Ostbayerische Technische Hochschule
Amberg-Weiden

KAPITEL 2: EINFÜHRUNG IN PYTHON (TEIL 2)

Beim letzten Mal:

- Arbeiten in der interaktiven Shell
- Variablen definieren und verwenden
- Einfache Datentypen: int, float, complex, bool
- Überblick zu sequentiellen Datentypen (z.B. str, list, tuple)
- Strings und String-Formatierung
- Listen und Erzeugung von Listen durch „list comprehension“
- Dictionaries
- Sets
- Verzweigungen (if-then-else)
- Schleifen (for, while)
- Flaches und tiefes Kopieren
- Funktionen
- Globale und lokale Variablen in Funktionen

Heute:

- Modularisierung
- Namensräume und Gültigkeitsbereiche
- Ausnahmebehandlung
- Objektorientierte Programmierung
- Dekorateure
- Iteratoren und Generatoren
- lambda, map, filter
- Funktionen mit beliebiger Anzahl an Argumenten
- Daten konservieren mit pickle

Ostbayerische Technische Hochschule
Amberg-Weiden

Modularisierung

Modularisierung

Ostbayerische Technische Hochschule Amberg-Weiden

Modularisierung:

- Ziel der modularen Programmierung ist es, Programme systematisch in logische Teilblöcke (Module) aufzuspalten.
- Die Aufteilung des Quelltexts in einzelne Teile bezeichnet man als Modularisierung.
- Ziele: Lesbarkeit, Zuverlässigkeit, einfache Wartung

Module in Python:

- Ein Modul ist in Python eine Datei, die Definitionen und Statements enthalten kann.
- Der Dateiname ist der Name des Moduls mit dem Suffix „.py“.
- Innerhalb eines Moduls ist der Name des Moduls in der globalen Variable „__name__“ verfügbar.

Module erstellen und verwenden

Ostbayerische Technische Hochschule Amberg-Weiden

- Schritt 1: Erstellen der Datei fibo.py mit den Funktionen fib und fib2:

fibo.py

```
def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

- Mit **dir()** kann man sich die in einem Modul definierten Namen ausgeben lassen:

```
>>> import fibo
>>> dir(fibo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'fib', 'fib2']
```

Module erstellen und verwenden

Ostbayerische Technische Hochschule Amberg-Weiden

- Schritt 2: Importieren des Moduls mittels „import“
- Über den Modulnamen kann nun auf die Funktionen zugegriffen werden:

```
>>> import fibo
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

- Möchte man auf die im Modul enthaltenen Funktionen ohne Präfix des Modulnamens zugreifen, kann man die Funktionen auch explizit in den globalen Namensraum importieren (mehr zu Namensräumen im Anschluss):

```
>>> from fibo import fib, fib2
>>> fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Module erstellen und verwenden

- Beim Import eines Moduls kann man mittels „import … as“ auch einen neuen Namen für den Namensraum wählen:

```
>>> import fibo as f
>>> f.fib(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>> f.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Alternativ kann man auch eine ganzes Modul in den globalen Namensraum einbinden, sodass der Zugriff auf Funktionen und Variablen ohne Präfix möglich ist. Dadurch werden allerdings ggf. bereits vorhandene gleichlautende Namen überschrieben:

```
>>> pi = 12.5
>>> from math import *
>>> pi
3.141592653589793
```

Suchpfad für Module

- Beim ersten Import eines Moduls wird der erzeugte Byte-Code im Ordner __pycache__ als .pyc-Datei abgelegt.
- Wird ein Modul importiert, z.B. „meinmodul“, sucht der Interpreter nach „meinmodul.py“ in der folgenden Reihenfolge:
  1. im aktuellen Verzeichnis
  2. im PYTHONPATH
  3. falls PYTHONPATH nicht gesetzt ist, im Default-Path.
- sys.path enthält alle Verzeichnisse, in denen Module gesucht werden:

```
>>> import sys
>>> for p in sys.path:
...     print(p)
...

C:\...\Anaconda3\python37.zip
C:\...\Anaconda3\DLLs
C:\...\Anaconda3\lib
C:\...\Anaconda3
C:\...\Anaconda3\lib\site-packages
C:\...\Anaconda3\lib\site-packages\win32
C:\...\Anaconda3\lib\site-packages\win32\lib
C:\...\Anaconda3\lib\site-packages\Pythonwin
```

Eigene Pakete erzeugen

- Mehrere Module können in Paketen zusammengefasst werden.
- Dazu erzeugt man einen Unterordner in einem Verzeichnis, in dem Python Module erwartet bzw. sucht.
- In diesem Ordner legt man eine Datei namens „__init.py__“ an, die leer sein kann oder Initialisierungscode enthält, der beim Einbinden des Pakets ausgeführt wird.
- Anschließend legt man die für das Paket bestimmten Module im Verzeichnis ab.

Verzeichnis „new_package“

__init__.py

```
```

a.py

```
def print_hallo():
        print("Hallo")
```

b.py

```
def print_welt():
        print("Welt")
```

Eigene Pakete verwenden

- Nun kann man das neue Paket wie folgt benutzen:

```
>>> from new_package import a,b
>>> a.print_hallo()
Hallo
>>> b.print_welt()
Welt
```

- Versucht man, das Paket direkt mittels „import new_package“ zu importieren und auf ein enthaltenes Modul zuzugreifen, erhält man eine Fehlermeldung. Module in einem Paket werden nicht automatisch importiert.

```
>>> import new_package
>>> new_package.a.print_hallo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'new_package' has no attribute 'a'
>>> new_package.a.print_hallo()
```

- Damit dies funktioniert, könnte man das Initialisierungsverhalten von __init__.py modifizieren und die obige Anweisung in __init__.py schreiben:

__init__.py

```
from new_package import a,b
```

Komplettes Paket importieren

Ostbayerische Technische Hochschule Amberg-Weiden

- Um ein komplettes Paket zu importieren, gibt es den Sternchen-Operator.
- Angenommen, die __init__.py des Pakets „new_package“ lautet wie folgt:

__init__.py

```
print("Initialisierung des Moduls new_package")
```

- Importiert man mittels „import *“ das Paket „new_package“, so wird lediglich die unten stehende Meldung ausgegeben.
- Die Module a und b werden jedoch, anders als man vermuten würde, nicht geladen:

```
>>> from new_package import *
Initialisierung des Moduls new_package
>>> a.print_hallo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Komplettes Paket importieren

Ostbayerische Technische Hochschule Amberg-Weiden

- Das automatische Nachladen kann erreicht werden, wenn der Autor eines Pakets einen Paketindex anlegt.
- Einen Paketindex erzeugt man, indem man in der __init__.py-Datei eine Liste mit dem Namen „__all__“ definiert.
- Die Liste „__all__“ ist die Liste aller Modulnamen, die importiert werden sollen, wenn ein Sternchen im Import verwendet wird.

__init__.py

```
print("Initialisierung des Moduls new_package")
__all__ = ["a", "b"]
```

- Wird nun mittels import * das Paket new_package importiert, so werden die Module a und b geladen:

```
>>> from new_package import *
Initialisierung des Moduls new_package
>>> a.print_hallo()
Hallo
>>> b.print_welt()
Welt
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Namensräume und Gültigkeitsbereiche

Namensräume

- Ein Namensraum ist eine Zuordnung von Namen zu Objekten.
- In Python gibt es folgende Namensräume:
  - die Namen der built-in-Funktionen
  - die globalen Namen eines Moduls
  - die lokalen Namen eines Funktionsaufrufs
- Verschiedene Namensräume existieren isoliert voneinander und können unterschiedliche Lebensdauer haben.
- Mit globals() kann man sich den Namensraum des Moduls ausgeben lassen, mit locals() den lokalen Namensraum:

```
>>> def fun(x,y):
...     z = x+y
...     print("Lokale Variablen in fun: ", locals())
...     print("Globale Variablen in fun: ", globals())
...     return z
>>> fun(1,2)
Lokale Variablen in fun:  {'x': 1, 'y': 2, 'z': 3}
Globale Variablen in fun:  {'__name__': '__main__', '__doc__': None,
'__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__spec__': None, '__annotations__': {},
'__builtins__': <module 'builtins' (built-in)>,
'fun': <function fun at 0x000001CF08E73730>}
```

Namensräume beim Importieren von Modulen

Ostbayerische Technische Hochschule Amberg-Weiden

Wir betrachten erneut das Beispiel des Imports des Moduls fibo:

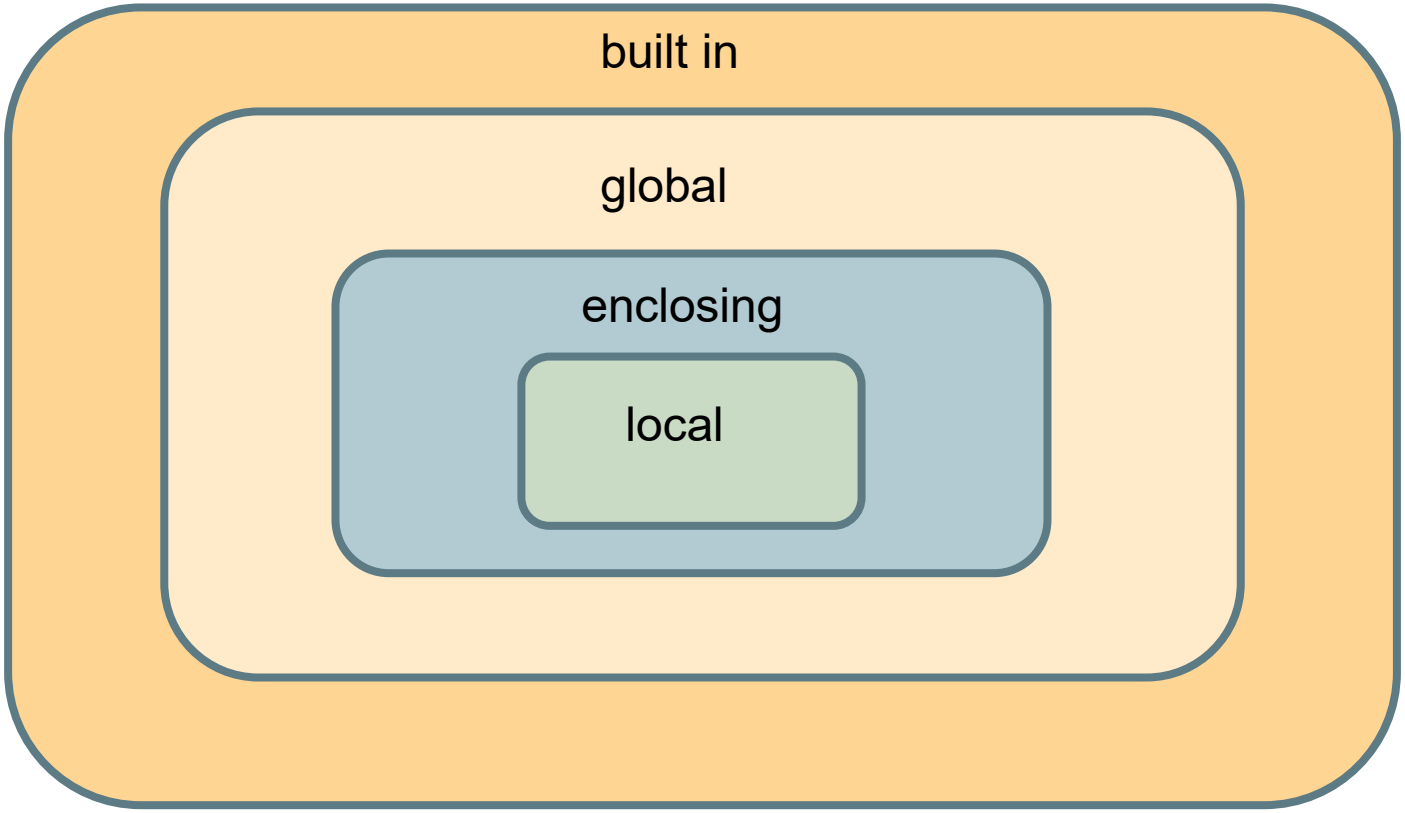
- Durch „import fibo“ wird das Modul fibo im globalen Namensraum bekannt gemacht. Der Zugriff auf die Funktionen erfolgt dann mittels fibo.fib() und fibo.fib2()
- Durch „from fibo import fib, fib2“ werden die Funktionsnamen direkt in den globalen Namensraum geholt.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, ... ,
 '__builtins__': <module 'builtins' (built-in)>}
>>> import fibo
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, ... ,
 'fibo': <module 'fibo' from 'C:\\xxx\\fibo.py'>}
>>> fibo.fib(10)
0 1 1 2 3 5 8
>>> from fibo import fib, fib2
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__': None, ... ,
 'fibo': <module 'fibo' from 'C:\\xxx\\fibo.py'>,
 'fib': <function fib at 0x000002604CE93620>,
 'fib2': <function fib2 at 0x000002604CE93598>}
>>> fib(10)
0 1 1 2 3 5 8
```

Gültigkeitsbereiche

- Ein Gültigkeitsbereich („scope“) ist eine Region eines Python-Programms, in der ein Namensraum direkt verfügbar ist, d.h. es einem unqualifizierten Namen möglich ist, einen Namen in diesem Namensraum zu finden.
- An einem beliebigen Zeitpunkt während der Ausführung kann es mehrere verschachtelte Gültigkeitsbereiche geben, deren Namensräume verfügbar sind:
  - Der innerste Gültigkeitsbereich, der zuerst durchsucht wird und die lokalen Namen enthält;
  - der Gültigkeitsbereich mit allen umgebenden Namensräumen (enthält auch die globalen Namen des momentanen Moduls), der vom nächsten umgebenden Namensraum aus durchsucht wird, und nicht-lokale, aber auch nicht-globale Namen enthält;
  - der vorletzte Gültigkeitsbereich enthält die globalen Namen des aktuellen Moduls;
  - der letzte Gültigkeitsbereich (zuletzt durchsuchte) ist der Namensraum, der die built-in-Namen enthält.

Verschachtelte Gültigkeitsbereiche



- Die **global**-Anweisung kann benutzt werden, um anzuzeigen, dass bestimmte Variablen im globalen Gültigkeitsbereich existieren und neu gebunden werden sollen.
- Die **nonlocal**-Anweisung zeigt an, dass eine bestimmte Variable im umgebenden Gültigkeitsbereich existiert und hier neu gebunden werden soll.

Gültigkeitsbereiche: Beispiel 1

Ostbayerische Technische Hochschule Amberg-Weiden

- Die Variable x ist jeweils lokal in den Funktionen f und h definiert.
- Die Veränderung des Werts in der Funktion h hat keine Auswirkung auf übergeordnete Gültigkeitsbereiche:

gueltigkeitsbereiche.py

```
def f():
    x = 1
    print("x vor Anwendung von h(): ", x)
    def h():
        x = 2
    h()
    print("x nach Anwendung von h(): ", x)
x = 0
f()
print("x nach Anwendung von f() ", x)
```

```
User>python gueltigkeitsbereiche.py
x vor Anwendung von h():  1
x nach Anwendung von h():  1
x nach Anwendung von f()  0
```

Gültigkeitsbereiche: Beispiel 2

Ostbayerische Technische Hochschule Amberg-Weiden

- Die Variable x in der Funktion h wird nun als „nonlocal“ definiert.
- Die Veränderung von x in h wirkt sich auf den übergeordneten Gültigkeitsbereich aus, aber nicht auf den globalen.

gueltigkeitsbereiche.py

```
def f():
    x = 1
    print("x vor Anwendung von h(): ", x)

    def h():
        nonlocal x
        x = 2
    h()
    print("x nach Anwendung von h(): ", x)

x = 0
f()
print("x nach Anwendung von f() ", x)
```

```
User>python gueltigkeitsbereiche.py
x vor Anwendung von h():  1
x nach Anwendung von h():  2
x nach Anwendung von f()  0
```

Gültigkeitsbereiche: Beispiel 3

- Die Variable x in der Funktion h als „global“ definiert.
- Die Veränderung von x in h wirkt sich auf den globalen Gültigkeitsbereich aus, aber nicht auf den übergeordneten.

gueltigkeitsbereiche.py

```
def f():
    x = 1
    print("x vor Anwendung von h(): ", x)

    def h():
        global x
        x = 2
    h()
    print("x nach Anwendung von h(): ", x)

x = 0
f()
print("x nach Anwendung von f() ", x)
```

```
User>python gueltigkeitsbereiche.py
x vor Anwendung von h():  1
x nach Anwendung von h():  1
x nach Anwendung von f()  2
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Ausnahmebehandlung

Ausnahmebehandlung mittels „try“ und „except“

Beispiel: Warten auf Eingabe eines Integers durch den Nutzer

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

Funktionsweise von „try“:

- Zuerst wird der try-Block ausgeführt.
- Tritt keine Ausnahme auf, wird der except-Block übersprungen und das Programm normal fortgesetzt.
- Tritt eine Ausnahme auf, wird der Rest des try-Blocks übersprungen. Falls der Typ der Ausnahme mit dem Schlüsselwort von except übereinstimmt, wird der except-Block ausgeführt und die Programmausführung wird nach dem try-Statement fortgesetzt.
- Tritt eine Ausnahme auf, die nicht mit dem Typ im except-Statement übereinstimmt, wird die Ausnahme ggf. an ggf. äußere try-Statements weitergereicht und kann dort behandelt werden.

Ausnahmebehandlung mittels „try“ und „except“

Ostbayerische Technische Hochschule Amberg-Weiden

Beispiel zur Weiterreichung einer Ausnahme:

exception_example.py

```
try:
    print("Jetzt wird gleich durch 0 dividiert...")
    try:
        print(1.0/0.0)

    except ValueError:
        print("Hier wird nicht reingelaufen")

except ZeroDivisionError:
    print("Division durch Null wird im aeusseren try abgefangen")
```

Ausgabe:

```
user>python beispiel_exceptions.py
Jetzt wird gleich durch 0 dividiert...
Division durch Null wird im aeusseren try abgefangen
```

Abfangen mehrerer Ausnahmen

- Der unspezifische except-Block wird ausgeführt, wenn eine Ausnahme auftritt, die keiner der angegebenen spezifischen Ausnahmen entspricht.
- Die Anweisung „raise“ führt die Ausnahme, die gerade abgefangen wurde, erneut aus.

```
import sys

try:
        f = open('integers.txt')
        s = f.readline()
        i = int(s.strip())
        print("1.0/i =", 1.0/i)

except IOError as err:
        (errno, strerror) = err.args
        print("I/O error({0}): {1}".format(errno, strerror))

except ValueError:
        print("Kein gueltiger Integer in der Zeile!")

except:
        print("Unerwarteter Fehler:", sys.exc_info()[0])
        raise
```

Finalisierungsaktion

- Neben except-Klauseln gibt es im Zusammenhang mit try auch noch eine finally-Klausel.
- Diese wird unter allen Umständen ausgeführt, egal ob eine Ausnahme aufgetreten ist oder nicht:

finally.py

```
try:
    x = float(input("Bitte gib eine Zahl ein: "))
    inverse = 1.0 / x
finally:
    print("Hier wird immer reingelaufen")

print("Hier wird nur reingelaufen, falls keine Ausnahme aufgetreten ist")
```

```
User>python finally.py
Bitte gib eine Zahl ein: 0
Hier wird immer reingelaufen
Traceback (most recent call last):
  File "finally.py", line 3, in <module>
    inverse = 1.0 / x
ZeroDivisionError: float division by zero
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Objektorientierte Programmierung

Objektorientierte Programmierung - Prinzipien

Ostbayerische Technische Hochschule Amberg-Weiden

Python als objektorientierte Sprache:

- Wir haben bisher bereits implizit Elemente der objektorientierten Programmierung kennengelernt.
- Wir hatten beispielsweise Objekte und Methode von Klassen benutzt, z.B. String-Methoden.
- In diesem Abschnitt wollen wir uns näher mit dem objektorientierten Ansatz in Python beschäftigen.

Wichtige Prinzipien der Objektorientierung:

- Datenabstraktion
- Datenkapselung
- Geheimnisprinzip
- Vererbung
- Polymorphismus

Klassendefinition, Klassen- und Instanzvariablen

Ostbayerische Technische Hochschule Amberg-Weiden

- Eine Klassendefinition in Python sieht wie folgt aus:

```
class Klassenname:
    anweisung1
    .
    .
    .
    anweisungN
```

- Eine Klassendefinition muss wie eine Funktionsdefinition ausgeführt werden, bevor sie Auswirkungen hat.
- Es wäre beispielsweise vorstellbar, eine Klassendefinition in einer if-Anweisung oder Funktion zu platzieren.
- Wird eine Klassendefinition betreten, wird ein neuer Namensraum erzeugt und als lokaler Gültigkeitsbereich benutzt.
- Wird eine Klassendefinition verlassen, wird ein Klassenobjekt erstellt.
- Der ursprüngliche Namensraum wird wieder betreten und das Klassenobjekt wird in ihm an den Namen, der in der Klassendefinition verwendet wurde, gebunden („Klassenname“ im obigen Beispiel).

Klassenobjekte

- Klassenobjekte unterstützen zwei Arten von Operationen: Attributreferenzierungen und Instanzierung.
- Attributreferenzierung benutzen die normale Notation **obj.name** zur Referenzierung des Attributs **name** des Objekts **obj**.
- Betrachten wir die folgende Beispielklasse:

```
class MyClass:
    """Eine einfache Beispielklasse"""
    i = 12345
    def f(self):
        return 'Hallo Welt'
```

- MyClass.i, MyClass.f und MyClass.__doc__ sind Referenzen auf die Klassenattribute i, f und __doc__

```
print(MyClass.i)
print(MyClass.f)
print(MyClass.__doc__)
```

```
12345
<function MyClass.f at 0x000001B1DD3136A8>
Eine einfache Beispielklasse
```

Instanzierung

- Die Klasseninstanzierung benutzt die Funktionsnotation und erzeugt zunächst ein leeres Objekt und weist dieses an einen Namen zu:

```
x = MyClass()
```

- Besitzt die Klasse eine Funktion namens __init()__, so wird diese bei der Instanzierung aufgerufen und kann einen bestimmten Anfangszustand herstellen:

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
>>> x.r = 2
>>> x.r, x.i
(2, -4.5)
```

Namespaces von Klassen

- Ein Klassenobjekt besitzt einen eigenen Namespace, der mit „__dict__“ angezeigt werden kann.
- Auch Instanzobjekte haben eigene Namespaces, die Werte von Instanzattributen enthalten:

```
>>> class Student:
...     def __init__(self, name, nr):
...             self.name = name
...             self.matrikelnr = nr
...     def print_name(self):
...             return self.name
...     def print_matrikelnr(self):
...             return self.matrikelnr
...
>>> Student.__dict__
mappingproxy({'__module__': '__main__', '__init__':
 <function Student.__init__ at 0x0000025BA3DD3400>, 'print_name':
 <function Student.print_name at 0x0000025BA3DC9488>, 'print_matrikelnr':
  <function Student.print_matrikelnr at 0x0000025BA3DC9598>, '__dict__':
  <attribute '__dict__' of 'Student' objects>,
 '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__': None})
>>> s = Student("Heinz", 123)
>>> s.__dict__
{'name': 'Heinz', 'matrikelnr': 123}
```

Methoden

- Neben Datenattributen können Instanzen Methodenattribute haben.
- Methoden können den Zustand des Objekts verändern.
- Bei der Methodendefinition steht das erste Attribut für die Klasse. Es ist Konvention, es mit „self“ zu benennen.
- Achtung: Datenattribute überschreiben Methodenattribute bei gleichem Namen.

```
>>> class Student:
...     def __init__(self, nr, studgang):
...             self.matrikelnr = nr
...             self.studiengang = studgang
...     def aendere_studiengang(self, studgang):
...             self.studiengang = studgang
...
>>> s = Student(123,"II")
>>> s.matrikelnr
123
>>> s.studiengang
'II'
>>> s.aendere_studiengang("MI")
>>> s.studiengang
'MI'
```

Methoden

Es ist nicht notwendig, dass die Funktionsdefinition innerhalb der Klassendefinition erfolgt. Die Zuweisung eines Funktionsobjekts an eine lokale Variable innerhalb der Klasse ist ebenfalls zulässig:

```
>>> def f1(self, x, y):
...     return min(x, x+y)
...
>>> class C:
...     f = f1
...     def g(self):
...             return "Hallo Welt"
...     h = g
...
>>> c = C()
>>> c.f(1,2)
1
>>> c.g()
'Hallo Welt'
>>> c.h()
'Hallo Welt'
```

Methoden

- Mittels „MethodType“ aus dem Modul „types“ kann man auch nachträglich eine Methode an eine Instanz binden.
- Frage: warum funktioniert im folgenden Beispiel der Aufruf c.f(2,4) im Anschluss an die Zuweisung c.f = f2 nicht?

```
>>> def f2(self, x, y):
...     return min(x, x-y)
...
>>> c.f = f2
>>> c.f(2,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f2() missing 1 required positional argument: 'y'
>>> from types import MethodType
>>> c.f = MethodType(f2,c)
>>> c.f(2,3)
-1
```

- Antwort: f2 ist eine Referenz auf eine Funktion, die drei Argumente erwartet. Durch die Zuweisung c.f = f2 wird das Attribut f in der Instanz c zu einer Funktionsreferenz auf eine Funktion, die drei Argumente erwartet.

Klassenattribute und Instanzattribute

Ostbayerische Technische Hochschule Amberg-Weiden

- Instanzattribute beziehen sich nur auf die jeweilige Instanz und können sich zwischen den Instanzen unterscheiden.
- Dagegen sind Klassenattribute für alle Instanzen der Klasse identisch.
- Im folgenden Beispiel ist „name“ ein Instanzattribut und „typ“ ein Klassenattribut.

```
>>> class Vierbeiner:
...     typ = "Hund"
...     def __init__(self, name):
...             self.name = name
...
>>> x = Vierbeiner("Bello")
>>> y = Vierbeiner("Waldi")
>>> x.name
'Bello'
>>> y.name
'Waldi'
>>> Vierbeiner.typ = "Schaeferhund"
>>> x.typ
'Schaeferhund'
>>> y.typ
'Schaeferhund'
```

Klassenmethoden

- Oben hatten wir bereits Beispiele für Methoden gesehen, die an Instanzen gebunden waren.
- Es ist auch möglich, Methoden zu definieren, die an Klassenobjekte gebunden sind. Diese können die Instanzvariablen nicht verändern. Dazu dekoriert man die Methode mit „@classmethod“ (mehr zu Dekorateuren später).
- Der Aufruf erfolgt über den Klassennamen oder über eine Instanz.

```
>>> class Student:
...         __counter = 0
...         def __init__(self):
...             Student.__counter += 1
...         @classmethod
...         def Anzahl(cls):
...             return Student.__counter
...
>>> x = Student()
>>> print(Student.Anzahl())
1
>>> y = Student()
>>> print(Student.Anzahl())
2
>>> print(x.Anzahl())
2
```

Statische Methoden

- Neben den Klassenmethoden, die an Klassen gebunden sind, und Methoden, die an Instanzen gebunden sind, gibt es noch die Möglichkeit, statische Methoden zu definieren, die weder an die Klasse noch an eine Instanz gebunden sind
- Dies wird erreicht, indem man die Funktion mittels „@static“ dekoriert.
- Die Methode kann sowohl über eine Instanz als auch über die Klasse aufgerufen werden:

```
>>> class Philosoph:
...     @staticmethod
...     def antwort():
...             return 42
...
>>> print(Philosoph.antwort())
42
>>> plato = Philosoph()
>>> print(plato.antwort())
42
```

Public-, Protected- und Private-Attribute

Ostbayerische Technische Hochschule Amberg-Weiden

Die Definition von Attributen als **public**, **protected** und **private** erfolgt über die Namensgebung:

Name		
name	Public	Attribute ohne führende Unterstriche sind sowohl innerhalb einer Klasse als auch von außen les- und schreibbar.
_name	Protected	Man kann zwar auch von außen lesend und schreibend zugreifen, aber der Entwickler macht damit klar, dass man diese Member nicht benutzen sollte. Protected-Attribute sind insbesondere bei der Vererbung wichtig.
__name	Private	Sind von außen nicht sichtbar und nicht benutzbar.

Public-, Protected- und Private-Attribute

```
>>> class A:
...     def __init__(self):
...         self.__priv = "Ich bin ein privates Datenattribut"
...         self._prot = "Ich bin ein protected Datenattribut"
...         self.pub = "Ich bin ein public Datenattribut"
...     def __priv_method(self):
...         print("Ich bin eine private Methode")
...     def _prot_method(self):
...         print("Ich bin eine protected Methode")
...     def pub_method(self):
...         print("Ich bin eine public Methode")
...
>>> a = A()
>>> a.__priv
!!!FEHLER!!!
>>> a._prot
'Ich bin ein protected Datenattribut'
>>> a.pub
'Ich bin ein public Datenattribut'
>>> a.__priv_method()
!!!FEHLER!!!
>>> a._prot_method()
Ich bin eine protected Methode
>>> a.pub_method()
Ich bin eine public Methode
```

Datenkapselung, getter- und setter-Methoden

Ostbayerische Technische Hochschule Amberg-Weiden

- Unter Datenkapselung versteht man den Schutz von Daten bzw. Attributen vor dem unmittelbaren Zugriff von außerhalb einer Klasse.
- Der Zugriff auf die Daten bzw. Attribute erfolgt meistens über entsprechende Methoden, die invariante Interfaces für die Klassenbenutzung darstellen:

```
>>> class Customer:
...     def __init__(self, age):
...         self.__age = age
...     def get_age(self):
...         return self.__age
...     def set_age(self, age):
...         self.__age = age
...
>>> x = Customer(24)
>>> x.get_age()
24
>>> x.set_age(30)
>>> x.get_age()
30
>>> x.__age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Customer' object has no attribute '__age'
```

Properties

- Mit denn sog „Properties“ bietet Python ein Sprachkonstrukt, das einen leichteren lesenden und schreibenden Zugriff auf private-Attribute ermöglicht.
- Für den Benutzer sieht es so aus als würde er ein public-Attribut verwenden.

```
>>> class Customer:
...     def __init__(self, age):
...             self.age = age
...     def __get_age(self):
...             return self.__age
...     def __set_age(self, age):
...             if age>=0:
...                     self.__age = age
...             else:
...                     self.__age = 0
...     age = property(fget=__get_age, fset=__set_age)
...
>>> c = Customer(25)
>>> c.age
25
>>> c.age = -5
>>> c.age
0
```

Properties mit Dekorateuren

Ostbayerische Technische Hochschule Amberg-Weiden

- Die Properties kann man syntaktisch auch mit Hilfe von Dekorateuren realisieren.
- Die getter- und setter-Methoden nennen wir „age“ statt „get_age“ bzw. „set_age“.
- Außerdem dekorieren wir die getter-Methode mit „@property“ und die setter-Methode mit „@age.setter“

```
>>> class Customer:
...     def __init__(self, age):
...         self.age = age
...     @property
...     def age(self):
...         return self.__age
...     @age.setter
...     def age(self, age):
...         self.__age = age
...
>>> c = Customer(23)
>>> c.age = 24
>>> c.age
24
```

Die __str__-Methode

- Wenden wir auf ein Objekt vom Typ Customer die **print**-Funktion an, so erhalten wir die folgende Ausgabe:

```
>>> c = Customer(23)
>>> print(c)
<__main__.Customer object at 0x0000017043CFA470>
```

- Intern hat **print** einfach **str**(c) aufgerufen. Die Ausgabe von **str**(c) kann man verändern, indem man in der Klasse die Methode „__str__“ implementiert:

```
>>> class Customer:
...     def __init__(self, age):
...             self.age = age
...     def __str__(self):
...             return str(self.age)
...
>>> c = Customer(36)
>>> print(c)
36
```

Vererbung

- Die Syntax für eine abgeleitete Klasse lautet wie folgt:

```
class AbgeleiteteKlasse(Basisklasse):
        statement1
        .
        .
        .
        statementN
```

- Bei der Erzeugung eines Klassenobjekts erinnert sich eine Klasse an ihre Basisklassen. Wird ein angefordertes Attribut nicht innerhalb der Klasse gefunden, so wird in der Basisklasse weitergesucht usw.
- Bei der Auflösung von Methodenreferenzen wird entlang der Vererbungskette (nach oben) nach einer Methode gesucht und die Methodenreferenz ist gültig, sofern ein Funktionsobjekt die Methode bereithält.
- Abgeleitete Klassen können Methoden ihrer Basisklassen überschreiben.

Vererbung

- Beim Anlegen von B wird die „__init__“ - Methode der Basisklasse A ausgeführt, da B keine eigene „__init__“ - Methode hat.
- Beim Aufruf der Methode „f“ für y wird die Methode „f“ von A aufgerufen, da B die Methode „f“ nicht implementiert.

```
>>> class A:
...     def __init__(self):
...         self.content = "42"
...         print("__init__ von A wurde aufgerufen")
...     def  f(self):
...         print("f von A wurde aufgerufen")
...
>>> class B(A):
...     pass
...
>>> x = A()
__init__ von A wurde aufgerufen
>>> x.f()
f von A wurde aufgerufen
>>> y = B()
__init__ von A wurde aufgerufen
>>> y.f()
f von A wurde aufgerufen
```

Überschreibung von Methoden

- Wir definieren nun in B eine eigene Methoden „__init__“ und „f“:

```
class A:
    def __init__(self):
        print("__init__ von A wurde aufgerufen")
    def  f(self):
        print("f von A wurde aufgerufen")
class B(A):
    def __init__(self):
            print("__init__ von B wurde aufgerufen")
    def f(self):
        print("f von B wurde aufgerufen")
```

- Beim Erzeugen einer Instanz von B wird nun die__init__-Methode von B aufgerufen. Außerdem wird nun die Methode „f“ von B statt der Methode „f“ von A verwendet:

```
>>> x = A()
__init__ von A wurde aufgerufen
>>> x.f()
f von A wurde aufgerufen
>>> y = B()
__init__ von B wurde aufgerufen
>>> y.f()
f von B wurde aufgerufen
```

Überladen von Methoden

Überladen von Funktionen wird in statisch getypten Sprachen wie Java und C++ benötigt, um die gleiche Funktion mit verschiedenen Typen zu definieren:

Beispiel 1: gleicher Name, gleiche Parameteranzahl, unterschiedliche Typen

Funktionsueberladung in C++

```
#include <iostream>
#include <cstdlib>
using namespace std;

int multi(int irgendwas, int faktor){
        return irgendwas * faktor;
}

double multi(double irgendwas, int faktor){
        return irgendwas * faktor;
}

int main(){
        cout << multi(10,3) << endl;
        cout << multi(10.3, 4) << endl;
        return 0;
}
```

Überladen von Methoden

Ostbayerische Technische Hochschule Amberg-Weiden

Parameterüberladung gibt es in C++ auch in der Form, dass die gleiche Funktion oder Methode mehrfach definiert wird, jedoch mit einer unterschiedlichen Zahl von Parametern.

Beispiel 2: gleicher Name, aber unterschiedliche Parameteranzahl

Funktionsueberladung in C++

```
#include <iostream>
using namespace std;

int f(int n){
        return n+42;
}
int f(int n, int m){
        return n+m+42;
}
int main(){
        cout << "f(3): " << f(3) << endl;
        cout << "f(3, 4): " << f(3,4) << endl;
}
```

Überladen von Methoden

- Methoden und Funktionen in Python haben bereits eine implizite Typpolymorphie wegen des dynamischen Typkonzepts von Python.
- Das zweite Beispiel (gleicher Name bei unterschiedlicher Parameteranzahl) lässt sich in Python nicht direkt implementieren.

```
>>> def f(n):
...     return n+42
...
>>> def f(n):
...     return n+42
...
... def f(n,m):
...     return n+m+42
...
>>> f(3,4)
49
>>> f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() missing 1 required positional argument: 'm'
```

Überladen von Methoden

- Man kann das polymorphe Verhalten jedoch mit Default-Parametern realisieren.
- Man hat aber streng genommen nur eine Funktionsdefinition und kann daher nicht von Polymorphie sprechen.

```
>>> def f(n, m=None):
...     if m:
...             return n+m+42
...     else:
...             return n+42
```

- Ganz allgemein lassen sich Polymorphien in der Parameteranzahl in Python mit einem *-Argument für eine beliebige Anzahl von Parametern definieren (mehr dazu später):

```
>>> def f(*x):
...     if len(x) == 1:
...             return x[0] + 42
...     else:
...             return x[0] + x[1] + 42
...
>>> f(3,4)
49
```

Überladen von Operatoren

- Wir hatten bereits gesehen, dass der Operator „+“ sowohl für die Klasse „int“ als auch für die Klasse „str“ definiert ist:

```
>>> 1+1
2
>>> "Hallo "+"Welt"
'Hallo Welt'
```

- Python erlaubt es, für eigene Klassen den „+“-Operator zu überladen.
- Um dies zu tun, muss die Methode __add__ implementiert werden.
- Für verschiedene Operatoren gibt es jeweils eigene, sog. „magische“ Methoden, die beim Aufruf des Operators auf eine Klasse aufgerufen wird. Diese muss man implementieren, um den jeweiligen Operator zu überladen (s. Dokumentation).

Überladen von Operatoren: „magische“ Methoden für Operatoren

Operator	Methode
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Bemerkung: es gibt analog magische Methoden für Zuweisungsoperatoren, unäre Operatoren und Vergleichsoperatoren.

Überladen von Operatoren

Beispiel einer eigenen Vektor-Klasse mit „+“-Operator:

```
>>> class Vec:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     def __add__(self, other):
...         return Vec(self.x+other.x, self.y+other.y)
...     def __str__(self):
...             return "("+str(self.x)+","+str(self.y)+")"
...     def __repr__(self):
...             return "("+str(self.x)+","+str(self.y)+")"
...
>>> x = Vec(1,2)
>>> y = Vec(3,4)
>>> x+y
(4,6)
```

Dekorateure

Was sind Dekorateure?

Ostbayerische Technische Hochschule Amberg-Weiden

- Ein Dekorateur in Python ist ein beliebiges aufrufbares Objekt, welches zur Modifikation einer Funktion oder einer Klasse genutzt wird.
- Eine Referenz zur Funktion „func“ oder zur Klasse „C“ wird an den Dekorateur übergeben, und dieser liefert eine modifizierte Funktion oder Klasse zurück.
- Die modifizierten Funktionen oder Klassen rufen üblicherweise intern die Originalfunktion „func“ oder die Original-Klasse „C“ auf.
- Im folgenden Abschnitt wollen wir uns mit Funktions-Dekorateuren beschäftigen.
- Klassen-Dekorateure werden nicht behandelt.

Verschachtelte Funktionen

Ostbayerische Technische Hochschule Amberg-Weiden

- Wir haben bereits die Möglichkeit kennengelernt, Funktionen zu verschachteln.

```
def f():
    def g():
        print("Hallo, ich bin die Funktion g")

    print("Dies ist die Funktion 'f'.")
    print("'f' ruft nun 'g' auf!")
    g()
```

- Führt man den obigen Code aus, erhält man folgende Ausgabe:

```
>>> f()
Dies ist die Funktion 'f'.
'f' ruft nun 'g' auf!
Hallo, ich bin die Funktion g
```

Funktionen als Parameter

- Jeder Parameter einer Funktion ist in Python eine Referenz auf ein Objekt.
- Funktionen sind ebenfalls Objekte. Somit können wir Funktionen, oder besser gesagt „Referenzen auf Funktionen“, ebenfalls als Argumente an Funktionen übergeben.

```
def g():
    print("Hallo, ich bin es, 'g'.")

def h():
    print("Hallo, ich bin es, 'h'.")

def f(func):
    print("'f' ruft jetzt gleich " + func.__name__ + " auf.")
    func()
```

```
>>> f(g)
'f' ruft jetzt gleich g auf.
Hallo, ich bin es, 'g'.
>>> f(h)
'f' ruft jetzt gleich h auf.
Hallo, ich bin es, 'h'.
```

Funktionen als Rückgabewert

- Die Rückgabe einer Funktion ist in Python eine Referenz auf ein Objekt.
- Somit können auch Referenzen auf Funktionen zurückgegeben werden.
- Beispiel:

```
def f(x):
    def h(y):
        return x+y+1
    return h
```

- Durch f(1) und f(10) werden zwei Funktionsreferenzen erzeugt und an fun1 bzw. fun2 zugewiesen. Anschließend kann man fun1 und fun2 wie gewohnt aufrufen:

```
>>> fun1 = f(1)
>>> fun1(5)
7
>>> fun2 = f(10)
>>> fun2(5)
16
```

Ein einfacher Dekorateur

Ostbayerische Technische Hochschule Amberg-Weiden

- Wir sind nun in der Lage, einen einfachen Dekorateur zu definieren.
- Dieser erhält als Argument eine Funktionsreferenz auf eine Funktion namens „fun“ und liefert eine Funktionsreferenz auf eine neue Funktion, die vor und nach der Ausführung von „fun“ einen Text ausgibt.
- Zuletzt wird die Funktion „sag_hallo“ mit unserem Dekorateur „dekoriert“:

```
def mein_decorator(fun):
    def wrapper():
            print("Textausgabe vor Funktionsaufruf")
            fun()
            print("Textausgabe nach Funktionsaufruf")
    return wrapper

def sag_hallo():
    print("Hallo")

sag_hallo = mein_decorator(sag_hallo)
```

```
>>> sag_hallo()
Textausgabe vor Funktionsaufruf
Hallo
Textausgabe nach Funktionsaufruf
```

@-Syntax für Dekorateure

Ostbayerische Technische Hochschule Amberg-Weiden

- Die Dekoration in Python wird üblicherweise mit der „@-Syntax“ vorgenommen.
- Anstatt

```
def sag_hallo():
        print("Hallo")

sag_hallo = mein_decorator(sag_hallo)
```

verwendet man die folgende Schreibweise:

```
@mein_decorator
def sag_hallo():
        print("Hallo")
```

Bemerkung: Die „@-Syntax“ ist nicht für Funktionen, die von Dritten geschrieben wurden und die aus einem Modul importiert wurden, verwendbar.

Beispiel für die Anwendung von Dekorateuren

Ostbayerische Technische Hochschule Amberg-Weiden

- Im folgenden Beispiel zeigen wir, wie unter Benutzung eines Dokorateurs elegant die Aufrufe einer Funktion gezählt werden können.

```
def call_counter(func):
    def helper(x):
        helper.calls += 1
        return func(x)
    helper.calls = 0
    return helper

@call_counter
def succ(x):
    return x + 1
```

- Ausgabe:

```
>>> print(succ.calls)
0
>>> for i in range(10):
...     succ(i)
...
>>> print(succ.calls)
10
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Generatoren und Iteratoren

Generatoren

- Unter einem Generator versteht man in der Informatik Programmcode, der dazu benutzt wird, das Iterationsverhalten einer Schleife zu kontrollieren.
- Sie verhalten sich wie eine Funktion mit Parametern, die bei einem Aufruf eine Folge von Werten zurückliefert, über die iteriert werden kann.
- Die Folge von Werten wird nicht auf einmal erzeugt wie beispielsweise bei einer Liste, sondern ein Wert nach dem anderen.
- Ein Generator ist also eine Funktion, die sich wie ein Iterator verhält.
- Vorteile:
  - Man spart Speicher, da nicht alle Werte auf einmal vorgehalten werden müssen.
  - Man muss nicht warten, bis die gesamte Folge von Werten erzeugt wurde, sondern kann direkt mit einer Scheife beginnen.

Iteratoren

- Iteratoren haben wir bereits in for-Schleifen kennengelernt. Im folgenden durchlaufen wir ein Element der Liste „cities“:

```
>>> cities = ["Regensburg", "Amberg", "Weiden", "Neumarkt"]
>>> for city in cities:
...     print("Stadt: "+city)
...
Stadt: Regensburg
Stadt: Amberg
Stadt: Weiden
Stadt: Neumarkt
```

- Bevor die Schleife gestartet wird, ruft Python intern die Funktion „iter“ mit der Liste „cities“ als Argument auf. Diese liefert ein Objekt zurück, mit dem es möglich ist, die Elemente der Liste zu durchlaufen.
- Der Rückgabewert von iter() ist ein Element der Klasse „list_iterator“.
- iter() ruft die Methode __iter__ der list-Klasse auf.
- Weitere Beispiele:
  - In der letzten Vorlesung hatten wir bereits das zip-Objekt als Iterator kennengelernt.
  - Die Funktion range() liefert ein range-Objekt, das ein iterierbares Objekt ist, allerdings kein Iterator.

Beispiel zu Iteratoren

- Nach jedem Schleifendurchlauf wird die **next()**-Funktion auf den list-Iterator angewendet, um das nächste Element zu erhalten.
- Dies geschieht so lange, bis der Vorrat an Listenelementen aufgebraucht ist
- Man kann diese Iteration auch „manuell“ nachvollziehen:

```
>>> l = [12,13,14,15]
>>> iter_x = iter(l)
>>> type(iter_x)
<class 'list_iterator'>
>>> next(iter_x)
12
>>> next(iter_x)
13
>>> next(iter_x)
14
>>> next(iter_x)
15
>>> next(iter_x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iteratoren

- Die sequentiellen Basistypen sowie ein Großteil der Klassen der Standardbibliothek von Python unterstützen Iterationen.
- Auch der Datentyp Dictionary (dict) unterstüzt die Iteration.
- In diesem Fall durchläuft die Iteration die Schlüssel der Dictionaries:

```
telefonnummern = {"Brunner" : 3636, "Heckmann" : 3612 , "Winter" : 3701}

for elem in telefonnummern:
    print(elem + ": " + str(telefonnummern[elem]))
```

- Die Ausgabe lautet:

```
Brunner: 3636
Heckmann: 3612
Winter: 3701
```

Generatoren und ihre Arbeitsweise

Ostbayerische Technische Hochschule Amberg-Weiden

- Eine einfache Möglichkeit, Iteratoren zu erzeugen, sind die sog. „Generatoren“ bzw. „Generator-Funktionen“.
- Ein Generator wird wie eine Funktion aufgerufen. Er liefert als Rückgabewert ein Iterator-Objekt zurück. Dabei wird der Code des Generators noch nicht ausgeführt.
- Wird nun der zurückgelieferte Iterator verwendet, wird jedesmal, wenn ein neues Objekt des Iterators benötigt wird (d.h. beim Aufruf der next-Methode), der Code innerhalb des Generators so lange ausgeführt, bis er bei einer **yield**-Anweisung angelangt.
- **yield** wirkt dann wie ein **return** einer Funktion, d.h. der Wert des Ausdrucks oder Objektes hinter **yield** wird zurückgegeben. Allerdings wird der Generator dabei nicht wie eine Funktion beendet, sondern er wird nur unterbrochen wartet nun auf den nächsten Aufruf, um hinter dem **yield** weiterzumachen. Sein gesamter Zustand wird bis zum nächsten Aufruf zwischengespeichert.
- Der Generator wird erst beendet, wenn entweder der Funktionskörper vollständig abgearbeitet wurde oder der Programmablauf auf eine **return**-Anweisung (ohne Wert) stößt.

Beispiel zu Generatoren

- Der city_generator() liefert nacheinander die Städtenamen zurück.
- Nachdem „Neumarkt“ zurückgeliefert wurde, ist der Vorrat an Städten in „cities“ aufgebraucht und der Generator erreicht beim nächsten Aufruf das Ende der Funktion, ohne auf „yield“ zu treffen.
- Die next-Funktion liefert beim nächsten Aufruf eine StopIteration-Exception, sodass die for-Schleife über den erzeugten Iterator endet.

```
>>> def city_generator():
...         cities = ["Regensburg", "Amberg", "Weiden", "Neumarkt"]
...         for city in cities:
...             yield city
...
>>> for city in city_generator():
...     print(city)
...
Regensburg
Amberg
Weiden
Neumarkt
```

Beispiel zu Generatoren

- Man kann auch Generatoren erzeugen, die beliebige Vorräte an Iterierten liefern. Allerdings birgt dies die Gefahr von Endlosschleifen.
- Alternativ kann man dies beim Design der Generator-Funktion berücksichtigen, z.B. durch Angabe einer maximalen Anzahl an zu erzeugenden Iterierten.

```
>>> def city_generator():
...         cities = ["Regensburg", "Amberg", "Weiden", "Neumarkt"]
...         while True:
...             city = cities.pop(0)
...             yield city
...             cities.append(city)
...
>>> cnt = 0
>>> for city in city_generator():
...     print(city, end=" ")
...     cnt += 1
...     if cnt == 10:
...             break
...
Regensburg Amberg Weiden Neumarkt Regensburg Amberg Weiden Neumarkt
 Regensburg Amberg >>>
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Lambda-Funktionen, map und filter

Lambda-Funktionen

Ostbayerische Technische Hochschule Amberg-Weiden

- Anonyme Funktionen oder Lambda-Funktionen sind Funktionen, die keinen Namen besitzen.
- Solche Funktionen können daher nur über Verweise angesprochen werden.
- Die Definition erfolgt mit Hilfe des lambda-Operators.
- Lambda-Funktionen können eine beliebige Anzahl von Parametern haben, führen einen Ausdruck aus und liefern den Wert dieses Ausdrucks als Rückgabewert zurück.
- Anonyme Funktionen sind vor allem nützlich, wenn man Funktionen aufruft, die andere Funktionen als Argumente beinhalten.

```
>>> lambda x: x+42
<function <lambda> at 0x000001653427368A8>
>>> (lambda x: x+42)(3)
45
```

Lambda-Funktionen

- In der Übung war die Aufgabe, eine Liste von 2-Tupeln nach den hinteren Elementen zu sortieren.
- Dazu dient die Funktion „sort“, die eine key-Funktion entgegennehmen kann und die Eingabewerte nach den Werten der key-Funktion sortiert.
- Dies lässt sich wie folgt lösen:

```
>>> l = [("a", 1), ("c", -1), ("b", -2)]
>>> def g(x):
...     return x[1]
...
>>> l.sort(key=g)
>>> l
[('b', -2), ('c', -1), ('a', 1)]
```

- Die Funktion g ist allerdings eine „Wegwerffunktion“, nur einmal beim Funktionsaufruf von „sort“ benötigt wird. Alternativ hätte man eine Lambda-Funktion verwenden können:

```
>>> l = [("a", 1), ("c", -1), ("b", -2)]
>>> l.sort(key = lambda x: x[1])
>>> l
[('b', -2), ('c', -1), ('a', 1)]
```

map

Mit Hilfe von „map“ kann man aus einer Sequenz (z.B. Liste, Tupel) unter Anwendung einer Funktion auf ein map-Objekt, also einen Iterator, abbilden:

```
>>> kmh = [30, 40, 25, 120]
>>> ms = map(lambda x: x/3.6, kmh)
>>> list(ms)
[8.333333333333334, 11.11111111111111, 6.944444444444445, 33.333333333333336]
>>> list(ms)
[]
>>> ms = map(lambda x: x/3.6, kmh)
>>> for speed in ms:
...     print(speed)
...
8.333333333333334
11.11111111111111
6.944444444444445
33.333333333333336
```

filter

Mit Hilfe von „filter“ kann man aus einem iterierbaren Objekt unter Anwendung einer Funktion, die einen Wahrheitswert zurückliefert, Elemente herausfiltern:

```
>>> numbers = [1,2,3,4,5,6,7,8,9,10]
>>> odd_numbers = filter(lambda x: x % 2, numbers)
>>> list(odd_numbers)
[1, 3, 5, 7, 9]
>>> type(odd_numbers)
<class 'filter'>
>>> odd_numbers = filter(lambda x: x % 2, numbers)
>>> for n in odd_numbers:
...     print(n)
...
1
3
5
7
9
```

Funktionen mit beliebiger Anzahl an Argumenten

Variable Anzahl an Parametern

- Man hat häufig Fälle, in denen die Anzahl der beim Aufruf nötigen Parameter einer Funktion a priori nicht bekannt ist.
- Zur Definition von Funktionen, die eine beliebige Anzahl an Parametern erhalten können, verwendet man den „*“-Operator:

```
>>> def printall(*x):
...     print(x)
...
>>> printall(1,2,3)
(1, 2, 3)
>>> printall(1,2,3,4,5)
(1, 2, 3, 4, 5)
```

- Man sieht, dass die an „printall“ übergebenen Argumente beim Aufruf in einem Tupel gesammelt werden. Die Argumente werden durch den „*“-Operator „gepackt“.

Variable Anzahl an Parametern

- Es ist auch möglich, eine feste Anzahl an Positionsparametern mit einer beliebigen Anzahl an weiteren Parametern zu kombinieren.
- Dies ist möglich, allerdings müssen die Positionsparameter immer zuerst kommen.
- Im folgenden Beispiel wird ein Positionsparameter „erster_wert“ erwartet, gefolgt von beliebig vielen weiteren Parametern.
- Auf diese Weise kann verhindert werden, dass die Funktion ohne Parameter aufgerufen wird, was eine Division durch Null zur Folge hätte.

```
>>> def arimittel(erster_wert, *weitere_werte):
...     return (erster_wert+sum(weitere_werte))/(1+len(weitere_werte))
...
>>> arimittel(1)
1.0
>>> arimittel(1,2)
1.5
```

Funktionsaufrufe mit *

- Ein Stern kann auch in einem Funktionsaufruf erscheinen. Die Semantik ist in diesem Fall invers zu der Verwendung eines Sterns in der Funktionsdefinition.
- Das bedeutet: die Elemente einer Liste oder eines Tupels werden vereinzelt bzw. „entpackt“:

```
>>> arimittel(1,2,3,4,5,6,7,8,9,10)
5.5
>>> p = [i for i in range(1,11)]
>>> arimittel(*p)
5.5
```

- Es gibt auch einen analogen Mechanismus für eine beliebige Anzahl von Schlüsselwortparametern. Dazu wurde als Notation ein doppeltes Sternchen eingeführt:

```
>>> def f(**kwargs):
...     print(kwargs)
...
>>> f(de="German", en="English", fr="French")
{'de': 'German', 'en': 'English', 'fr': 'French'}
```

Ostbayerische Technische Hochschule
Amberg-Weiden

Dateien lesen und schreiben

Lesen aus einer Datei

- Die „**open**“-Funktion erzeugt ein Dateiobjekt und liefert eine Referenz auf dieses Objekt. Sie erwartet als Argumente einen Dateinamen und optional einen Modus.
- Im folgenden Beispiel öffnen wir die Daten „ad_lesbiam.txt“ zum Lesen (der Modus ist auf „r“ gesetzt).
- Mittels einer for-Schleife kann man über die Zeilen der Datei iterieren.
- Mittels „**close**“ wird die Datei wieder geschlossen.

```
>>> fobj = open("ad_lesbiam.txt", "r")
>>> for line in fobj:
...     print(line.rstrip())
...
Vivamus, mea Lesbia, atque amemus
Rumoresque senum severiorum
Omnes unius aestimemus assis !

Soles occidere et redire possunt:
Nobis cum semel occidit brevis lux,
Nox est perpetua una dormienda.
>>> fobj.close()
```

Schreiben in eine Datei

- Zum Schreiben einer Datei verwendet man als Modus „w“ anstatt „r“.
- Daten schreibt man mit der Methode „write“ des Dateiobjekts.

```
fobj_in = open("ad_lesbiam.txt", "r")
fobj_out = open("ad_lesbiam2.txt", "w")

counter = 0
for line in fobj_in:
        counter += 1
        out_line = "{0: >3s} {1:s}\n".format(str(counter), line.rstrip())
        fobj_out.write(out_line)

fobj_in.close()
fobj_out.close()
```

ad_lesbiam2.txt

```
  1 Vivamus, mea Lesbia, atque amemus
  2 Rumoresque senum severiorum
  3 Omnes unius aestimemus assis !
  4
  5 Soles occidere et redire possunt:
  6 Nobis cum semel occidit brevis lux,
  7 Nox est perpetua una dormienda.
```

read und readlines

- Bis jetzt haben wir Dateien Zeile für Zeile mit einer Schleife eingelesen.
- Alternativ kann man auch mit **read**() und **readlines**() den gesamten Inhalt in eine Datenstruktur speichern.
- Die Methode **read**() erzeugt einen String aus dem Dateiinhalt.
- Die Methode **readlines**() erzeugt eine Liste, die die Zeilen der Datei als Einträge enthält.

```
>>> f = open("ad_lesbiam.txt", "r")
>>> f.read()
'Vivamus, mea Lesbia, atque amemus\nRumoresque senum severiorum\nOmnes
unius aestimemus assis !\n\nSoles occidere et redire possunt:\nNobis
cum semel occidit brevis lux,\nNox est perpetua una dormienda.'
>>> f.close()
```

```
>>> f = open("ad_lesbiam.txt", "r")
>>> f.readlines()
['Vivamus, mea Lesbia, atque amemus\n', 'Rumoresque senum severiorum\n',
'Omnes unius aestimemus assis !\n', '\n', 'Soles occidere et redire
possunt:\n', 'Nobis cum semel occidit brevis lux,\n', 'Nox est perpetua
una dormienda.']
>>> f.close()
```

with-Anweisung

- Die Dateiverarbeitung mit **open**() und **close**() funktioniert zwar, sollte aber in dieser Form nicht verwendet werden.
- Stattdessen sollte eine **with**-Anweisung verwendet werden.
- Diese stellt sicher, dass die Datei im Fall einer Ausnahme oder am Ende der Verarbeitung automatisch geschlossen wird. Ein explizites **close**() ist dann nicht mehr notwendig.

```
>>> with(open("ad_lesbiam.txt", "r")) as fobj:
...     for line in fobj:
...             print(line.rstrip())
...
Vivamus, mea Lesbia, atque amemus
Rumoresque senum severiorum
Omnes unius aestimemus assis !

Soles occidere et redire possunt:
Nobis cum semel occidit brevis lux,
Nox est perpetua una dormienda.
```

Pickle

Daten sichern mit pickle.dump()

Ostbayerische Technische Hochschule Amberg-Weiden

- Mit Hilfe des Moduls „pickle“ kann man Daten über das Programmende hinaus persistieren.
- Mit der Methode **dump**() lassen sich Objekte in serialisierter Form abspeichern, sodass sie später wieder deserialisiert werden können.
- Die Syntax lautet:

```
dump(obj, file, protocol=None, *, fix_imports=True)
```

Argument	Erläuterung
obj	Objekt, das serialisiert werden soll
file	Dateiobjekt, in das gepickelt wird
protocol	Art der Ausgabe (z.B. für Menschen lesbares Format, Binärformat. etc.)
fix_imports	falls True und protocol<3, versucht Pickle, die neuen Python3-Namen auf die alten Modulnamen zu setzen, sodass der Pickle-Datenstrom auch mit Python 2 lesbar ist.

Daten sichern mit pickle

- Mit dump() kann man mehrere Objekte in derselben Datei abspeichern.
- Es stellt sich die Frage, welche Art von Daten man eigentlich pickeln kann. Im Prinzip fast alles:
  - alle Python-Basistypen (z.B. Booleans, Integers, Floats, Strings)
  - Listen und Tupel, Mengen und Dictionaries
  - Funktionen, Klassen und Instanzen
  - file-Objekte lassen sich nicht pickeln.
- Beispiel:

```
>>> cities = ["Regensburg", "Amberg", "Weiden", "Neumarkt"]
>>> import pickle
>>> with(open("cities.pkl", "wb")) as fh:
...     pickle.dump(cities, fh)
```

- Das Wiederherstellen erfolgt mit **pickle.load**():

```
>>> import pickle
>>> with(open("cities.pkl", "rb")) as fh:
...     cities = pickle.load(fh)
...
>>> cities
['Regensburg', 'Amberg', 'Weiden', 'Neumarkt']
```