FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Einführung in Python

5. Vorlesung

Objektorientierte Programmierung

Eigene Klassen & Module

Wiederholung letztes Mal

- Methoden zur Manipulation von Zeichenketten
  - Formatieren & Schreibweise ändern
  - Einfache Suffix-, Präfix- und Zeichentests
  - Entfernen & Aufspalten
  - Suchen & Ersetzen
- Codierung und Decodierung

platonisches Zeichen ⇄ (Codierung / Decodierung) Oktette (Bytestring)

Ägäis ⇄ (UTF-8 Coderiung / UTF-8 Decodierung)

Ä	g	ä	i	s
195 132	103	195 164	105	115
1100 1000	0110	1100 1010	0110	0111
0011 0100	0111	0011 0100	1001	0011

Wiederholung letztes Mal

- Automatische Textproduktion
  - Direkt: in-place casting
  - Indirekt: Platzhalter (*format()*-Methode)
  - Direkte Platzhalter: f-Strings

- Reguläre Ausdrücke

```
>>> from re import *                    #Import aller Funktionen aus dem Modul re
>>> regex = compile(‘\d+‘)              #Erzeugen eines RE-Objekts namens ‘regex‘

>>> print(regex)                        #print gibt den re-Funktionsaufruf wieder
re.compile('\\d+')

>>> type(regex)
<class '_sre.SRE_Pattern'>
```

Wiederholung letztes Mal

- Dateiformate allgemein: Binär vs. Text

- CSV – *Character separated values*

```
Name,Vorname,Alter,Rolle
Chapman,Graham,48,“König Arthur,Wächter,Stimme Gottes“
Cleese,John,66,“Sir Lancelot,Tim der Zauberer,Schwarzer Ritter“
Gilliam,Terry,65,“Knappe Patsy,Sir Bors,Grüner Ritter“
Idle,Eric,63,“Sir Robin,Diener Concord,Bruder Maynard“
Jones,Terry,65,“Sir Bedevere,Prinz Herbert,Landarbeiterin“
Palin,Michael,64,“Sir Galahad,Dennis,Herr des Sumpfschlosses“
```

- JSON – *JavaScript Object Notation*

```
[
   {
      “name“: “idle“, “vorname“: “eric“, “alter“: 63,
      “rolle“: [“Sir Robin“, “Diener Concord“, “Bruder Maynard“,]
   }
]
```

- XML – *Extensible Markup Language*

```
<someRoot>
    <someElem>
        <otherElem someAttr="value">Some text</otherElem>
    </someElem>
</someRoot>
```

Eigene Klassen & Module

- Definition von Klassen
- Klassenattribute und -methoden
- Vererbung
- Überladung und Magische Methoden
- Klassenbibliotheken und Module

Klassen und Objekte

- Python ist von Grund auf für objektorientierte Programmierung ausgelegt, aber man muss dieses Konzept nicht aktiv nutzen

- Grundkonzept: Daten und deren Funktionen (Methoden), die auf diese angewendet werden können, sind in so genannte Objekte zusammengefasst

- Objekte sind nach außen abgekapselt, so dass die Daten von Benutzern und anderen Objekten nicht verändert werden können

- Objekte können untereinander kommunizieren, in dem sie Informationen austauschen

Klassen und Objekte

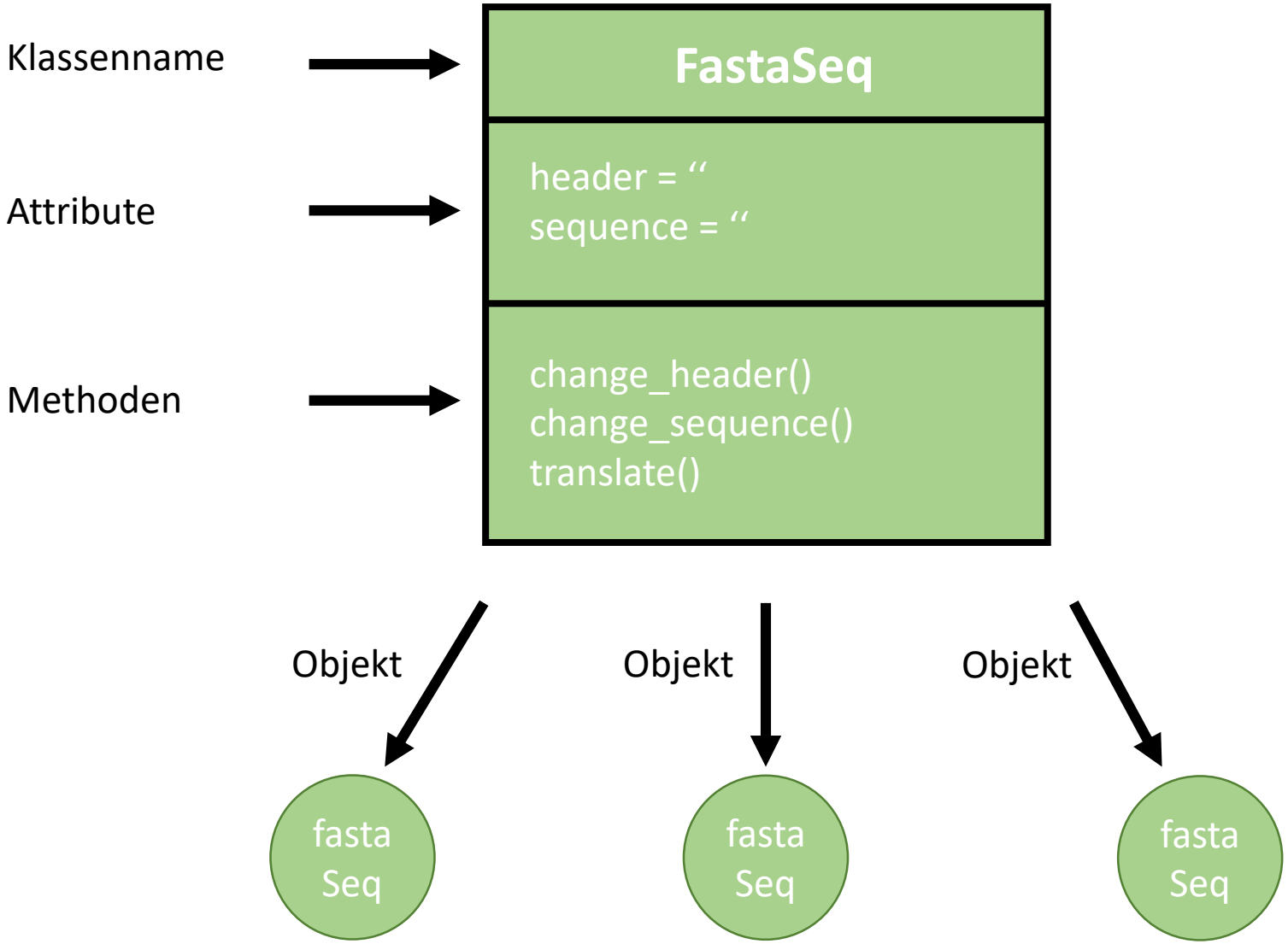
- **Klassen** sind *Baupläne* für Objekte; sie beschreiben formal wie ein Objekt beschaffen ist, d. h. welche **Attribute** und **Methoden** es besitzt

- Aber: Eine Klasse ist **kein** Objekt, sondern erzeugt Objekte (**Instanzen**)

- Attribute sind Variablen, d. h. sie bestehen aus einem Namen und haben einen Wert → Alle Objekte einer Klassen besitzen die gleichen Attribute, aber möglichweise unterschiedliche Wertebelegungen (Ausnahme: *Klassenattribute*)

- Objekte können Operationen (Methoden) ausführen, wenn sie dazu veranlasst werden

Klassen und Objekte



Klassen und Objekte

```
>>> class FastaSeq():
        pass                    #Das Schlüsselwort pass erlaubt ‘leere‘ Anweisungen


>>> x = FastaSeq()                  #Erzeugen einer Instanz von FastaSeq mit dem Namen x
>>> y = FastaSeq()          #Erzeugen einer zweiten Instanz von FastaSeq mit dem Namen y
>>> y2 = y

>>> x == y
False                   #x und y sind zwei unterschiedliche Objekte der gleichen Klasse

>>> y == y2
True                        #y2 ist nur eine weitere Referenz (ein zweiter Name) für y
```

Eigenschaften und Attribute

- Einem Objekt können dynamisch beliebige Attributsnamen zugewiesen werden

- Diese haben aber noch nichts mit den eigentlich Instanzattributen zu tun

```
>>> x = FastaSeq()
>>> y = FastaSeq()
>>> x.sequence = ‘ACGT‘                #dynamische Zuweisung eines Attributs
>>> x.header = ‘>XYZ‘

>>> x.sequence                  #auf gesetzte Attribute kann auch zugegriffen werden
‘ACGT‘
>>> x.header
‘>XYZ‘
>>> y.header                     #Fehler bei Zugriff auf nicht vorhandene Attribute
AttributeError: 'FastaSeq' object has no attribute 'header'
```

Eigenschaften und Attribute

```
>>> FastaSeq.length = 0          #Man kann auch dynamisch Klassenattribute setzen
>>> z = FastaSeq()
>>> z.length                #Jede Instanz besitzt automatisch alle Klassenattribute
0

>>> def antwort(*x):                        #Auch Funktionen sind Objekte in Python
        return(42)

>>> antwort.farbe = ‘grün‘          #Daher können auch Funktionen Attribute haben
>>> antwort.farbe
‘grün‘
>>> antwort(‘Was ist der Sinn des Lebens?‘)
‘42‘                           #Diese haben aber keine Auswirkung auf die Funktion

>>> def antwort(x):                                  #Attribute können als Ersatz für
        if hasattr(antwort, ‘zähler‘):             #statische Funktionsvariablen
            antwort.zähler += 1                          #genutzt werden
        else:
            antwort.zähler = 1
        return(42)
```

Eigenschaften und Attribute

- Die Objekte der meisten Klassen haben ein Attribute-Dictionary **__dict__** in dem alle Attribute und deren Werte gespeichert sind

```
>>> x = FastaSeq()
>>> x.sequence = ‘ACGT‘
>>> x.header = ‘>XYZ‘

>>> x.__dict__
{'header': '>XYZ', 'sequence': 'ACGT'}
```

Methoden und Instanzattribute

- **Instanzattribute** müssen unmittelbar in der Klassendefinition durch Methoden erzeugt werden

- Im Prinzip unterscheiden sich Methoden von Funktionen nur in zwei Dingen:
  - Methoden sind Funktionen die **innerhalb** eines class-Anweisungsblocks stehen
  - Das erste Argument einer Methode ist immer die Referenz auf die Instanz, von der sie aufgerufen wird (*Pythonschlüsselwort*: **self**)

Methoden und Instanzattribute

```
>>> class FastaSeq():

        def alpha(self):          #Definieren einer Methode; self nicht vergessen!
            print('ACGTU')

>>> x = FastaSeq()
>>> x.alpha()                     #Jede Instanz besitzt alle Methoden der Klassen
'ACGTU'                           #diese werden ohne das Argument self aufgerufen
```

Methoden und Instanzattribute

- Um Instanzattribute zu setzen muss der Objektname nicht bekannt sein

```
>>> class FastaSeq():

        def alpha(self):        #Definieren einer Methode; self nicht vergessen!
            print('ACGTU')

        def set_header(self, header):
            self.header = header

        def get_header(self):
            print(self.header)

>>> x = FastaSeq()
>>> x.set_header('>ABCDE')
>>> x.get_header()
'>ABCDE'
```

Methoden und Instanzattribute

- Um Attribute sofort nach der Erzeugung der Instanz zu setzen, definiert man die magische **__init__** Methode, diese wird bei der Initialisierung eines Objekts aufgerufen (Pseudokonstruktor)

```
>>> class FastaSeq():

        def __init__(self, header, sequence):        #Auch die __init__ Methode
            self.header = header                     #braucht die Selbstreferenz
            self.sequence = sequence

        ...

>>> x = FastaSeq(‘>XYZ‘, ‘ACGT‘)
>>> x.get_header()
‘>XYZ‘
>>> y = FastaSeq()
TypeError: __init__() takes exactly 3 arguments (1 given)
```

Methoden und Instanzattribute

- Um Attribute sofort nach der Erzeugung der Instanz zu setzen, definiert man die magische **__init__** Methode, diese wird bei der Initialisierung eines Objekts aufgerufen (Pseudokonstruktor)

```
>>> class FastaSeq():

        def __init__(self, header='>', sequence=''):
            self.header = header
            self.sequence = sequence

        ...

>>> x = FastaSeq()
>>> x.get_header()
'>'
```

Datenabstraktion = Datenkapselung + Geheimnisprinzip

- Unter Datenkapslung versteht man den Schutz der Daten eines Objekts vor **direktem Zugriff**

- Zugriff auf Objektdaten/-attribute sollte nur über entsprechende (Zugriffs)Methoden erfolgen (Getter und Setter)

- Unter Geheimnisprinzip versteht man das ‘verstecken‘ interner Implementierungsdetails → Objektdaten sind von ausserhalb **nicht sichtbar**

Datenabstraktion = Datenkapselung + Geheimnisprinzip

- Python kennt drei Stufen von Datenabstraktion:
  - name (**Public**): Attribut ohne führende Unterstriche; sind innerhalb einer Klasse und auch von außen les- und schreibbar
  - _name (**Protected**): Man kann zwar von außen lesend und schreibend zugreifen, der Entwickler macht aber klar das man diese Attribute so nicht benutzen sollte; Protected-Attribute sind beim Importieren wichtig
  - __name (**Private**): Sind von außen weder sichtbar noch benutzbar

Datenabstraktion = Datenkapselung + Geheimnisprinzip

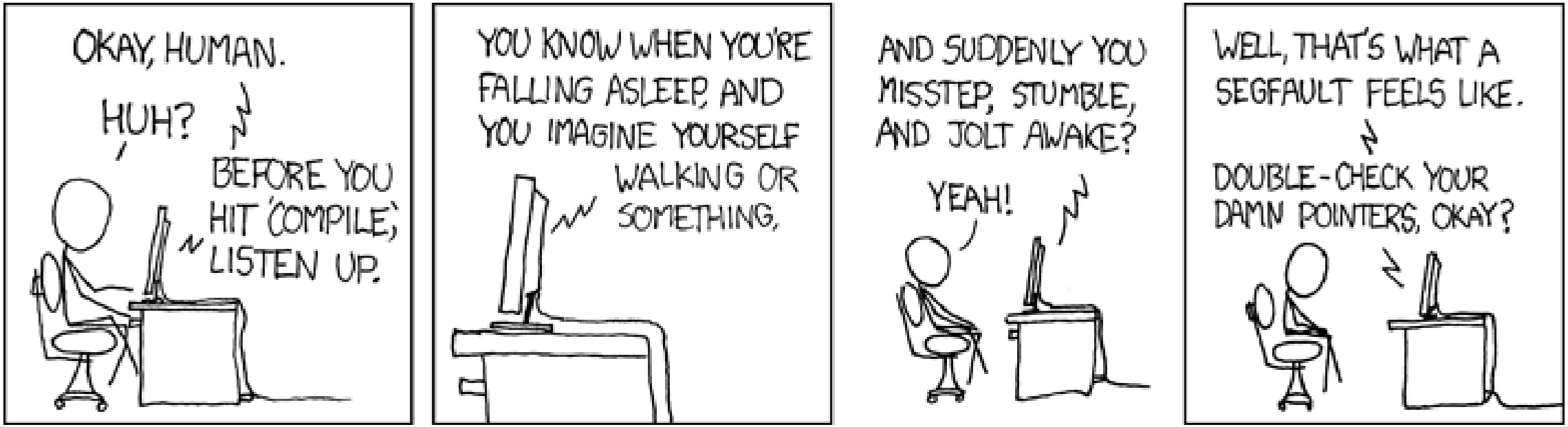
```
>>> class Abstraktion():

        def __init__(self):
            self.__priv = ‘Lass mich, ich bin privat!‘
            self._prot = ‘Eigentlich solltest du mich nicht anfassen!‘
            self.pub = ‘Mit mir kannst du machen was du willst!‘

>>> x = Abstraktion()
>>> x.pub
‘Mit mir kannst du machen was du willst!‘
>>> x.pub = ‘Oh ja, ändere mich hart!‘

>>> x._prot
‘Eigentlich solltest du mich nicht anfassen!‘
>>> x._prot = ‘Ich weiß nicht ob mir gefällt was du mit mir machst...‘

>>> x.__priv
AttributeError: ‘Abstraktion‘ object has no attribute ‘__priv‘
```

Datenabstraktion = Datenkapselung + Geheimnisprinzip

```
>>> class FastaSeq():

        def __init__(self, header='>', sequence=''):
            self.__header = header                    #Beide Attribute sind private
            self.__sequence = sequence

        #Hier werden getter und setter für beide Attribute definiert
        def set_header(self, newH):
            self.__header = newH

        def set_sequence(self, newS):
            self.__sequence = newS

        def get_geader(self):
            print(self.__header)

        def get_header(self):
            print(self.__sequence)
```

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Good programming practice: Lektion VI



Good programming practice: Lektion VI

- In den meisten objektorientierten Sprachen sollte man möglichst alle Attribute private setzen und gegebenenfalls mit Gettern und Settern versehen

- Python ist sich zu cool für häßliche get_X() und set_X() Methodenaufrufe → **Properties**

```
>>> class P:

        def __init__(self, x):
            self.__x = x

        def __get_X(self):
            return(self.__x)

        def __set_X(self, x):
            self.__x = x

        x = property(__get_X, __set_X)

>>> p = P(19)
>>> p.x = 119
>>> p.x
119
#Zugriff als ob Attribut x public wäre
```

Good programming practice: Lektion VI

- Eigentlich ist sich Python überhaupt zu cool für echte Datenkapselung, denn nichts ist wirklich richtig privat

- Getter und Setter sollten in Python nur verwendet werden, um bestimmte Tests oder Umwandlungen von Eingaben vorzunehmen

```
>>> class P:
        def __init__(self, x):
            self.__x = x

>>> p = P(19)
>>> p._P__x = 119
>>> p._P__x
119
#Auch wenn es möglich ist, sollte man
#private Attribute so niemals ändern
>>> class P:

        def __init__(self, x):
            self.__x = x

        def set_X(self, x):
            if x > 9000:
                self.__x = 9000
            else:
                self.__x = x
```

Klassenattribute und -methoden

- Bis jetzt haben wir nur **nicht-statische** Instanzattribute betrachtet
  → diese Attribute wurden für jedes Objekt dynamisch neu erstellt

- Um Klassenrelevante Informationen zu speichern, die sich nicht auf ein bestimmtes Objekt beziehen nutzt man statische **Klassenattribut**

- Klassenattribute existieren unabhängig von Instanzen und sind für alle Instanzen gleich

Klassenattribute und -methoden

```
>>> class FastaSeq():
        counter = 0                                  #Definieren eines Klassenattributes

        def __init__(self, header='>', sequence=''):
            self.__header = header
            self.__sequence = sequence
            type(self).counter += 1            #Auch möglich: FastaSeq.counter += 1

        def __del__(self):
            type(self).counter -= 1

>>> FastaSeq.counter                                 #Auf Klassenattribute kann man
0                                                    #auch ohne Instanz zugreifen
>>> x = FastaSeq()
>>> y = FastaSeq()
>>> x.counter
2
>>> del x                                       #Löschen von x und Aufruf von __del__
>>> y.counter
1
```

Klassenattribute und -methoden

```
>>> class FastaSeq():
        __counter = 0                          #Klassenattribute sollten private sein

        def __init__(self, header='>', sequence=''):
            self.__header = header
            self.__sequence = sequence
            type(self).__counter += 1

        def __del__(self):
            type(self).__counter -= 1

        def get_counter(self):                 #Instanzmethode als Getter für counter
           return(type(self).__counter)

>>> x = FastaSeq()
>>> x.get_counter()
1
```

- Man kann Instanzmethoden nutzen um private Klassenattribute auszulesen, ist allerdings nicht sehr sinnvoll

Klassenattribute und -methoden

```
>>> class FastaSeq():
        __counter = 0                          #Klassenattribute sollten private sein

        def __init__(self, header='>', sequence=''):
            self.__header = header
            self.__sequence = sequence
            type(self).__counter += 1

        def __del__(self):
            type(self).__counter -= 1

        def get_counter():                        #Instanzmethode ohne Selbstreferenz
           return(FastaSeq.__counter)

>>> FastaSeq.get_counter()
0                              #Jetzt ist der Getter zwar von der Klasse ausführbar
>>> x = FastaSeq()
>>> x.get_counter()            #aber nicht mehr von den Instanzen der Klasse
TypeError: get_counter() takes no arguments (1 given)
```

Klassenattribute und -methoden

```
>>> class FastaSeq():
        __counter = 0                          #Klassenattribute sollten private sein

        def __init__(self, header='>', sequence=''):
            self.__header = header
            self.__sequence = sequence
            type(self).__counter += 1

        def __del__(self):
            type(self).__counter -= 1

        @classmethod                    #Dekorator zur Markierung einer Klassenmethode
        def get_counter(cls):
           return(cls.__counter)                  #Auch möglich: FastaSeq.__counter

>>> FastaSeq.get_counter()
0
>>> x = FastaSeq()
>>> x.get_counter()
1
```

@classmethod versus @staticmethod

- Neben dem *@classmethod* gibt es noch den *@staticmethod* Dekorator um eine statische Methode zu definieren

- Diese statischen Methoden können auch von der Klasse und aller Instanzen dieser Klasse aufgerufen werden, haben aber keine impliziten Referenzen in ihrem Methodenkopf
  → sie wissen nicht wer sie aufgerufen hat

- Statische Methoden dienen hauptsächlich der Verknüpfung von Funktionen die logisch mit einer Klasse verbunden sind

@classmethod versus @staticmethod

```
>>> class A:

        def instance_method(self, x):
            print(f‘Diese Instanzmethode wurde aufgerufen von {self} mit {x}‘)

        @classmethod
        def class_method(cls, x):
            print(f‘Diese Klassenmethode wurde aufgerufen von {cls} mit {x}‘)

        @staticmethod
        def jon_schnee(x):
            print(f‘Jon Schnee weiß nichts, außer dem Wert von {x}‘)

>>> a = A()
>>> a.jon_schnee(4)
‘jonSchnee weiß nichts außer dem Wert von 4‘
>>> a.class_method(4)                                  #Entspricht: A.classMethod(A, 4)
‘Diese Klassenmethode wurde aufgerufen von <class '__main__.A'> mit 4‘
>>> a.instance_method(4)                             #Entspricht: A.instanceMethod(a, 4)
‘Diese Instanzmethode wurde aufgerufen von <__main__.A object at 0x03438FF0> mit 4‘
```

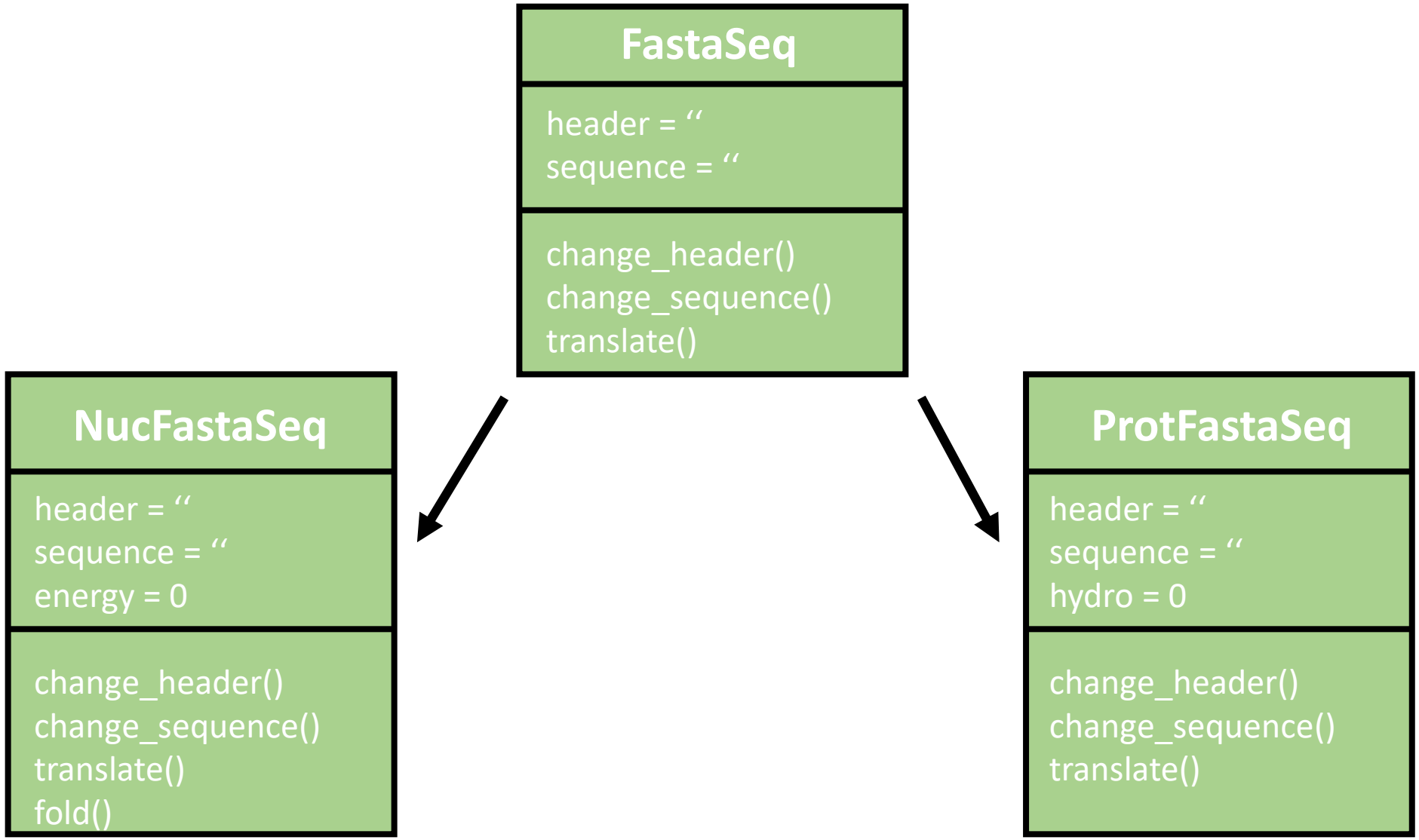
Vererbung

- Im Allgemeinen definiert man über Vererbung eine Verbindung zwischen einer Oberklasse und einer Unterklasse

- Dabei erbt die Unterklasse alle Attribute und Methoden der Oberklasse und besitzt in der Regel noch zusätzliche eigene Attribute und Methoden

- Damit sind Oberklassen allgemeine und Unterklassen spezialisierte Klassen

Vererbung


FastaSeq
header = ‘‘
sequence = ‘‘
change_header()
change_sequence()
translate()
NucFastaSeq
header = ‘‘
sequence = ‘‘
energy = 0
change_header()
change_sequence()
translate()
fold()
ProtFastaSeq
header = ‘‘
sequence = ‘‘
hydro = 0
change_header()
change_sequence()
translate()

Vererbung

```
>>> class NucFastaSeq(FastaSeq):  #In Klammern kommt die Klasse von der geerbt wird

        def __init__(self, header='>', sequence='', energy=0):
            FastaSeq.__init__(self, header, sequence)   #Aufruf vom Basis __init__
            self.energy = energy                        #Neues Instanzattribut

        def fold(self):                                 #Neue Instanzmethode
            print('Feature coming soon...')


>>> y = NucFastaSeq('>ABC', 'TGCA')        #Erzeugen einer Instanz von NucFastaSeq
>>> y.sequence                             #y hat alle Attribute von FastaSeq
'TGCA'
>>> y.energy                               #y kennt aber auch seine neuen Attribute
0
>>> y.get_counter()                        #y hat auch alle Methoden von FastaSeq
1
>>> y.fold()                               #y kennt aber auch seine neuen Methoden
'Feature coming soon...'
```

Überschreiben und Überladen

- Unter **Überschreiben** versteht man in der OOP, das es einer Unterklasse erlaubt ist, geerbte Methoden neu zu implementieren

- Dabei wird die geerbte Methode komplett ersetzt (man kann sie aber bei Bedarf innerhalb der Neuimplementierung noch aufrufen)

- **Überladen** ist auch ein Konzept aus der OOP, das es erlaubt mehrere Methoden mit dem gleichen Namen, aber unterschiedlicher Paramterlisten zu definieren

- Python kann Funktionen nicht überladen, nur überschreiben

Überschreiben und Überladen

```
>>> class FastaSeq():                                          #Oberklasse
        def __init__(self, header, sequence):
            self.header = header
            self.sequence = sequence

        def translate(self):
            pass

>>> class NucFastaSeq(FastaSeq):                              #abgeleitete Klasse
        def __init__(self, header='>', sequence='', energy=0):
            FastaSeq.__init__(self, header, sequence)
            self.energy = energy

        def translate(self):                                  #Überschriebene Basismethode
            print('Feature coming soon...')

>>> y = NucFastaSeq()
>>> y.translate()                                  #Aufruf der Basismethode ist von
'Feature coming soon...'                           #abgeleiteten Instanzen nicht möglich
```

Überschreiben und Überladen

```
>>> class NucFastaSeq(FastaSeq):
        def __init__(self, header='>', sequence='', energy=0):
            FastaSeq.__init__(self, header, sequence)
            self.energy = energy

        def reverse(self):
            print('Erste Definition von reverse.')

        def reverse(self, x):
            print('Zweite Definition von reverse.')

>>> y = NucFastaSeq()
>>> y.reverse(1)
'Zweite Definition von reverse.'
>>> y.reverse()
TypeError: reverse() missing 1 required positional argument: 'x'
```

- So funktioniert das Überladen von Funktionen nicht in Python, Funktionen gleichen Namens überschreiben sich immer, unabhängig von der Parameterliste

Überschreiben und Überladen

- „Überladen“ lässt sich mit Hilfe von **Default-Parametern** und der *args Parameterfunktion realisieren

```
>>> class NucFastaSeq(FastaSeq):
        def __init__(self, header='>', sequence='', energy=0):
            FastaSeq.__init__(self, header, sequence)
            self.energy = energy

        def reverse(self, x=None):
            if x != None:
                print('Es wurde ein zusätzlicher Parameter übergeben.')
            else:
                print('Aufruf ohne zusätzlichem Parameter.')

>>> y = NucFastaSeq()
>>> y.reverse(1)
'Es wurde ein zusätzlicher Parameter übergeben.'
>>> y.reverse()
'Aufruf ohne zusätzlichem Parameter.'
```

Überschreiben und Überladen

- „Überladen“ lässt sich mit Hilfe von Default-Parametern und der ***args Parameterfunktion** realisieren

```
>>> class NucFastaSeq(FastaSeq):
        def __init__(self, header='>', sequence='', energy=0):
            FastaSeq.__init__(self, header, sequence)
            self.energy = energy

        def reverse(self, *args):
            if len(args) == 1 and isinstance(args[0], int):
                print('Irgendeine Operation auf genau einem Integer.')
            elif len(args) > 1:
                if all(isinstance(x, int) for x in args):
                    print('Irgendeine Operation mit mehreren Integern.')
            else:
                raise(TypeError)
```

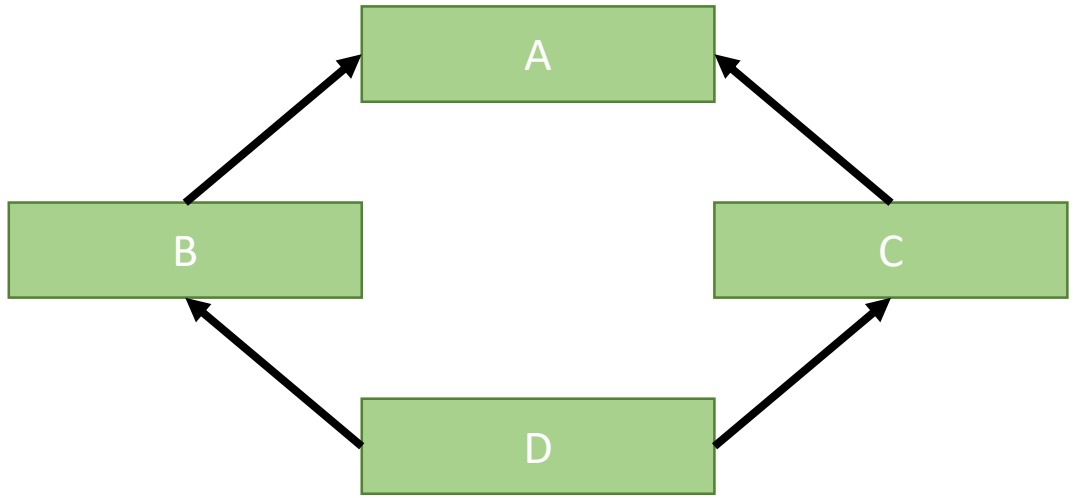
Mehrfachvererbung

- Eine Unterklasse erbt von mehr als einer Oberklasse alle Methoden und Attribute (nicht zu verwechseln mit sequentiellem mehrstufigem erben)

class Unterklasse(Oberklasse1, Oberklasse2, …)**:**

[Klassenkörper]

- Mehrfachvererbung kann zum sogenannten *Diamond-Problem* führen



Standardklassen als Oberklassen

- Durch das Ableiten von Standardklassen wie **int**, **float**, **str**, **list**, ... lassen sich diese um weitere Funktionalitäten erweitern

```
>>> class newList(list):
        def __init__(self, s=[], default=0):
            list.__init__(self, s)
            self.default = default

        def __getitem__(self, index):
             try:
                 return(list.__getitem__(self, index))
             except IndexError:
                 return(self.default)
>>> x = newList()
>>> x.append(19)
>>> x[0]
19
>>> x[100]
0
```

Magische Methoden

- Methoden die mit zwei Unterstrichen beginnen und enden, nennt man magische Methoden (wie z. B. __init__)

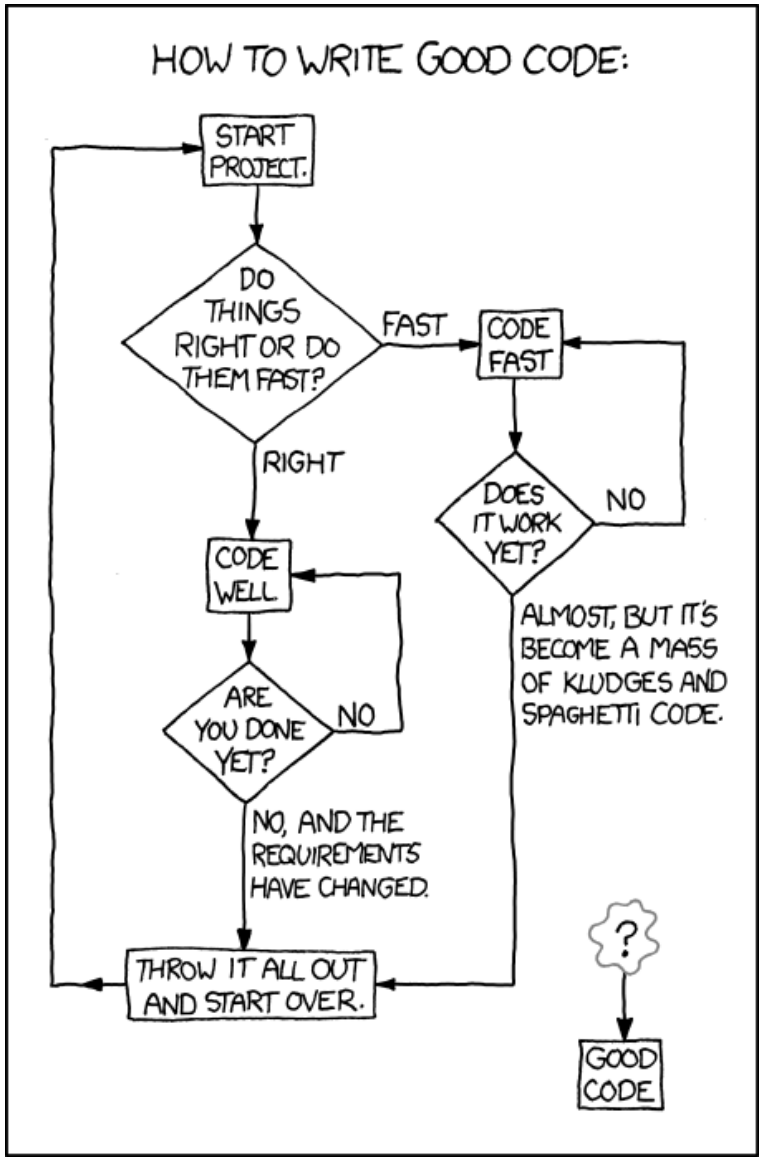
- Mit diesen speziellen Methoden kann man eigene Klassen um viele Grundoperationen oder –methoden einfach erweitern

- Hinter allen Grundoperationen wie + - * / ... oder Grundfunktionen wie str(), len(), ... stecken in Python gewöhnliche Klassen- und Instanzmethoden die sich vererben und überschreiben lassen

Magische Methoden

```
>>> class NucFastaSeq(FastaSeq):
        def __init__(self, head='>', seq='', energy=0):
            FastaSeq.__init__(self, head, seq)
            self.energy = energy

        def __len__(self):
            return(len(self.seq))

        def __str__(self):
            return(f'{self.head}\n{self.seq}')

        def __add__(self, other):
            return(NucFastaSeq(self.head+other.head[1:], self.seq+other.seq))
>>> x = NucFastaSeq('>ABC','ATGC')
>>> y = NucFastaSeq('>DEF','AAAA')
>>> z = x + y
>>> str(z)
'>ABCDEF\nATGCAAAA'
>>> len(z)
8
```

Good programming practice: Lektion VII



Good programming practice: Lektion VII

- Achtet beim Zugriff auf Objektattribute auf die richtige Schreibweise der Attributsnamen, sonst erzeugt ihr einfach dynamisch neue Werte ohne die alten zu ändern

```
>>> x = NucFastaSeq()
>>> x.sequenc = ‘GGAATTCC‘          #da das Attribut sequenc nicht existiert wird
                                    #es dynamisch an x gebunden
>>> x.sequence                      #das eigentliche Attribute wurde nicht geändert
‘‘
```

- Verwechseln von Methoden und Attributen vermeiden, da ein weglassen der Klammern nicht zwangsläufig zu einem Laufzeitfehler führt

```
>>> x = NucFastaSeq()
>>> x.reverse                                #lässt man Methodenklammern weg, wird
                                             #eine Stringrepräsentation wiedergegeben
<bound method NucFastaSeq.reverse of <__main__.NucFastaSeq instance at
0x000000000312C488>>
```

Klassenbibliotheken und Module

- Um eigene Klassen in anderen Skripten zu verwenden, muss man sie in einer eigenen Datei abspeichern
- Diese Dateien müssen wie normale Pythonskripte mit **.py** enden und werden Module genannt
- Ein Modul kann mehrere Klassendefinitionen und/oder einzelne Funktionsdefinitionen oder sogar einzelne Variablen enthalten

Klassenbibliotheken und Module

- Mit dem Schlüsselwort **import** lassen sich alle Definitionen eines Moduls in den aktuellen globalen Namensraum laden

- Mit **from ... import** lassen sich gezielt einzelne Definitionen eines Moduls laden

Klassenbibliotheken und Module

```
>>> import random                       #Import aller Klassen des Moduls random
>>> Random.randint(1,6)
4

>>> from random import randint          #Import einer bestimmten Funktion des Moduls
>>> randint(1,6)
4

>>> from random import *                #Import aller Klassen und Funktionen des Moduls
>>> randrange(6)
4

>>> import random as r                  #Import der Klasse Random unter anderem Namen
>>> r.randint(1,6)
4
```

The 5 steps of programming with Perl: