

Informatik I

17. OOP: Klassenmethoden, Klassen und Typen

Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

10. Dezember 2013

Methoden für Klassen

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

Warum Methoden für Klassen?

- Da es ja **Klassenattribute** gibt, macht es Sinn, Funktionen zu schreiben, die auf solchen arbeiten.
- Beispiele: Drucken der Anzahl aller Instanzen einer Klasse, Verringerung der Anzahl.
- Diese sollten auch **ohne** Instanz als erstes Argument aufrufbar sein.
- Dafür könnte man globale, externe Funktionen außerhalb der Klassendefinition einführen.
- Besser allerdings: **Integration** in die Klassendefinition
- U.U. auch Vererbung von Methoden, die auf Klassen-lokalen Attributen arbeitet.

- In unseren geometrischen Klassen wollen wir die jeweils tatsächlich für eine Klasse erzeugten Instanzen zählen.
- In der TwoDObject-Klasse soll die Anzahl aller Instanzen aller Subklassen gespeichert werden.
- Es soll eine Methode zum Drucken der Anzahlen geben.
- Die Methoden sollen über Instanzen und Klassennamen aufrufbar sein.
- Klassenattribute:
 - `instcnt` in allen Klassen, die Instanzen erzeugen
 - `allinstcnt` in `TwoDObject`

Statische Methoden

- Es gibt in Python sogenannte **statische Methoden**, bei denen kein Instanzen-Argument übergeben wird.
- Diese werden innerhalb der Klasse wie normale Funktionen (ohne `self`-Parameter) definiert, dann folgt die Zeile
`methodname = staticmethod(methodname)`

classmethods.py (1)

```
class TwoDObjects:
    allinstcnt = 0
    def __init__(self, x, y):
        TwoDObject.allinstcnt += 1

    def print_allinstcnt():
        print("%d global instances" %
              TwoDObject.allinstcnt)
print_allinstcnt = staticmethod(print_allinstcnt)
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

Aufruf der statischen Methode

Python-Interpreter

```
>>> t1 = TwoDObject()
>>> r1 = Rectangle()
>>> TwoDObject.allinstcnt
2
>>> r1.allinstcnt
2 >>> Rectangle.print_allinstcnt()
2 global instances
```

- Wenn wir jetzt für jede Klasse eine eigene Klassenvariable `instcnt` einführen, dann soll diese ja in jeder eigenen Klasse zählen!
- Dann müssten wir in jeder Klasse die entsprechenden Methoden einführen!
- Besser: **Klassenmethoden**, bei denen als erster Parameter ein Klassenobjekt übergeben wird.
- Ähnlich wie statische Methoden, aber mit folgender Zeile deklariert:

```
methodname = classmethod(methodname)
```

Klassenmethodenbeispiel (1)

classmethods.py (2)

```
class TwoDObjects:
    instcnt = 0

    def __init__(self, x, y):
        self.incr_instcnt()

    def print_instcnt(cls):
        print("%d local instances" %
              cls.instcnt)
    print_instcnt = classmethod(print_instcnt)

    def incr_instcnt(cls):
        cls.instcnt += 1
    incr_instcnt = classmethod(incr_instcnt)
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

Klassenmethodenbeispiel (2)

- In den Subklassen wird jetzt nur zusätzlich jeweils eine Klassenvariable `instcnt` eingeführt.

classmethods.py (3)

```
class Rectangle(TwoDObject):
    instcnt = 0
    def __init__(self, x=0, y=0, height=1, width=1):
        self.height = height
        self.width = width
        super().__init__(x, y)

class Square(Rectangle):
    instcnt = 0
    def __init__(self, x=0, y=0, side=1):
        super().__init__(x, y, side, side)
```

Klassenmethodenbeispiel (3)

Python-Interpreter

```
>>> t1 = TwoDObject(); t2 = TwoDObject()
>>> s1, s2, s3 = Square(), Square(), Square()
>>> r1 = Rectangle()
>>> TwoDObject.print_allinstcnt()
6 global instances
>>> TwoDObject.print_instcnt()
2 local instances
>>> r1.print_instcnt()
1 local instances
>>> s1.print_instcnt()
3 local instances
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

Es gibt drei Arten von Methoden:

- 1 **Instanzenmethoden**: (normale) Methoden, die auf einer Instanz agieren, und als ersten Parameter immer die Instanz erwarten – `self`
- 2 **Statische Methoden**: Methoden, die keine Referenz auf das Klassenobjekt (sic!) haben. Diese sollten am besten dann benutzt werden, wenn nur auf lokale Klassenvariablen zugegriffen werden soll.
- 3 **Klassenmethoden**: Methoden, die als ersten Parameter ein Klassenobjekt (`cls`) erwarten. Gut zu benutzen, wenn es Attribute gibt, die in mehreren Klassen mit gleichem Namen eingeführt werden.
- 4 ... und dann gibt es noch **Metaklassenmethoden**,

Typen sind Klassen

Informatik I

Bernhard
Nebel

Methoden
für Klassen

**Typen sind
Klassen**

Standard-
methoden

Standard-
methoden
des
String-Typs

- Alle Typen, die es in Python gibt, sind Klassen. D.h. alle Objekte der jeweiligen Typen (ob `int` oder `list`) sind **Instanzen** der entsprechenden Klassen.
- Insbesondere können wir Subklassen einführen und das Verhalten modifizieren!
- Beispiel: Ein Typ `ArabicInt`, bei dem die Evaluierungsreihenfolge rechts vor links ist:

$$9 - 35 // 7 \mapsto -9$$

$$35 // 7 \mapsto 0$$

$$9 - 0 \mapsto -9$$

Die Klasse ArabicInt

arabic.py

```
class ArabicInt(int):
    def __sub__(self, right):
        return ArabicInt(super().__rsub__(right))
    def __rsub__(self, left):
        return ArabicInt(super().__sub__(left))
    def __floordiv__(self, right):
        return ArabicInt(super().__rfloordiv__(right))
    def __rfloordiv__(self, left):
        return ArabicInt(super().__floordiv__(left))
    def __mod__(self, right):
        return ArabicInt(super().__rmod__(right))
    def __rmod__(self, left):
        return ArabicInt(super().__mod__(left))
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

ArabicInt in Aktion

Python-Interpreter

```
>>> 9 - 35 // 7
```

```
4
```

```
>>> ArabicInt(9 - 35 // 7)
```

```
4
```

```
>>> ArabicInt(9) - ArabicInt(35) // ArabicInt(7)
```

```
-9
```

```
>>> 9 - 35 // ArabicInt(7)
```

```
-9
```

```
>>> 9 - 5 - 35 // ArabicInt(7)
```

```
-4
```

```
>>> ArabicInt(10) - ArabicInt(3) - ArabicInt(29)
```

```
36
```

Verfolgen von Dict-Änderungen

- Ein etwas sinnvollerer Beispiel: Das Verfolgen von Änderungen in einem dict

logdict.py

```
class LogDict(dict):
    def __setitem__(self, key, value):
        print("Setting %r to %r in %r" % (key, value, self))
        super().__setitem__(key, value)

    def __delitem__(self, key):
        print("Deleting %r in %r" % (key, self))
        super().__delitem__(key)
```

- Bei jeder Änderung eine Ausgabe auf der Konsole.
- Im allgemeinen werden neue Typen aber nicht unbedingt als Erweiterung eingeführt (Beispiele : decimal und fractions)

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

Standardmethoden

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

**Standard-
methoden**

Sequentielle
Typen
Änderbare
Sequenzen

Standard-
methoden
des
String-Typs

- Wir hatten gesehen, dass es nicht nur vordefinierte Funktionen und Operationen auf den Objekten der Standardtypen gibt (die ja auch wieder durch magische Methoden implementiert sind), sondern auch **Methoden**.
- Wir wollen diese Methoden für alle Typen (bis auf Sets und Dicts) durchgehen, und dabei einige neue kennen lernen.
- Speziell für Strings gibt es einen ganzen Zoo zur Stringverarbeitung.

Methoden für sequentielle Objekte

Sequentielle Typen sind `str`, `bytes`, `bytearray`, `list`, und `tuple`.

- `s.index(value, start, stop)`:
start und stop sind optionale Parameter.
Sucht in der Sequenz (bzw. in `s[start:stop]`) nach einem Objekt mit Wert `value`. Liefert den Index des ersten Treffers zurück. Erzeugt eine Ausnahme, falls kein passendes Element existiert.
- `s.count(value)`:
Liefert die Zahl der Elemente in der Sequenz, die gleich `value` sind.

Python-Interpreter

```
>>> [1, 2, 3, 4, 5].index(3)
2
>>> [1, 2, 3, 2, 4, 5, 2].count(2)
3
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden
**Sequentielle
Typen**
Änderbare
Sequenzen

Standard-
methoden
des
String-Typs

Methoden von list und bytearray: Einfügen

Das Objekt wird modifiziert, Rückgabewert ist None:

- `s.append(element)`:
Hängt ein Element an die Liste an.
Äquivalent zu `s += [element]`, aber effizienter.
- `s.extend(seq)`:
Hängt die Elemente einer Sequenz an die Liste an.
Äquivalent zu `s += seq`.
- `s.insert(index, element)`:
Fügt `element` vor Position `index` in die Liste ein.

Python-Interpreter

```
>>> s = [1, 2, 3]; s.extend(['a', 'b', 'c']); s
[1, 2, 3, 'a', 'b', 'c']
>>> s.insert(3, 'X'); s
[1, 2, 3, 'X', 'a', 'b', 'c']
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden
Sequentielle
Typen
**Änderbare
Sequenzen**

Standard-
methoden
des
String-Typs

Methoden von list und bytearray: Entfernen

Das Objekt wird modifiziert.

- `s.pop()`:
Entfernt das letzte Element und liefert es zurück.
- `s.pop(index)`:
Entfernt das Element an Position `index` und liefert es zurück.
- `s.remove(value)`:
Entfernt das erste Element aus der Liste, das gleich `value` ist und liefert `None` zurück. Entspricht `del s[s.index(value)]`, inklusive eventueller Ausnahmen.

Python-Interpreter

```
>>> s = [1, 2, 3, 4]; s.pop(1)
2
>>> s
[1, 3, 4]
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden
Sequentielle
Typen
**Änderbare
Sequenzen**

Standard-
methoden
des
String-Typs

Methoden von list und bytearray: Umdrehen

Die folgende Methode verändert das betroffene Objekt direkt und liefert None zurück:

- `s.reverse()`:
Dreht die Reihenfolge der Sequenz um; entspricht `s[:] = s[::-1]`.

Python-Interpreter

```
>>> l = [1, 2, 3, 4]
>>> l.reverse()
>>> l
[4, 3, 2, 1]
```

Methode von list: Sortieren

- `l.sort(key=None, reverse=None)`: (nur auf Listen)
Sortiert die Liste `l`. Der Sortieralgorithmus ist stabil, d.h. Elemente die gleichen Wert haben werden in ihrer relativen Anordnung nicht geändert. Damit kann man mehrstufig sortieren!
 - Wird für `reverse=True` angegebene, wird absteigend statt aufsteigend sortiert.
 - Bei dem `key`-Parameter kann eine Funktion angegeben werden, die für das jeweilige Element den Sortier-Schlüssel berechnet. Beispiel: `str.lower`. Mit Hilfe des Moduls `operator` kann man nach einfach nach dem *i*-ten Element sortieren lassen

Python-Interpreter

```
>>> from operator import itemgetter
>>> l = [('john', 15), ('jane', 12), ('dave', 10)]
>>> l.sort(key=itemgetter(1)); l
[('dave', 10), ('jane', 12), ('john', 15)]
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden
Sequentielle
Typen
Änderbare
Sequenzen

Standard-
methoden
des
String-Typs

Sortieren und Umdrehen von unveränderlichen Sequenzen

- Da Tupel und Strings unveränderlich sind, gibt es für sie auch keine mutierenden Methoden zum Sortieren und Umdrehen.
- Es gibt dafür 2 Funktionen:
- `sorted(seq, key=None, reverse=None)`:
Liefert eine Liste, die dieselben Elemente hat wie `seq`, aber (stabil) sortiert ist. Es gilt das über `list.sort` Gesagte.
- `reversed(seq)`:
Generiert die Elemente von `seq` in umgekehrter Reihenfolge.
Liefert wie `enumerate` einen *Iterator* und sollte genauso verwendet werden.

Standardmethoden des String-Typs

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

**Standard-
methoden
des
String-Typs**

String-Methoden: Suchen

start und stop sind *optionale* positionale Parameter:

- `s.index(substring, start, stop)`:
Liefert analog zu `list.index` den Index des ersten Auftretens von `substring` in `s`. Im Gegensatz zu `list.index` kann ein *Teilstring* angegeben werden, nicht nur ein einzelnes Element.
- `s.rindex(substring, start, stop)`:
Ähnlich `index`, aber von rechts suchend.
- `s.find(substring, start, stop)`:
Wie `s.index`, erzeugt aber keine Ausnahme, falls `substring` nicht in `s` enthalten ist, sondern liefert dann `-1` zurück.
- `s.rfind(substring, start, stop)`:
Die Variante von rechts suchend.

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs

- `s.count(substring, start, stop)`:
start und stop sind optionale positionale Parameter.
Berechnet, wie oft substring als (nicht-überlappender) Teilstring in s enthalten ist.
- `s.replace(old, new, count)`:
count ist ein optionaler positionaler Parameter.
Ersetzt im Ergebnis überall den Teilstring old durch new.
Wird das optionale Argument angegeben, werden maximal count Ersetzungen vorgenommen.
Es ist kein Fehler, wenn old in s seltener oder gar nicht auftritt.

String-Methoden: Zusammenfügen

- `s.join(seq)`:
seq muss eine Sequenz (z.B. Liste) von Strings sein.
Berechnet `seq[0] + s + seq[1] + s + ... + s + seq[-1]`, aber viel effizienter.

Häufig verwendet für Komma-Listen und Verkettung vieler Strings:

Python-Interpreter

```
>>> ", ".join(["ham", "spam", "egg"])  
'ham, spam, egg'
```

```
>>> "".join(['List', 'With', 'Many', 'Strings'])  
'ListWithManyStrings'
```

String-Methoden: In Worte aufteilen

- `s.split()`:
Liefert eine Liste aller Wörter in `s`, wobei ein ‚Wort‘ ein Teilstring ist, der von **Whitespace** (Leerzeichen, Tabulatoren, Newlines etc.) umgeben ist.
- `s.split(separator)`:
`separator` muss ein String sein und `s` wird dann an den Stellen, an denen sich `separator` befindet, zerteilt. Es wird die Liste der Teilstücke zurückgeliefert, wobei anders als bei der ersten Variante leere Teilstücke in die Liste aufgenommen werden.

Python-Interpreter

```
>>> " 1 2 3 ".split()
['1', '2', '3']
>>> "1,,2".split(",")
['1', '', '2']
```

String-Methoden: In Zeilen aufteilen

- `s.splitlines(keepends=None)`:
Liefert eine Liste aller Zeilen in `s`, wobei eine ‚Zeile‘ ein Teilstring ist, der von Newlines umgeben ist. Wird für den optionalen Parameter `keepends True` angegeben, so werden die Newline-Zeichen erhalten

Python-Interpreter

```
>>> " 11\n 22\n 3 4 5 6 ".splitlines()  
[' 11', ' 22', ' 3 4 5 6 ']
```

```
>>> " 11\n 22\n 3 4 5 6 ".splitlines(True)  
[' 11\n', ' 22\n', ' 3 4 5 6 ']
```

String-Methoden: Zerlegen

- `s.partition(sep)`:
Zerlegt `s` in drei Teile. Von links suchend wird nach `sep` gesucht. Wird `sep` in `s` gefunden, wird ein Tupel zurück gegeben, bei dem der erste Teil der Substring bis `sep` ist, dann kommt `sep` und dann der rechte Teilstring. Ansonsten wird `s` als erste Komponente zurück gegeben.
- `s.rpartition(sep)`:
Die Variante, bei der von rechts gesucht wird.

Python-Interpreter

```
>>> "links mitte mitte rechts".partition("mitte")
('links ', 'mitte', ' mitte rechts')
>>> "links mitte mitte rechts".partition("oben")
('links mitte mitte rechts', '', '')
>>> "links mitte mitte rechts".rpartition("mitte")
('links mitte ', 'mitte', ' rechts')
```

String-Methoden: Zeichen abtrennen

- `s.strip()`, `s.lstrip()`, `s.rstrip()`:
Liefert `s` nach Entfernung von Whitespace an den beiden Enden (bzw. am linken bzw. am rechten Rand).
- `s.strip(chars)`, `s.lstrip(chars)`, `s.rstrip(chars)`:
Wie die erste Variante, trennt aber keine Whitespace-Zeichen ab, sondern alle Zeichen, die in dem String `chars` auftauchen.

Python-Interpreter

```
>>> " a lot of   blanks   ".strip()
'a lot of   blanks'
>>> " a lot of   blanks   ".lstrip()
'a lot of   blanks   '
>>> "banana".strip("ba")
'nan'
```


String-Methoden: Groß- und Kleinschreibung

- `s.capitalize()`:
Erster Buchstabe des Strings wird Großbuchstabe, alle anderen Kleinbuchstaben.
- `s.lower()`:
Ersetzt im Ergebnis alle Groß- durch Kleinbuchstaben.
- `s.upper()`:
Ersetzt im Ergebnis alle Klein- durch Großbuchstaben.
- `s.casefold()`:
Transformiert alles in Kleinbuchstaben (wie `lower`) und macht noch andere Ersetzungen wie „ß“ in „ss“. Speziell für Groß-/Kleinschreibungs-freie Vergleiche gedacht.
- `s.swapcase()`:
Großbuchstaben werden klein, Kleinbuchstaben groß.
- `s.title()`:
Jedes einzelne Wort beginnt mit einem Großbuchstaben, gefolgt von Kleinbuchstaben.

String-Methoden: Eigenschaften (1)

- `s.isalnum()`:
True, wenn alle Zeichen in `s` Ziffern oder Buchstaben sind
- `s.isalpha()`:
True, wenn alle Zeichen in `s` Buchstaben sind
- `s.isdigit()`:
True, wenn alle Zeichen in `s` Ziffern sind
- `s.islower()`:
True, wenn alle Buchstaben in `s` Kleinbuchstaben sind
- `s.isupper()`:
True, wenn alle Buchstaben in `s` Großbuchstaben sind

Python-Interpreter

```
>>> 'This string contains 1 word'.isalnum()  
False  
>>> '123'.isdigit()  
True
```

String-Methoden: Eigenschaften (2)

- `s.isspace()`:
True, wenn alle Zeichen in `s` *Whitespace* sind.
- `s.istitle()`:
True, wenn alle Worte in `s` groß geschrieben sind.
- `s.startswith(prefix, start, stop)`:
start und stop sind optional.
True, wenn `s` (bzw. `s[start:stop]`) mit `prefix` beginnt.
- `s.endswith(suffix, start, stop)`:
start und stop sind optional.
True, wenn `s` (bzw. `s[start:stop]`) mit `suffix` endet.

Python-Interpreter

```
>>> 'This Is A Title'.istitle()
True
>>> 'This Is A Title'.endswith('Title')
True
```

String-Methoden: Textformatierung

Bei allen Methoden ist `fillchar` ein optionaler positionaler Parameter:

- `s.center(width, fillchar):`
Zentriert `s` in einer Zeile der Breite `width`.
- `s.ljust(width, fillchar):`
Richtet `s` in einer Zeile der Breite `width` linksbündig aus.
- `s.rjust(width, fillchar):`
Richtet `s` in einer Zeile der Breite `width` rechtsbündig aus.
- `s.zfill(width):`
Richtet `s` in einer Zeile der Breite `width` rechtsbündig aus, indem links mit Nullen aufgefüllt wird.

Python-Interpreter

```
>>> "Python".center(30, '*')
'*****Python*****'
>>> '123'.zfill(6)
'000123'
```

String-Methoden: Decoding und Encoding

- `s.encode(encoding)`:
Übersetzt einen String in eine Sequenz von Bytes (bytes) unter Benutzung des Encodings `encoding` (z.B., `ascii`, `latin9`, `utf-8`, `cp1250`).
- `b.decode(encoding)`:
Übersetzt bytes in einen String Unter Benutzung des angegebenen Encodings.

Python-Interpreter

```
>>> 'abc€'.encode('latin9')
b'abc\xa4'
>>> 'abc€'.encode('utf8')
b'abc\xe2\x82\xac'
>>> b'abc\xa4'.decode('latin9')
'abc€'
```

String-Methoden: Alternative Stringformatierung, Tabs und Übersetzung

- `s.format(*args, **kwargs)`:
Ermöglicht eine sehr komfortable, alternative Stringformatierung.
- `s.expandtabs(tabsize)`:
Expandiert Tabs, wobei der optionale Parameter `tabsize` den Default-Wert 8 hat.
- `s.translate(map)`:
Transformiert einen String mit Hilfe eines Dictionaries `map`, dessen Schlüssel Unicode-Werte sind und dessen Werte, Unicode-Werte Strings, oder `None` sein können. Bei `None` wird das Zeichen gelöscht, sonst wird ersetzt.

Python-Interpreter

```
>>> m = {ord('a'):'A',ord('b'):None,ord('c'):'XY'}
>>> 'abcd'.translate(m)
'AXYd'
```

Informatik I

Bernhard
Nebel

Methoden
für Klassen

Typen sind
Klassen

Standard-
methoden

Standard-
methoden
des
String-Typs