIEDERHOLUNG

- Klassen
- Vererbung

KLASSEN

```
class Rectangle():
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def getX(self):
        return self._x
    def getY(self):
        return self._y
    def getArea(self):
        return self._x * self._y
```

VERERBUNG

```
class Square(Rectangle):
    def __init__(self,x):
        super().__init__(x, x)
    # alle Methoden und Eigenschaften der Basisklasse geerbt
    def getY(self):
        # print("there is no y")
        return self._x
```

KLASSENVARIABLEN

```
class Circle:
    # Eigenschaft wird von allen Instanzen geteilt
    pi = 3.1415926535897793
    def __init__(self,r):
      # Eigenschaft einer einzelnen Instanz
      self._r = r
    def getArea(self):
        return self._r ** 2 * Circle.pi
```

KLASSENVARIABLEN

```
class Circle:
    # wird von allen Instanzen geteilt
    pi = 3.141592653589793
    circle_counter = 0
    def __init__(self,r):
        self._r = r
        Circle.circle_counter += 1
    def getArea(self):
        return self._r ** 2 * Circle.pi
    def __del__(self):
        Circle.circle_counter -= 1
        if Circle.circle_counter == 0:
            print("no more circles left")
```

MODULE IN PYTHON

- helfen Programme aufzuteilen
- Programmcode kann wieder verwendet werden
- einzubinden über `import`
- externe Module/Bibliotheken können importiert werden

MODULE IN PYTHON

- Bsp:
  - `math`
    - `sin(),cos(),log()`
    - `pi,e`
  - `os`
    - `listdir(),name,uname()...`
  - `datetime`
  - `random`

PYPI

- Python Package Index
- pip3 install

EINBINDEN EIGENER MODULE

- Bsp:
  - Helfer Funktionen in `myfunctions.py`
  - `import myfunctions`
    - stellt Funktionen von `myfunctions.py` bereit

```
# myfunctions.py
def helper(test):
    return test*100
```

```
# myprogram.py
import myfunctions

print(myfunctions.helper(10))
```

EINBINDEN EIGENER MODULE

```
# myfunctions.py
def helper(test):
    return test*100
```

```
# myprogram.py
import myfunctions as mf

print(mf.helper(10))
```

MODULE ZU PAKETEN ZUSAMMENFASSEN

- Ordner mit Paketnamen
- Ein File für jedes Modul

```
mypackage/
  __init__.py # optional
  myfunctions.py
  module2.py
  module3.py
```

```
from mypackage import myfunctions
from mypackage import * # alles importieren, setzt __init__.py voraus
```

Siehe Doku

FEHLER-BEHANDLUNG IN PYTHON

3.1

FEHLERBEHANDLUNG IN PYTHON

› Ausnahmebehandlung, engl: **Exception Handling**

› Vereinfacht Fehlerbehandlung durch speziellen Mechanismus

› Rückgabewerte von Funktionen können für ordentlichen Programmablauf verwendet werden

› Fehler können strukturiert behandelt werden

EXCEPTION HANDLING

› Fehler 'wirft' eine *Exception* (Objekt) nach 'oben', Funktion ist beendet.

› Übergeordnete Funktion kann:

» fangen, fortfahren

» fangen, weiterwerfen, Funktion ist beendet

» lässt passieren, Funktion ist beendet

EXCEPTION OBJEKT

Enthält Attribute und Methoden (Funktionen) zur Klassifizierung des Fehlers

```
>>> e = Exception("My custom error")
>>> e.args
('My custom error',)
>>> e = Exception("My custom error","test", 1,2)
>>> e.args
('My custom error', 'test', 1, 2)
```

EXCEPTION WERFEN

```
>>> raise Exception("My Exception")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: My Exception
```

EXCEPTION WERFEN

Nur BaseException und davon Abgeleitete dürfen geworfen werden

```
>>> raise "test"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: exceptions must derive from BaseException
```

EINGEBAUTE EXCEPTIONS

- BaseException
- Exception (Basisklasse für Benutzer)
- SyntaxError
- NameError
- TypeError
- ImportError
- ...

EXCEPTION BEHANDLUNG

› *try* öffnet den Try-Block

› Exceptions aus dem Try-Block werden im Except-Block gefangen

› *except* definiert welche Exceptions behandelt werden

EXCEPTION FANGEN

```
>>> open("/tmp/non_existing_file",'r')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/non_ex
```

```
try:
    open("/tmp/non_existing_file")
except OSError as e:
    print("Caught:", e)
```

```
Caught: [Errno 2] No such file or directory: '/tmp/non_existing_file
```

TRY - EXCEPT - ELSE

```
try:
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")
```

EXCEPT - ELSE

```
try:
    print("all good")
    open("/tmp/non_existing_file")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
all good
Don't know this error!
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/tmp/non_ex
```

EXCEPT - ELSE

```
try:
    print(a) # undefined!
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
Undefined vars found
normal program flow
```

EXCEPT - ELSE

```
try:
    print("all good")
except NameError:
    print("Undefined vars found")
except:
    print("Don't know this error!")
    raise
else:
    print("everything is fine")

print("normal program flow")
```

```
all good
everything is fine
normal program flow
```

FINALLY

```
try:
  open("/tmp/non_existing_file",'r')
except FileNotFoundError:
  print("file does not exist")
except:
  print("don't know this error")
  raise
finally:
  print("cleaning up")
```

```
file does not exist
cleaning up
```

ASSERT

› Setzt Bedingung, die, wenn falsch, zu einer Exception führt.

› Nur zur Entwicklung sinnvoll.

› Nur mit __debug__== `True` aktiv

› Wird mit `python3 -O` deaktiviert (__debug__= `False`)

ASSERT

```
a = [1,2]
a[0] = 17
assert a == [17,2]
```

```
a[1] = a[1] + 3
assert a == [17,4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

FRAGEN?