Informatik I

15. Zusammengesetzte Daten und Klassen

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

11. Januar 2011

Informatik I

11. Januar 2011 — 15. Zusammengesetzte Daten und Klassen

15.1 Zusammengesetzte Daten

15.2 Gemischte Daten

15.3 Veränderlichkeit von Objekten

15.1 Zusammengesetzte Daten

- Einführung
- Schokokekse
- Syntax einer Klassendefinition
- Klasse testen
- Zugriff auf Attribute
- “Eingabe”
- Ausdrucken
- Objekte als Ausgabe

Zusammengesetzte Daten in Python

- In Scheme haben wir zusammengesetzte Daten (Schokokekse, kartesische Koordinaten) behandelt. Wie lautet hierfür die technische Bezeichnung?
- Records! Records gibt es nicht nur in Scheme, sondern auch in anderen Programmiersprachen (z.B. C). In Python allerdings führen sie höchstens ein Schattendasein.
- Um zusammengesetzte Daten in Python befriedigend modellieren zu können, brauchen wir eine Verallgemeinerung von Records: Klassen.
- Programmierung mit Klassen bezeichnet man als objektorientierte Programmierung.

Objektorientierte Programmierung

- Objektorientierte Programmierung ist ein Programmierparadigma. Python ist eine objektorientierte Programmiersprache.
- Wieso jetzt doch objektorientiert? Sollten wir hier nicht imperative Programmierung lernen?
  "Objektorientiert" und "imperativ" ist kein strikter Gegensatz ("imperativ" und "funktional" schon eher!).
- Objektorientierte Programmierung wird in Informatik II ausgedehnt behandelt und deshalb versuche ich mich hier äußerst zurückzuhalten. Trotzdem brauchen wir sie, um gemischte Daten vernünftig zu modellieren.

Schokokekse

Wir erinnern uns: Ein Schokokeks sei definiert durch seinen Schokoladenanteil und seinen Keksanteil. Z.B.

Komponente	Wert
Schoko:	12g
Keks:	14g

Erinnerung: Schokokekse in Scheme

Die Recorddefinition für Schokokekse in Scheme sah folgendermaßen aus:

```
(define-record-procedures chocolate-cookie
  make-chocolate-cookie chocolate-cookie?
  (chocolate-cookie-chocolate chocolate-cookie-cookie))
```

Die Klasse Schokokeks

Das Pendant in Python sieht folgendermaßen aus:

`cookies.py`

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- `ChocolateCookie` ist eine Klasse. Eine Klasse ist wie ein Typ, aber noch manches mehr ...
- `chocolate` und `cookie` sind die Attribute der Klasse `ChocolateCookie` (in Scheme: Komponenten).
- Ein Datum, das zu einer Klasse gehört (also z.B. ein bestimmter Schokokeks) wird Objekt genannt. Man sagt: das Objekt ist eine Instanz der Klasse.
- Obige Syntax definiert einen Konstruktor, der ebenfalls `ChocolateCookie` heißt.

Konstruktion eines Schokokekses

Python-Interpreter

```
>>> from cookies import *
>>> keks = ChocolateCookie(12, 14)
```

- Soeben haben wir einen Schokokeks konstruiert und der Variablen `keks` diesen Schokokeks zugewiesen.
- Wie sieht dieser Keks aus? Können wir ihn uns "ansehen", ähnlich wie in Scheme

  ```
  #<record:chocolate-cookie 12 14>
  ```

  ?
- Wir erinnern uns: mit dem "Aussehen" ist es bei zusammengesetzten Daten so eine Sache, und den ganzen Keks "ansehen" können wir vorläufig erst einmal nicht ...

Die Syntax einer Klassendefinition

Hier noch einmal die Klassendefinition:
cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- `class` ist ein Schlüsselwort.
- Das Schlüsselwort `def` deutet darauf hin, dass hier eine Funktion `__init__` definiert wird. Das ist auch so, nur haben Funktionen, die innerhalb einer Klassendefinition definiert werden, einen besonderen Namen: sie heißen Methoden.

Die Syntax einer Klassendefinition II

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Eine Methode ist eine Funktion, die "auf einem Objekt rechnet". Das erste Argument der Methode ist daher immer (in Python fest eingebaut!) das Objekt selbst, weswegen, das Argument per Konvention (nicht in Python fest eingebaut!) `self` heißt.
- Eine Methode kann weitere Argumente haben, in unserem Fall `chocolate` und `cookie`.

Die Syntax einer Klassendefinition III

`cookies.py`

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Wenn `self` ein Objekt der Klasse `ChocolateCookie` ist, dann sorgt die Zuweisung

  `self.`*name* `= ...`

  dafür, dass *name* ein Attribut dieser Klasse ist (in anderen Sprachen müsste man das Attribut deklarieren).
  Somit hat `ChocolateCookie` die Attribute `chocolate` und `cookie`.

Die Syntax einer Klassendefinition IV

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- In den Zuweisungen ist das linke Vorkommen von `chocolate` bzw. `cookie` ein Attribut der Klasse, das rechte hingegen ein Argument der Methode.
  Alternative Definition:

cookies.py

```
    class ChocolateCookie:
        def __init__(self, chocolateXX, cookieXX):
            self.chocolate = chocolateXX
            self.cookie = cookieXX
```

Die Syntax einer Klassendefinition V

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Der Methodenname `__init__` ist kein beliebiger vom Programmierer gewählter Name, sondern er hat in Python eine spezielle Bedeutung: Durch die Definition der Methode `__init__` mit $n+1$ Argumenten (in unserem Fall 3) existiert ein Konstruktor mit n Argumenten (in unserem Fall 2).
- Der Name des Konstruktors ist identisch mit dem Klassennamen, also in unserem Fall `ChocolateCookie`.
- Die Argumente des Konstruktors korrespondieren mit den Argumenten von `__init__`, bis auf das erste.

Die Syntax einer Klassendefinition VI

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Die Zeile

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
```

hat also folgenden Effekt: konstruiere ein Objekt der Klasse `ChocolateCookie` und rufe `__init__` mit als erstem Argument das Objekt selbst, und weiteren Argumenten 12 und 14, auf. Dadurch bekommen das `chocolate`- bzw. `cookie`-Attribut die Werte 12 bzw. 14.

Die Syntax einer Klassendefinition VII

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Die Zeile

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
```

hat folgenden Effekt: ...
Der Variablen `keks` wird dieses Objekt zugewiesen.

Die Syntax einer Klassendefinition VIII

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
```

- Schlussbemerkung: All dies ließe sich noch beliebig verkomplizieren.

Klasse testen

- In Scheme ist ein Bestandteil einer Recorddefinition das Sortenprädikat.
- Man würde nun erwarten, dass man analog in Python eine Funktion schreiben kann, die testet, ob ein Objekt in einer bestimmten Klasse ist, also z.B. ob es ein Schokokeks ist.

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> isinstance(keks,ChocolateCookie)
True
>>> isinstance(5,ChocolateCookie)
False
```

- Wir werden dies aber höchstwahrscheinlich nie benötigen!

Zugriff auf die Attribute

- In Scheme hatten wir, um auf die Komponenten eines Records zuzugreifen, die Selektoren.
- In Python geht es so: wenn *obj* ein Objekt ist und *attr* der Name eines Attributs der Klasse des Objekts, dann ist *obj.attr* der Wert des Attributs für das Objekt.

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks.chocolate
12
>>> keks.cookie
14
```

- Direkt auf die Attribute eines Objekts zuzugreifen entspricht nicht ganz der reinen Lehre des objektorientierten Programmierens, aber wir tun es.

Gemischte Daten als Eingabe

- In Scheme hatten wir Prozeduren betrachtet, die Records als Eingabe haben, z.B. die Berechnung des Gewichts eines Kekses oder des Abstands eines kartesischen Punktes vom Ursprung.
- In Python gibt es das Konzept der Methode: eine Funktion, die "auf einem Objekt rechnet".
- Wir definieren jetzt eine Methode zur Berechnung des Gewichts eines Schokokekses.

Die Methode weight

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie

    def weight(self):
        return self.chocolate + self.cookie
```

Das einzige Argument von weight ist das Objekt selbst.

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks.weight()
26
```

Methodendefinitionen allgemein

some_program.py

```
class Class:
    def method(self, arg_1, ..., arg_n):
        ...
```

Wenn *obj* ein Objekt der Klasse *Class* ist, dann wird die Methode *method* für *obj* so aufgerufen:

obj.*method*(arg_1,..., arg_n)

Also nochmals am Beispiel:

Python-Interpreter

```
>>> keks.weight()
26
```

Beim Aufruf hat `weight` 0 Argumente, denn das Objekt selbst ergibt sich ja schon durch `keks`..

Eingabe? Konsumieren?

Zu sagen "Die Methode konsumiert den Keks" passt nicht zur objektorientierten Sichtweise. Wir fragen nicht die Methode: "wieviel wiegt dieser Keks?", wir fragen den Keks: "wieviel wiegst du?".

Kekse auf dem Bildschirm ausgeben

- Bisher können wir nicht den ganzen Keks “ansehen”.
- Also schreiben wir eine Methode, um dies zu tun. Wir nennen diese `prettyprint`.

prettyprint

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def prettyprint(self):
        print("ChocolateCookie",
              self.chocolate,
              self.cookie)
```

Die eingebaute Funktion `print` akzeptiert beliebig viele Argumente und druckt diese nacheinander aus, getrennt durch je ein Leerzeichen.

Verwendung von prettyprint

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks.prettyprint()
ChocolateCookie 12 14
```

Alternative Definition

Natürlich ist die Definition von `prettyprint` Geschmackssache. Man könnte auch definieren:

`cookies.py`

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def uglyprint(self):
        print("#<record:chocolate-cookie",
              self.chocolate,
              self.cookie,
              ">")
```

Verwendung von uglyprint

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks.uglyprint()
#<record:chocolate-cookie 12 14 >
```

Objekte als Ausgabe

Wir betrachten jetzt Funktionen/Methoden, die Objekte als Ausgabe haben.

Mit "Ausgabe" meine ich hier das `return` einer Funktion/Methode, nicht "Bildschirmausgabe".

Kekse verdoppeln

Um einen Keks zu verdoppeln, definieren wir eine Methode:

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def new_double(self):
        return ChocolateCookie(2 * self.chocolate,
                               2 * self.cookie)
```

Die Methode ruft den Konstruktor auf, sie konstruiert also einen neuen Keks.

Verwendung von `new_double`

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks2 = keks.new_double()
>>> keks.prettyprint()
ChocolateCookie 12 14
>>> keks2.prettyprint()
ChocolateCookie 24 28
```

Wir sehen: es gibt jetzt zwei Kekse.

Keks beliebig vervielfachen

Statt `new_double` könnte man eine Methode definieren, die den Keks um einen beliebigen Faktor vergrößert.

Übung!

Die Keks-Klasse im Überblick (ohne uglyprint)

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie

    def weight(self):
        return self.chocolate + self.cookie

    def new_double(self):
        return ChocolateCookie(2 * self.chocolate,
                               2 * self.cookie)

    def prettyprint(self):
        print("ChocolateCookie",
              self.chocolate,
              self.cookie)
```

15.2 Gemischte Daten

Gemischte Daten

- Als nächstes wären gemischte Daten (`mixed` in Scheme) an der Reihe.
- Um diese in Python einzuführen, müssten wir zu sehr im Terrain von Informatik II (Objektorientierte Programmierung) wildern.
- Das tun wir jetzt nicht und wahrscheinlich auch später nicht.

Warum kein `isinstance`?

- Erinnern wir uns: die Funktion `isinstance` testet auf Klassenzugehörigkeit, aber ich sagte: Wir werden dies höchstwahrscheinlich nie benötigen! Warum?

Warum kein `isinstance`?

- Sortenprädikate werden in Scheme im Zusammenhang mit gemischten Daten benötigt, z.B.

```
; Gewicht eines Kekses ermitteln
(: cookie-weight (cookie -> real))
(define cookie-weight
  (lambda (c)
    (cond
     ((chocolate-cookie? c) ...)
     ((cream-jelly-cookie? c) ...))))
```

- Da wir gemischte Daten für Python nicht behandeln, benötigen wir hier also auch kein `isinstance`.
- Selbst wenn wir gemischte Daten behandeln würden, würden wir kein `isinstance` benötigen.

15.3 Veränderlichkeit von Objekten

- Bankkonto
- Schokokekse
- Kein `return`
- Gleichheit vs. Identität
- Objekte kopieren
- Zusammenfassung

Beispiel: Bankkonto

Wir wollen ein sehr einfaches Bankkonto modellieren. Uns interessiert nur der Kontostand, nicht der Kontoinhaber, die Kontonummer ...

account.py

```
class Account:
    def __init__(self, amount):
        self.balance = amount
```

Python-Interpreter

```
>>> from account import *
>>> konto = Account(90)
>>> konto.balance
90
```

Abhebungen

- Wir wollen nun eine Methode schreiben, die eine Abhebung von dem Konto vornimmt.
- Wir gehen davon aus, dass es keinen Überziehungskredit gibt. Die Methode soll `True` zurückgeben, falls die Abhebung funktioniert hat, und `False`, falls sie mangels Deckung nicht durchgeführt wurde.

withdraw

account.py

```
class Account:
    def __init__(self, amount):
        self.balance = amount

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return True
        else:
            return False
```

Beachten Sie das -=.

Verwendung von `withdraw`

Python-Interpreter

```
>>> konto = Account(90)
>>> konto.withdraw(1000)
False
>>> konto.balance
90
>>> konto.withdraw(40)
True
>>> konto.balance
50
>>> konto.withdraw(40)
True
>>> konto.balance
10
```

Mehrere Konten

Natürlich kann es gleichzeitig mehrere Konten geben:

Python-Interpreter

```
>>> dagoberts_konto = Account(1000000)
>>> donalds_konto = Account(500)
>>> dagoberts_konto.withdraw(500000)
True
>>> donalds_konto.withdraw(500000)
False
>>> dagoberts_konto.balance
500000
>>> donalds_konto.balance
500
```

Veränderlichkeit

Die Methode `withdraw` tut etwas, was keine der Methoden für `ChocolateCookie` tut: sie verändert das Objekt mittels einer Zuweisung an ein Attribut.
Vergleichen wir nochmals mit `ChocolateCookie` ...

Erinnerung: Keks-Klasse

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie

    def weight(self):
        return self.chocolate + self.cookie

    def new_double(self):
        return ChocolateCookie(2 * self.chocolate,
                               2 * self.cookie)

    def prettyprint(self):
        print("ChocolateCookie",
              self.chocolate,
              self.cookie)
```

Veränderlichkeit

- Die Methode `__init__` enthält zwei Zuweisungen und "verändert" somit die Werte der Attribute.
- Ansonsten enthält keine Methode Zuweisungen.

Im Gegensatz dazu `Account`:

account.py

```
class Account:
    def __init__(self, amount):
        self.balance = amount

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return True
        else:
            return False
```

Kekse verdoppeln

Wir haben oben eine Methode `new_double` definiert, die zu einem Keks einen neuen Keks konstruiert, dessen Schokoladen- und Keksanteile doppelt so groß sind wie die des ursprünglichen Kekses.

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def new_double(self):
        return ChocolateCookie(2 * self.chocolate,
                               2 * self.cookie)
```

Kekse verdoppeln alternativ

Was tut folgende Methode?

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def self_double(self):
        self.chocolate = 2 * self.chocolate
        self.cookie = 2 * self.cookie
```

Hier wird kein neuer Keks konstruiert, sondern ein existierender Keks verändert.

Verwendung von `self_double`

Python-Interpreter

```
>>> keks = ChocolateCookie(12, 14)
>>> keks.prettyprint()
ChocolateCookie 12 14
>>> keks.self_double()
>>> keks.prettyprint()
ChocolateCookie 24 28
```

Kein `return`

Wenn eine Methode ausschließlich das Objekt verändert oder eine Bildschirmausgabe produziert, aber darüber hinaus keinen Wert zurückgibt, braucht sie kein `return`.

Auf return achten

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie

    def weight(self):
        return self.chocolate + self.cookie

    def self_double(self):
        self.chocolate = 2 * self.chocolate
        self.cookie = 2 * self.cookie

    def new_double(self):
        return ChocolateCookie(2 * self.chocolate,
                               2 * self.cookie)

    def prettyprint(self):
        print("ChocolateCookie",
              self.chocolate,
              self.cookie)
```

Auf return achten II

account.py

```
class Account:
    def __init__(self, amount):
        self.balance = amount

    def withdraw(self, amount):
        if self.balance >= amount:
            self.balance -= amount
            return True
        else:
            return False
```

Das Gleiche oder das Selbe?

Im Deutschen können wir eine feine Unterscheidung machen: der gleiche Keks (Gleichheit) vs. der selbe Keks (Identität).

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = ChocolateCookie(12, 14)
>>> keks1.prettyprint()
ChocolateCookie 12 14
>>> keks2.prettyprint()
ChocolateCookie 12 14
```

`keks1` und `keks2` sind der gleiche Keks. Es gibt also zwei Kekse, die gleich aussehen.

Einen Keks verändern

Wir haben Obiges eingetippt und machen jetzt weiter:

Python-Interpreter

```
>>> keks1.self_double()
>>> keks1.prettyprint()
ChocolateCookie 24 28
>>> keks2.prettyprint()
ChocolateCookie 12 14
```

Wir haben `keks1` verändert, so dass er jetzt nicht mehr der gleiche Keks ist wie `keks2`.

Der selbe Keks

Nun fangen wir wieder frisch an:

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = keks1
>>> keks1.prettyprint()
ChocolateCookie 12 14
>>> keks2.prettyprint()
ChocolateCookie 12 14
```

`keks1` und `keks2` sind der selbe Keks. Es gibt also nur einen Keks!

Einen Keks verändern

Wir haben Obiges eingetippt und machen jetzt weiter:

Python-Interpreter

```
>>> keks1.self_double()
>>> keks1.prettyprint()
ChocolateCookie 24 28
>>> keks2.prettyprint()
ChocolateCookie 24 28
```

Wir haben `keks1` verändert, aber da `keks2` der selbe Keks ist, wurde er nolens volens mitverändert!

Was tut eine Zuweisung?

An dieser Stelle wollen wir eine scheinbar einfache Frage beantworten:
Was bewirkt die Anweisung `x = <ausdruck>`?

Fische und Radiosignale

Man stelle sich die Situation so vor:

- Die Daten eines Python-Programms sind Fische (Objekte), die in einem großen Meer schwimmen.
- Einige dieser Fische wurden von Meeresbiologen gekennzeichnet: Sie haben Transponder-Chips (Variablen) in der Haut, über die sie ausfindig gemacht werden könnten.
- Natürlich kann derselbe Fisch mit mehreren Chips (oder auch gar keinem) gekennzeichnet sein.

Fische und Radiosignale II

Eine Zuweisung wie `x = z + 3` entspricht der Kennzeichnung eines Fischs:

- Zunächst sucht der Meeresbiologe den Fisch mit dem Transponder `z` und holt dann einen neugeborenen Dreierfisch aus einem speziellen Zuchtbecken für Konstanten. Anschließend werden die Fische gepaart und ein Nachkomme ausgesucht.
- Danach überprüft der Meeresbiologe, ob bereits ein Fisch mit dem Transponder `x` im Meer schwimmt. Falls ja, wird er gefangen und wieder ins Meer geworfen, nachdem der Chip entfernt wurde.
- Schließlich wird dem neuen Nachkommen der Chip `x` eingepflanzt, bevor auch er ins Meer geworfen wird.

Identität: To be or not to be

- Identität lässt sich mit den Operatoren `is` und `is not` testen:
- `x is y` ist `True`, wenn `x` und `y` dasselbe Objekt bezeichnen, und ansonsten `False`.
- `x is not y` ist äquivalent zu `not (x is y)`.

Python-Interpreter

```
>>> x = ChocolateCookie(12, 14)
>>> y = ChocolateCookie(12, 14)
>>> z = y
>>> print(x is y, x is z, y is z)
False False True
>>> print(x is not y, x is not z, y is not z)
True True False
```

Gleichheit

- Wie war der Operator, um in Python Gleichheit zu testen? ==
- Aber wann sind zwei Objekte eigentlich gleich?

Gleiches Auto?

Gleiches Auto?

Gleichheit definieren

Wir sehen: wann zwei Objekte gleich sind, ist Ermessenssache!
Deshalb muss man, wenn man == für eine Klasse sinnvoll verwenden will, eine spezielle Methode definieren.

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie
    ...

    def __eq__(self, other):
        return (self.chocolate == other.chocolate and
                self.cookie == other.cookie)
```

Die Methode __eq__

Ebenso wie `__init__` ist `__eq__` kein beliebiger vom Programmierer gewählter Name, sondern er hat in Python eine spezielle Bedeutung: die Methode wird verwendet, wenn zwei Objekte dieser Klasse mittels == auf Gleichheit getestet werden.

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = ChocolateCookie(12, 14)
>>> keks1 is keks2
False
>>> keks1 == keks2
True
```

Als ich oben von “gleichen Keksen” sprach, meinte ich das in diesem Sinne.

Bedeutung der Unterscheidung

Die Unterscheidung zwischen Gleichheit und Identität ist in der imperativen Programmierung enorm wichtig und manifestiert sich in verschiedenen Programmiersprachen auf verschiedene Weise.

Vergleich

- Folgendes braucht Speicher für zwei Kekse, aber wir können sicher sein, dass eine Veränderung von `keks1` an `keks2` nichts ändert:

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = ChocolateCookie(12, 14)
```

- Folgendes braucht Speicher für nur einen Keks, aber wir müssen bedenken, dass eine Veränderung von `keks1` automatisch auch `keks2` betrifft:

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = keks1
```

Veränderliche und unveränderliche Typen

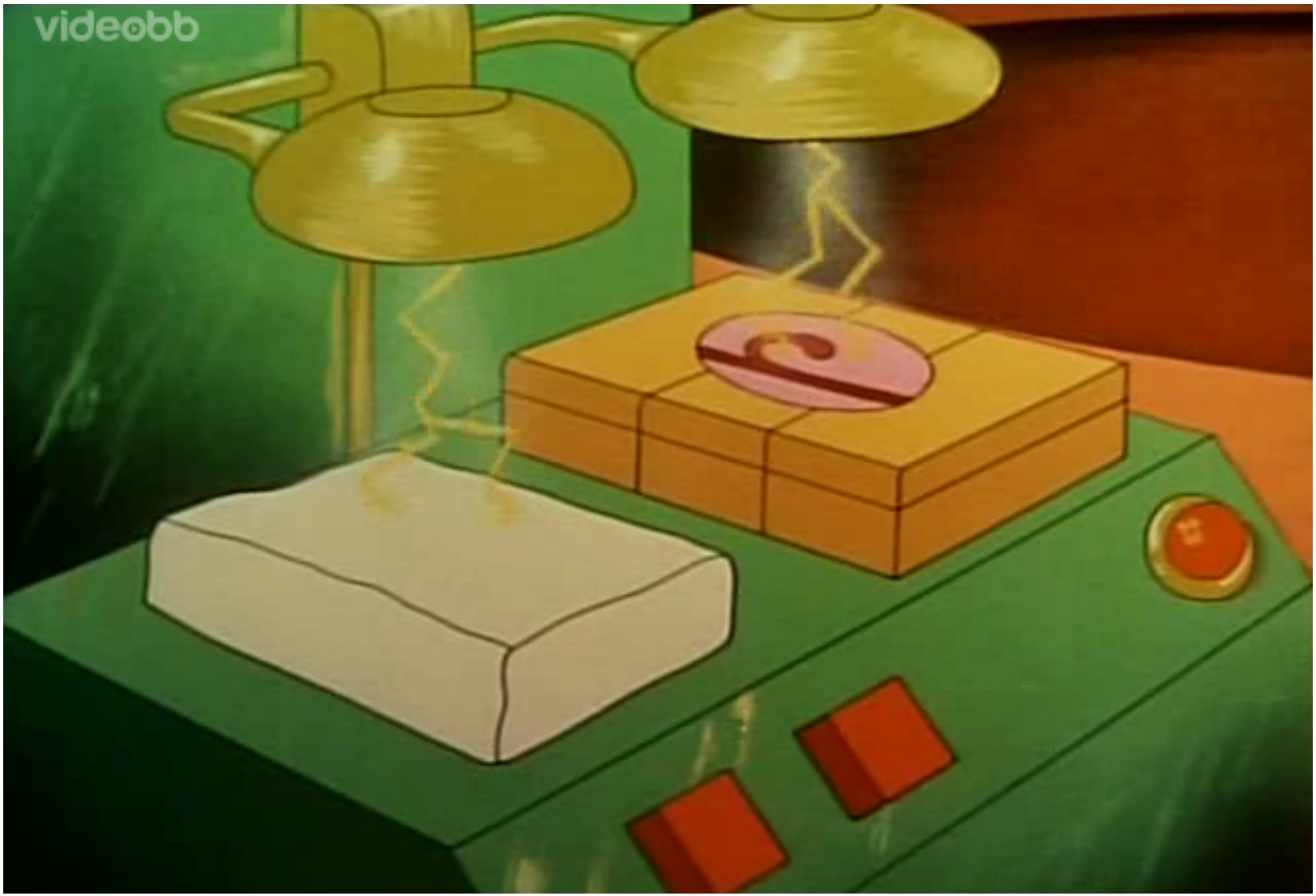
- Wie wir gesehen haben, sind Objekte der Klassen `ChocolateCookie` und `Account` veränderlich; wir sprechen von veränderlichen Typen.
  Auch Listen (`list`, kommt später).
  Hier muss man bei Zuweisungen wie `x = y` aufpassen: Operationen auf `x` beeinflussen auch `y`.
- Es gibt aber auch (eingebaute) unveränderliche Typen: `int`, `float`, `complex`, Strings (`str`), Tupel (`tuple`, kommt später).
  Diese können nicht modifiziert werden. Daher sind Zuweisungen wie `x = y` völlig unkritisch: Da man das durch `x` bezeichnete Objekt nicht verändern kann, besteht keine Gefahr für `y`.

Ein Objekt kopieren

- Ebenso wie bei der Frage, was Gleichheit bedeutet, ist es u.U. nicht ganz klar, was es bedeutet, ein Objekt zu kopieren.
- Aber natürlich kann der Programmierer einer Klasse dies selbst entscheiden, d.h., eine Methode definieren, um ein Objekt zu kopieren ...

Objekte kopieren bei Tim und Struppi

Tim und der Haifischsee



Objekte kopieren bei Tim und Struppi II

Tim und der Haifischsee

Einen Keks kopieren

cookies.py

```
class ChocolateCookie:
    def __init__(self, chocolate, cookie):
        self.chocolate = chocolate
        self.cookie = cookie

    def copy(self):
        return ChocolateCookie(self.chocolate,
                               self.cookie)
```

Ähnlich wie `new_double`, aber noch einfacher.

Verwendung der copy-Methode

Python-Interpreter

```
>>> keks1 = ChocolateCookie(12, 14)
>>> keks2 = keks1.copy()
>>> keks1.self_double()
>>> keks1.prettyprint()
ChocolateCookie 24 28
>>> keks2.prettyprint()
ChocolateCookie 12 14
```

Vergleich zur funktionalen Programmierung

- Das Konto-Beispiel stammt aus Kapitel 12 des Scheme-Buchs [KS07]: *Zuweisung und Zustand*.
- Zuweisungen und Veränderlichkeit sind dem funktionalen Paradigma eigentlich fremd und werden deshalb im Buch spät als besonderes, mit Vorsicht zu genießendes Konzept behandelt.
- Im imperativen bzw. objektorientierten Paradigma sind Zuweisungen und Veränderlichkeit hingegen eine Selbstverständlichkeit.
- Trotzdem sind nicht alle Objekte veränderlich! Unveränderliche Objekte spielen auch eine wichtige Rolle.

Zusammenfassung

- Wir wollten zusammengesetzte Daten in Python implementieren; dies führte uns zu Klassen und objektorientierter Programmierung.
- Wichtigste Begriffe: Klasse (wie ein Typ, aber noch manches mehr), Attribut, Objekt, Konstruktor, Methode.
- Dem imperativen Paradigma entsprechend sind Objekte i.A. veränderlich.
- Man muss streng unterscheiden zwischen "das gleiche Objekt" und "das selbe Objekt".

Literatur


Herbert Klaeren and Michael Sperber.
Die Macht der Abstraktion.
Teubner Verlag, 2007.