Programmieren in Python

9. Klassen

Robert Mattmüller

Albert-Ludwigs-Universität Freiburg

Handlungsplanungs-Praktikum
Wintersemester 2010/2011

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Eigene Klassen definieren

- In dieser Lektion befassen wir uns mit der Erstellung eigener Klassen.
- Als Illustration dient dabei eine einfache (Stamm-) Baumklasse, anhand derer wir das Phänomen der Veronkelung untersuchen.

- Klassen werden mit dem Schlüsselwort `class` definiert, das einen eingerückten Block eröffnet. Innerhalb dieses Blocks werden die Methoden der Klasse (und evtl. weitere Attribute) definiert.
- Methoden notiert man nicht anders als Funktionen. Tatsächlich ist der Unterschied zwischen Funktionen und Methoden in Python relativ gering und es ist möglich, zur Laufzeit das eine in das andere zu konvertieren.

Klassen: Beispiel

Eine einfache Klassendefinition sieht etwa so aus:

tree.py

```
class Tree(object):
  def __init__(self, data, children=()):
    self.data = data
    self.children = children
  def prettyprint(self, indent=""):
    print(indent + self.data)
    for child in self.children:
      child.prettyprint(indent + "  ")
```

Wir entnehmen der Definition der Klasse `Tree`,

- dass diese von der Klasse `object` abgeleitet ist,
- dass sie einen Konstruktor aufweist, der zwei Parameter `data` und `children` hat, von denen der letzte optional ist, sowie
- dass sie über eine Methode `prettyprint` verfügt.

Vererbung und Angabe von Basisklassen

- Basisklassen werden im Kopf der Klassendefinition in Klammern hinter dem Namen der Klasse angegeben:
  - `class MyClass(BaseClass)`
- Mehrfachvererbung ist möglich:
  - `class MyClass(OneBase, AnotherBase)`
- Es ist auch möglich, von keiner Klasse abzuleiten:
  - `class MyClass`

  oder
  - `class MyClass(object)`

Ein zweiter Blick auf `Tree`

tree.py

```
class Tree(object):
  def __init__(self, data, children=()):
    self.data = data
    self.children = children
  def prettyprint(self, indent=""):
    print(indent + self.data)
    for child in self.children:
      child.prettyprint(indent + "  ")
```

- `__init__` und `prettyprint` sind Methoden von `Tree`. Methoden von Klassen in Python haben keinen *impliziten* this-Parameter wie in C oder Java, sondern erhalten das betroffene Objekt *explizit* als erstes Argument übergeben.
- Damit entspricht `t.prettyprint()` dem Aufruf der Funktion `prettyprint` mit `self = t` und `indent = ""`.

Veronkelungs-Beispiel

family.py

```
from tree import Tree
nephews = Tree("Tick"), Tree("Trick"), Tree("Track")
donald = Tree("Donald", nephews)
daisy, gustav =  Tree("Daisy"), Tree("Gustav")
dagobert = Tree("Dagobert", (donald, daisy, gustav))
```

Python-Interpreter

```
>>> import family
>>> family.dagobert.prettyprint()
Dagobert
  Donald
    Tick
    Trick
    Track
  Daisy
  Gustav
```

Methoden vs. Funktionen und der Parameter `self`

- Das erste Argument einer Methode bezeichnet das Objekt, für das die Methode aufgerufen wird. Es wird üblicherweise `self` genannt.
- Dies ist nicht vorgeschrieben: Jeder andere Variablenname könnte ebenso verwendet werden (was aber niemand macht).
- Tatsächlich sind Methoden von Klassen mit Funktionen in Modulen fast identisch: Statt `t.prettyprint()` könnten wir auch mit der normalen Funktionssyntax `Tree.prettyprint(t)` schreiben.
- `Tree.prettyprint` unterscheidet sich von einer normalen Funktion nur dadurch, dass geprüft wird, ob `t` tatsächlich eine Instanz der Klasse `Tree` oder einer abgeleiteten Klasse ist; andernfalls wird eine Ausnahme erzeugt.

Zugriff auf die Instanz innerhalb der Methode

- Während man in C++ oder Java `attribute` statt `this->attribute` bzw. `this.attribute` schreiben kann, ist in Python immer der Zugriff über `self.attribute` vonnöten.
- Da Variablen nicht deklariert werden müssen, wäre sonst auch völlig unklar, ob `attribute` eine Instanzvariable des Objekts oder eine lokale Variable der Methode sein sollte.

Die Methode `__init__`

Betrachten wir die Methode `Tree.__init__`:

```
def __init__(self, data, children=()):
  self.data = data
  self.children = children
```

- Die meisten Klassen definieren eine Methode `__init__`, die den *Konstruktor* der Klasse bildet.
- Bei Aufruf von `MyClass(args)` wird intern zunächst ein „leeres“ Objekt `obj` der Klasse `MyClass` vorbereitet und anschließend der Konstruktor `MyClass.__init__(obj, args)` aufgerufen, um `obj` zu initialisieren. Das Ergebnis der Operation ist das Objekt `obj`.
- Beispielsweise erzeugt `Tree("Tick")` ein Baum-Objekt `obj` mit `obj.data == "Tick"` und `obj.children == ()`.
- Konstruktoren müssen oft die Konstruktoren ihrer Basisklassen aufrufen. Notation: `BaseClass.__init__(self, args)`.

Klassen als Objekte und `obj.__class__`

- Nicht nur Instanzen von Klassen, auch Klassen selbst sind Objekte mit eigenen Attributen und allem, was dazu gehört:

Python-Interpreter

```
>>> from tree import Tree
>>> print(Tree)
<class 'tree.Tree'>
>>> print(id(Tree))
135936652
```

- Mit `obj.__class__` erhält man das Klassenobjekt der Klasse, deren Instanz `obj` ist.
- Klassen sind meistens Instanzen der Klasse `type`. Klassen wie `type`, deren Instanzen Klassen sind, nennt man Metaklassen.
- Mit Metaklassen kann man viele lustige Dinge anstellen, aber das ist ein deutlich fortgeschritteneres Thema.

Wie kommen die Attribute in die Objekte? (1)

- Hier stellt sich die Frage, was es eigentlich genau bedeutet, wenn wir `self.data` oder `self.children` schreiben. Kann man jedem Objekt beliebige Attribute zuweisen?
- Die Antwort darauf lautet in der Regel: ja.
- Es sind zwar auch Objekte möglich, die nur bestimmte Attribute haben können oder deren Attribute nicht neu gebunden werden können (wie z.B. die `real` und `imag` bei `complex`-Objekten), aber dies ist überlicherweise nur für Python-Klassen der Fall, die in C implementiert und auf absolute Effizienz getrimmt sind.
- „Normale“ Instanzen können beliebige Attribute haben und dynamisch während ihrer Lebenszeit neue Attribute erhalten oder alte verlieren. Es ist also nicht nötig, alle Attribute im Konstruktor zu initialisieren; oft ist es aber dennoch sinnvoll.

Wie kommen die Attribute in die Objekte? (2)

- Intern werden die Attribute von Objekten mit einem bekannten Mechanismus verwaltet:

Python-Interpreter

```
>>> from tree import Tree
>>> track = Tree("Track")
>>> print(track.__dict__)
{'data': 'Track', 'children': []}
```

- Jedes Objekt hat ein internes Dictionary, das alle seine Attribute enthält und auf das mit `obj.__dict__` zugegriffen werden kann.
- Ebenso wie `globals()` kann man den Inhalt dieses Dictionaries nach Belieben verändern, um die Attribute des Objekts zu modifizieren.

Attribut-Zugriff und `__dict__`

Intern werden Attribut-Zugriffe bei Objekten also auf Dictionary-Operationen umgesetzt. Dabei gelten folgende Beziehungen:

- `print(obj.attr)` $\Rightarrow$ `print(obj.__dict__["attr"])`
- `obj.attr = expr` $\Rightarrow$ `obj.__dict__["attr"] = expr`
- `del obj.attr` $\Rightarrow$ `del obj.__dict__["attr"]`

- Zuweisungen an Objektattribute und Entfernen von Objektattributen sind damit schon (naja, fast) vollständig erklärt.
- Das Auslesen von Objektattributen (`print(obj.attr)`) muss aber noch etwas geheimnisvoller sein.
  Sonst könnte `t.prettyprint()` nicht funktionieren, denn schließlich gilt `"prettyprint" not in t.__dict__`.

Wie ein Objekt sein Attribut findet (1)

- Objekte sind mehr als nur komisch notierte Dictionaries, weil sie neben ihren „eigenen“ Attributen in `obj.__dict__` auch auf die Attribute ihrer Klasse zugreifen können.
- Auch Klassen haben ein entsprechendes Dictionary, wie wir unschwer erkennen können:

Python-Interpreter

```
>>> from tree import Tree
>>> print(Tree.__dict__)
{'prettyprint': <function prettyprint at 0x402f0b8c>,
 '__init__': <function __init__ at 0x402f0b54>,
 ...}
```

- Die Auslassungspunkte stehen für interne Attribute der Klasse.

Wie ein Objekt sein Attribut findet (2)

Der Mechanismus für einen lesenden Zugriff auf `obj.attr` im Einzelnen:

- Zunächst wird überprüft, ob `obj.__dict__` den Schlüssel `"attr"` enthält. Falls ja, ist die Suche beendet.
- Ansonsten wird als nächstes in `obj.__class__.__dict__` nach `"attr"` gesucht.
- Wird auch dort nichts gefunden, wird rekursiv in den Basisklassen von `obj.__class__` weitergesucht, bis das Attribut gefunden wird oder alle Klassen erfolglos durchprobiert wurden.

- Es stellt sich die Frage, in welcher Reihenfolge die Basisklassen bei Mehrfachvererbung besucht werden; dieses Thema wie auch andere für die Mehrfachvererbung spezifische Aspekte lassen wir aus.
- Die Basisklassen einer Klasse `Class` lassen sich zur Laufzeit mithilfe von `Class.__bases__` ermitteln.

Nicht behandelte Aspekte von Klassen

Das ist alles, was ich zur grundlegenden Funktionsweise von Klassen und Objekten sagen möchte.

Einige weiterführende Themen, für die leider keine Zeit ist, damit man weiß, dass es diese Dinge gibt:

- Metaklassen
- In C implementierte Klassen
- Gebundene Methoden (*bound methods*)
- Mehrfachvererbung
- statische Methoden und Klassenmethoden
- berechnete Attribute (*properties*)

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Magische Methoden

- Methoden wie `__init__`, deren Namen mit zwei Unterstrichen beginnen und enden, bezeichnet man als *magisch*.
- Daneben gibt es noch eine Vielzahl an weiteren magischen Methoden, die z.B. verwendet werden, um Operatoren wie + und % für eigene Klassen zu definieren.
- Magische Methoden wie `__add__` sind nicht prinzipiell anders als andere Methoden; der Grund dafür, warum man beispielsweise mit `__add__` das Verhalten der Addition beeinflussen kann, liegt einfach darin, dass Python intern versucht, beim Addieren die Methode `__add__` aufzurufen.

Magische Methoden: Beispiel Mitgliedschaftstest

- Wir wollen testen können, ob unser Baum einen bestimmten Namen enthält, und wir wollen diesen Test als `"name" in mytree` schreiben können.
- Dazu definieren wir eine neue Methode innerhalb der `Tree`-Klasse:

`tree.py` [Forts.]

```
def __contains__(self, data):
  if self.data == data:
    return True
  for child in self.children:
    if data in child:
      return True
  return False
```

Magische Methoden: Beispiel Mitgliedschaftstest (2)

Python-Interpreter

```
>>> import family
>>> print("Track" in family.dagobert)
True
>>> print("Goofy" in family.donald)
False
```

Magische Methoden: Übersicht

Wir können leider nicht alle magischen Methoden im Detail behandeln, aber zumindest sollten wir einen guten Überblick bekommen können.
Wir unterscheiden zwischen drei Arten von magischen Methoden:

- Allgemeine Methoden: verantwortlich für Objekterzeugung, Ausgabe und ähnliche grundlegende Dinge.
- Numerische Methoden: verantwortlich für Addition, Bitshift und ähnliches
- Container-Methoden: verantwortlich für Indexzugriff, Slicing und ähnliches

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Allgemeine magische Methoden

Die allgemeinen magischen Methoden werden weiter unterteilt:

- Konstruktion und Destruktion: `__init__`, `__new__`, `__del__`
- Vergleich und Hashing: `__eq__`, `__ne__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__hash__`, `__bool__`
- String-Konversion: `__str__`, `__repr__`, `__format__`
- Verwendung als Funktion: `__call__`
- Attributzugriff: `__getattribute__`, `__getattr__`, `__setattr__`, `__delattr__`

Konstruktion und Destruktion

- `__init__` haben wir bereits behandelt.
- `__new__` ist im Wesentlichen für fortgeschrittene Anwendungen mit Nicht-Python-Klassen interessant und wird von uns übergangen.
- `__del__` wird aufgerufen, wenn das Objekt aus dem Speicher gelöscht wird, weil es über keinen Namen mehr erreichbar ist, also Pythons Variante von Destruktoren. Die Methode erhält keine Argumente und ihr Rückgabewert wird ignoriert.

Vergleich und Hashing: `__eq__`, `__ne__`

- `obj.__eq__(other)`:
  Wird bei Tests `obj == other` aufgerufen.
  Der Rückgabewert der Methode ist das Ergebnis des Vergleichs.
- `obj.__ne__(other)`:
  Wird bei Tests `obj != other` aufgerufen,
  Der Rückgabewert der Methode ist das Ergebnis des Vergleichs.

- Definiert man diese Methoden nicht, werden eigene Objekte per `id()` verglichen, d.h. `x == y` gdw. `x is y`.
- Aufruf von `!=` gibt automatisch das Gegenteil vom Aufruf von `==` zurück, außer wenn `==` mit `NotImplemented` antwortet. Es reicht also, `obj.__eq__(other)` zu implementieren (Achtung: in Python 2.x muss man `obj.__ne__(other)` noch explizit implementieren).

Vergleich und Hashing: `__ge__`, `__gt__`, `__le__`,`__lt__`

- `obj.__ge__(other)`:
  Wird bei Tests `obj >= other` aufgerufen.
  Der Rückgabewert der Methode ist das Ergebnis des Vergleichs.

  Bei Tests `other <= obj` wird die Methode ebenfalls verwendet, falls `other` über keine `__le__`-Methode verfügt.
- `obj.__gt__(other)`, `obj.__le__(other)`, `obj.__lt__(other)`:
  Wird analog für die Vergleiche `obj > other` bzw. `obj <= other` bzw. `obj < other` aufgerufen.

Vergleich und Hashing: `__hash__`

- `obj.__hash__()`:
  Liefert einen Hashwert für `obj` bei Verwendung in einem Dictionary.
  Wird von der Builtin-Funktion `hash` verwendet.
- Damit Hashing funktioniert, muss immer gelten:

$$\texttt{x == y} \implies \texttt{hash(x) == hash(y).}$$

  Daher muss man in der Regel auch `__eq__` implementieren, wenn man `__hash__` implementiert.
- Eigene Klassen, die `__hash__` nicht implementieren, werden nach `id(obj)` gehasht (d.h. nur identische Objekte gelten bzgl. Hashing als gleich).

Vergleich und Hashing: `__bool__`

- `obj.__bool__()`:
  Wird von `bool(obj)` und damit auch bei `if obj:` und `while obj:` aufgerufen. Sollte `True` zurückliefern, wenn das Objekt als „wahr“ einzustufen ist, sonst `False`.
- Ist diese Methode nicht implementiert, dafür aber das später diskutierte `__len__`, dann wird genau dann `True` geliefert, wenn `__len__` einen von 0 verschiedenen Wert liefert.
- Ist weder diese Methode noch `__len__` implementiert, gilt das Objekt immer als wahr.

String-Konversion: `__str__`, `__repr__`, `__format__`

- `obj.__str__()`:
  Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen, z.B. bei `print(obj)`, `str(obj)` und `"%s" % obj`.

  `__str__` sollte eine menschenlesbare Darstellung erzeugen.
- `obj.__repr__()`:
  Wird aufgerufen, um eine String-Darstellung von `obj` zu bekommen, z.B. bei Angabe von `obj` im interaktiven Interpreter sowie bei `repr(obj)` und `"%r" % obj`.

  `__repr__` sollte eine möglichst exakte (für Computer geeignete) Darstellung erzeugen, idealerweise eine, die korrekte Python-Syntax wäre, um dieses Objekt zu erzeugen.
- `obj.__format__(format_spec)`:
  Wird von der `format`-Funktion aufgerufen, um eine formatierte String-Darstellung eines Objekts zu erzeugen.

Verwendung als Funktion: `__call__`

- `obj.__call__(*args, **kwargs)`:
  Wird aufgerufen, wenn das Objekt wie eine Funktion verwendet wird, z.B. als `obj("spam", "spam", "spam")`.
  Die Signatur von `__call__` kann beliebig sein.

  Beispiel:

unfit_adder.py

```
class UnfitAdder(object):
  def __init__(self):
    self.fitness = 10
  def __call__(self, x, y):
    if not self.fitness:
      return "Sorry, I'm too exhausted."
    if x + y >= 1000:
      self.fitness -= 1
    return x + y
```

Attributzugriff: `__getattribute__`

Mit den Attributzugriffsmethoden kann man verschiedene Aspekte der Notation `obj.attr` beeinflussen.

- `obj.__getattribute__(attrname)`:
  Bei (lesendem) Zugriff `obj.attr` wird `obj.__getattribute__("attr")` aufgerufen und das Ergebnis des Methodenaufrufs als Wert verwendet.
- Durch Implementation von `__getattribute__` kann man beispielsweise Attributszugriffe zum Debugging protokollieren.
- In vielen Fällen wird man die eigentliche Arbeit (Suche im Dictionary des Objekts und seiner Klasse usw.) der Standardimplementation überlassen, indem man `object.__getattribute__(attr)` aufruft.
- Da Attributzugriffe sehr häufig erfolgen und standardmäßig sehr effizient implementiert sind, können komplizierte Attributszugriffsmethoden starken Einfluss auf die Laufzeit haben.

Attributzugriff: `__getattr__`

- `obj.__getattr__(attrname)`:
  Wird von der Standardimplementation von `__getattribute__` verwendet, wenn der normale Zugriff erfolglos ist.
  Die `object`-Implementation erzeugt immer einen `AttributeError`.
- Kann z.B. verwendet werden, um noch nicht verwendete Attribute automatisch mit einem Default-Wert zu initialisieren:

getattr_example.py

```
class ZeroingClass(object):
  def __getattr__(self, attr):
    self.__dict__[attr] = 0
    return 0
```

__getattr__: Beispiel

Python-Interpreter

```
>>> from getattr_example import ZeroingClass
>>> spam = ZeroingClass()
>>> print(spam.egg)
0
>>> spam.sausage += 2
>>> print(spam.sausage)
2
>>> spam.baked_beans = 10
>>> print(spam.baked_beans)
10
>>> print(spam.__dict__)
{'sausage': 2, 'egg': 0, 'baked_beans': 10}
```

Attributzugriff: `__setattr__`

- `obj.__setattr__(attrname, value)`:
  Zum Binden von Attributen `obj.attr = value` wird `obj.__setattr__("attr", value)` aufgerufen.
- Innerhalb von `__setattr__` sollte man nicht `self.var = x` schreiben, sonst erhält man eine Endlos-Rekursion!
  Stattdessen modifiziert man `self.__dict__` direkt.
- Die Verwendung von `self.__dict__` führt *nicht* zur Rekursion, da `__dict__` kein „richtiges“ Attribut ist, sondern nur zufällig eine ähnliche Notation verwendet.

  Dass `__dict__` (ebenso wie im Übrigen `__class__`) kein Attribut ist, erkennt man auch daran, dass diese nicht in `__dict__` enthalten sind.

Attributzugriff: `__delattr__`

- `obj.__delattr__(attrname)`:
  Zum Entfernen von Attributen mit `del obj.attr` wird `obj.__delattr__("attr")` aufgerufen.
- Beispiel:

delattr_example.py

```
class DeletionForbidden(object):
  def __delattr__(self, attr):
    print("Don't you dare!")
```

`__delattr__`: Beispiel

Python-Interpreter

```
>>> from delattr_example import DeletionForbidden
>>> myobj = DeletionForbidden()
>>> myobj.spam = 10
>>> print(myobj.spam)
10
>>> del myobj.spam
Don't you dare!
>>> print(myobj.__dict__)
{'spam': 10}
>>> print(myobj.spam)
10
```

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Numerische Methoden

- Bei Operatoren wie +, *, - oder / verhält sich Python wie folgt (am Beispiel +):
- Zunächst wird versucht, die Methode `__add__` des linken Operanden mit dem rechten Operanden als Argument aufzurufen.
- Wenn `__add__` mit dem Typ des rechten Operanden nichts anfangen kann, kann sie die spezielle Konstante `NotImplemented` zurückliefern. Dann wird versucht, die Methode `__radd__` des rechten Operanden mit dem linken Operanden als Argument aufzurufen.
- Wenn dies auch nicht funktioniert, schlägt die Operation fehl.

Numerische Methoden (2)

In Code funktioniert der Mechanismus etwa so:

```
def plus(x, y):
  result = NotImplemented
  try:
    result = x.__add__(y)
  except AttributeError:
    pass
  if result is NotImplemented:
    try:
      result = y.__radd__(x)
    except AttributeError:
      pass
  if result is NotImplemented:
    raise TypeError("some diagnostic text")
  return result
```

Beispiel: Subtraktion

Wir definieren (etwas willkürlich) die Subtraktion von Bäumen als „Abziehen“ von Teilbäumen mit einem bestimmten Namen:

tree.py [Forts.]

```
  def __sub__(self, key):
    if key not in self:
      raise ValueError("Cannot subtract %r" % key)
    new_children = []
    for child in self.children:
      if child.data == key:
        continue
      elif key in child:
        new_children.append(child - key)
      else:
        new_children.append(child)
    return Tree(self.data, tuple(new_children))
```

Beispiel: Subtraktion (2)

Python-Interpreter

```
>>> from family import dagobert
>>> (dagobert - "Trick").prettyprint()
Dagobert
  Donald
    Tick
    Track
  Daisy
  Gustav
>>> (dagobert - "Donald").prettyprint()
Dagobert
  Daisy
  Gustav
```

Beispiel: Subtraktion (3)

Python-Interpreter

```
>>> from family import dagobert
>>> (dagobert - "Goofy").prettyprint()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "tree.py", line 18, in __sub__
    raise ValueError("Cannot subtract %r" % key)
ValueError: Cannot subtract 'Goofy'
```

Magische Methoden für Grundrechenarten

Hier sehen wir die Zuordnung zwischen den Grundrechenarten und den Namen der zugehörigen magischen Methoden:

- `+`: `__add__` und `__radd__`
- `-`: `__sub__` und `__rsub__`
- `*`: `__mul__` und `__rmul__`
- `/`: `__truediv__` und `__rtruediv__`
- `//`: `__floordiv__` und `__rfloordiv__`
- `%`: `__mod__` und `__rmod__`
- unäres `-`: `__neg__` (`-obj` entspricht `obj.__neg__()`).

Magische Methoden für Boole'sche Operatoren

Hier das gleiche für die Boole'schen Operatoren:

- `&`: `__and__` und `__rand__`
- `|`: `__or__` und `__ror__`
- `^`: `__xor__` und `__rxor__`
- `<<`: `__lshift__` und `__rlshift__`
- `>>`: `__rshift__` und `__rrshift__`
- $\sim$ (unär): `__invert__`

Magische Methoden für In-Place-Operationen

- Bei Klassen, deren Instanzen veränderlich sein sollen, wird man in der Regel zusätzlich zu Operatoren wie + auch Operatoren wie += unterstützen wollen.
- Dazu gibt es zu jeder magischen Methode für binäre Operatoren wie `__add__` auch eine magische Methode wie `__iadd__`, die das Objekt selbst modifizieren und `self` zurückliefern sollte. (Der Rückgabewert ist wichtig; die Gründe dafür sind etwas technisch.)
- Implementiert man `__add__`, aber nicht `__iadd__`, dann ist `x += y` äquivalent zu `x = x + y`.

Klassen

Überblick über diese Lektion:

- Grundlagen von Klassen
- Magische Methoden
- Allgemeine magische Methoden
- Numerische magische Methoden
- Magische Container-Methoden

Container-Methoden (1)

Mit den Container-Methoden kann man Klassen implementieren, die sich wie `list` oder `dict` verhalten.
Die Container-Methoden im Einzelnen:

- `obj.__len__()`:
  Wird von `len(obj)` aufgerufen.
- `obj.__contains__(item)`:
  Wird von `item in obj` aufgerufen.
- `obj.__iter__()`:
  Wird von `for x in obj` aufgerufen; mehr dazu folgt in der nächsten Lektion.

Container-Methoden (2)

Die Container-Methoden im Einzelnen:

- `obj.__getitem__(key)`:
  Wird bei lesendem Zugriff auf `obj[key]` und `obj[start:stop]` (sowie anderen Slicing-Varianten) aufgerufen.
  Bei Slice-Notation ist `key` eine Instanz der Klasse `slice`.
- `obj.__setitem__(key, value)`:
  Wird analog zu `__getitem__` bei `obj[key] = value` oder `obj[start:stop] = value` aufgerufen.
- `obj.__delitem__(key)`:
  Wird analog zu `__getitem__` bei `del obj[key]` oder `del obj[start:stop]` aufgerufen.

Slice-Objekte

- Slice-Objekte kann man „von Hand“ mit `slice(stop)` oder `slice(start, stop[, step])` erzeugen.
- Normalerweise werden sie aber von Python selbst bei Verwendung der Slice-Notation erzeugt.
- Slice-Objekte haben Attribute `start`, `stop` und `step` entspechend den Konstruktor-Argumenten. Im Konstruktor nicht angegebene Argumente haben den Wert `None`.
- Slice-Objekte sind unveränderlich.

Python-Interpreter

```
>>> x = slice(10)
>>> print(x)
slice(None, 10, None)
>>> print(x.stop, x.step)
10 None
```