Programmierkurs Python I

Stefan Thater & Michaela Regneri
Universität des Saarlandes
FR 4.7 Allgemeine Linguistik (Computerlinguistik)

Übersicht

- Objektorientierte Programmierung (Grundlagen)
  - Klassen
  - Module
- mehr zu Sammeltypen
  - Mengen
  - Dictionaries
  - mehr Operationen

Objektorientierte Programmierung

- Prozedurale Programmierung: Daten und Operationen sind separat
- Objektorientiere Programmierung: Daten und Operationen werden in Klassen zusammengefasst
  - Daten werden in Feldern (≈ Variablen) gespeichert
  - Methoden (≈ Funktionen) definieren Operationen auf den Feldern
  - Felder und Methoden werden auch Attribute genannt
- Objekte sind Instanzen von Klassen: Klassen definieren gleichartige Objekte mit ihren Operationen

Ein erstes Beispiel: Rationale Zahlen

- Daten
  - Zähler und Nenner
- Operationen
  - in einen String konvertieren
  - Addieren
  - Multiplizieren
  - […]

Rationale Zahlen: Prozedural

```
def rat_make(num, den):
   return (num, den)

def rat_tostring(rat):
   return rat[0] + "/" + rat[1]

def rat_mul(rat1, rat2):
   num = rat1[0] * rat2[0]
   den = rat1[1] * rat2[1]
   return rat_make(num, den)
...
```


Rationale Zahlen: Objektorientiert

```
class Rat:
   def __init__(self, num, den):
      self.num = num
      self.den = den

   def toString(self):
      return str(self.num) + "/" + str(self.den)

   def mul(self, other):
      num = self.num * other.num
      den = self.den * other.den
      return Rat(num, den)
```

Rationale Zahlen: Objektorientiert

- Rationale Zahlen instantiieren (erzeugen) und an r1 bzw. r2 binden:

```
r1 = Rat(1,2)
r2 = Rat(2,3)
```

- r1 mit r2 multiplizieren; Ergebnis an r3 binden:

```
r3 = r1.mul(r2)
```

- Als String ausgeben

```
print(r3.toString())
```


Klassen

- Klassen in Python *brauchen* nichts außer einem Namen; definiert werden sie mit dem Schlüsselwort `class`

```
class <name>:
  [statement1]
  ...
  [statementn]
```

- Klassen können Methoden definieren; das sind Funktionen innerhalb der Klasse, die als erstes Argument *self* haben (*self* ist später das Objekt, das die Methode gerade aufruft)

```
class <name>:
  def fun1(self[,...]):
      ...
```

- Die Klasse hat ihren eigenen Namensraum

Klassen

- Die Klassendefinition muss im Python-Programm ausgeführt werden, bevor man die Klasse verwenden kann
- Im globalen Namensraum befindet sich dann ein Klassen-Objekt, das den Namen der Klasse hat
- Klassen (genauer: Klassen-Objekte) unterstützen genau zwei Operationen:
  - Referenzieren von Attributen
  - Instantiierung (erzeugen von Instanz-Objekten)

Klassen

```
class K:
  a = 83
  def fun(self):
    ...
```

- Instantiierung: mit `k = K()` erzeugt man ein Instanz-Objekt von K (und bindet es an k).
- Referenzieren von Attributen: Wenn K ein Klassen-Objekt ist, kann man mit `K.a` auf das Attribute a zugreifen

```
k = K()
k.fun()
```

- Zuweisungen sind erlaubt (so wie K.a = 8)

Ein einfaches Beispiel

```
class MyClass:
   i = 123
   def f(self):
      print(MyClass.i)

>>> MyClass.i
123
>>> MyClass.f
<unbound method MyClass.f>
```


Instanz-Objekte

- Instanz-Objekte können Attribute der Klasse benutzen
- Wir unterscheiden:
  - Daten-Attribute („Instanz-Variablen“)
  - Methoden
- Methoden werden nach dem referenzieren direkt aufgerufen
- Namensraum-Auflösung: wenn das Attribut nicht in der Instanz gefunden wir, wird in der Klasse gesucht

Methodenaufrufe

- Die im Klassenkörper definierten Funktionen sind die Methoden der Instanz
- Das erste Argument (`self`) der Funktion ist an die Instanz gebunden:
  - Im Beispiel ist `k.f()` äquivalent zu `MyClass.f(k)`.

```
class MyClass:
  i = 123
  def f(self):
    print(MyClass.i)
```

```
>>> k = K()
>>> k.f()
123
>>> MyClass.f(k)
123
```


`__init__`

2 underscores!

- Instantiierung erzeugt zunächst ein „leeres“ Objekt.
- Die Methode `__init__` wird automatisch mit den bei der Instantiierung verwendeten Argumenten aufgerufen.
- Typischer Code:

```
class SomeClass:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  ...
inst = SomeClass(1, 2)
```

entspricht einem sog **Konstruktor**

Zusammenfassung Klassen (1. Teil)

```
class Rat:

   def __init__(self, num, den):
      self.num = num
      self.den = den

   def mul(self, other):
      num = self.num * other.num
      den = self.den * other.den
      return Rat(num, den)

r1 = Rat(1,2)
r2 = r1.mul(Rat(2,3))
```

- Klassen werden definiert mit Schlüsselwort „class“
- Klassenmethoden mit „*self*“-Parameter
- *init*-Methode als „Konstruktor“
- Instanz-Erzeugung mit Klassen-Name ruft init auf
- Funktionsaufruf mit instanz.methode()

Module

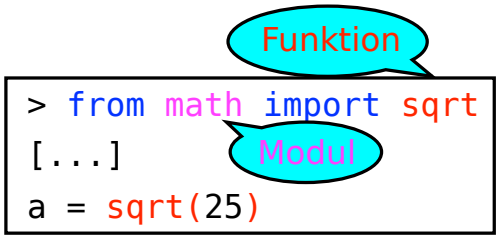
- Module sind Sammlungen von Klassen / Funktionen oder Code im allgemeinen (= *.py-Dateien)
- Module sind wiederverwertbar; man kann auf Code von anderen Modulen zugreifen
- Python hat (neben „builtins“) einige Standard-Module, auf die man bei Bedarf zurückgreifen kann (wie `sys`)
- Um die Module bzw. deren Elemente benutzen zu kennen, muss man sie importieren
  (mit `import <modulname>`)

```
> import sys
[...]
> a = sys.argv[0]
```

Modul-name

Module

- Um die Datei foo.py als Modul zu benutzen, importiert man das Modul „foo“
- Man kann auch einzelne Klassen oder Funktionen eines Moduls importieren, mit `from`



- Python findet die Module ohne zusätzliche Angaben nur, wenn
  - sie im gleichen Ordner liegen wie das aktuelle Modul
  - sie im Python-Bibliotheksverzeichnis liegen (unter UNIX z.B. oft `/usr/local/lib/python/`)

Module

- Man kann Module importieren, indem man den Pfad zu einem Unterverzeichnis explizit angibt:

```
import foo.bar.module
```

  wenn `module` im Unterordner `foo/baar` des aktuellen Verzeichnisses liegt
- mit dem Schlüsselwort `as` darf man Modulnamen an Variablen binden und später benutzen (praktisch für lange Namen)

```
import foo.bar.blah.blubb.module as fb
i = fb.methode()
```

Sammeltypen - Fortsetzung

- Listen (Klasse: `list`) und Tupel (Klasse: `tuple`) kennen wir schon
- Heute:
  - Mengen (Klassen `set` und `frozenset`)
  - Wörterbücher (Dictionaries, Maps,... Klasse: `dict`)
  - mehr Methoden aller Sammeltypen
- für Objekte `s` von irgendeinem Sammeltyp:
  - `len(s)`: Anzahl der Elemente in s
  - `s.clear()`: entfernt alle Elemente aus s
  - `s1 == s2`: (Wert-)Gleichheit von `s1` und `s2`

Listen - Methoden und Operatoren (1)

- Elemente hinzufügen:
  - Element anhängen: `li.append(elem)`
  - Element an Stelle i einfügen: `li.insert(elem,i)`
- Listen konkatenieren:
  - entweder: `newlist = list1 + list2`
  - oder: `list1.extend(list2)`
- Elemente löschen:
  - `li.remove(el)` löscht das erste `el` in der Liste `li`
  - `del li[n]` löscht das Element mit Index n
- Zugehörigkeit und Nicht-Zugehörigkeit:
  `elem in list` bzw. `elem not in list`

Listen - Methoden und Operatoren (2)

- Index des ersten Auftretens von elem in list: `list.index(elem)`
- Wie oft ist elem in list? `list.count(elem)`
- Liste invertieren: `list.reverse()`
- Liste sortieren: `list.sort()` (nur bei Typgleichheit)

```
> li = [5,2,7]
> li.reverse()
> li
[7, 2, 5]
```

```
> li = [5,2,7]
> li.sort()
> li
[2, 5, 7]
```

```
> li = [[1,2],[1,2,3], [3,2], [1,3]]
> li.sort()
> li
[[1, 2], [1, 2, 3], [1, 3], [3, 2]]
```


Listen - Methoden und Operatoren (3)

- Man darf Listen auch mit ganzen Zahlen „multiplizieren“:

```
> li = [1,2,3]
> li = li * 3
> li
[1,2,3,1,2,3,1,2,3]
```

- Bei `liste * n` erscheint `n` mal der Inhalt von `liste` in der Ergebnis-Liste; `n <= 0` ergibt die leere Liste

- Achtung: es werden keine sog. *tiefen* Kopien erzeugt! (Mehr dazu später)

```
> li = [[]] * 3
> li[0].append(1)
> li
[[1],[1],[1]]
```

Listen - *Slicing* (1)

- mit dem Slicing-Operator kann man sich Teillisten einer Liste zurückgeben lassen
  - `liste[i:]` ist die Teilliste von i bis zum Ende von `liste`
  - `liste[i:j]` ist die Teilliste von `i` bis (ausschließlich) `j`
  - `liste[i:j:k]` macht dabei immer `k`-er Schritte

```
> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
> zahlen[2:8]
[2, 3, 4, 5, 6, 7]
> zahlen[2:8:2]
[2, 4, 6]
> zahlen[8:2:-1]
[8, 7, 6, 5, 4, 3]
```


Listen - *Slicing* (2)

- mit Slicing lassen Listen sich elegant modifizieren

`del liste[0:3]` löscht die ersten 3 Elemente in `liste`

`del liste[0:5:2]` löscht jeden zweiten Eintrag ab dem 1. bis zum 5. Element in `list`

`list1[0:3] = list2` ersetzt die ersten 3 Elemente von `list1` durch die Elemente von `list2`

`list1[0:5:2] = list2` ersetzt jeden zweiten Eintrag ab dem 1. und bis zum 5. Element in `list` durch aufeinanderfolgende Einträge von `list2` (list2 muss genauso viele Elemente enthalten wie `list1[0:5:2]`!)

Listen - *Slicing* (3)

```
> zahlen = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
> zahlen[2:5] = [2,2,3,3,4,4]
> zahlen
[0, 1, 2, 2, 3, 3, 4, 4, 5, 6, 7, 8, 9, 10]
> zahlen[0:9:2] = [’a‘,’a‘,’a‘,’a‘,’a‘]
> zahlen
['a', 1, 'a', 2, 'a', 3, 'a', 4, 'a', 6, 7, 8, 9, 10]
```


Mengen: `set`

- Mengen sind ungeordnete Sammeltypen, die jedes Element höchstens ein mal enthalten

```
> numbers = [1, 2, 3, 1, 1]
> menge = set(numbers)
> menge
{1, 2, 3}
```

- als Literal: `menge ={1,2,4,5}` (leere Menge: `set()`)
- oder Definition über einen anderen Sammeltyp
- doppelte Elemente werden eliminiert
- effizient Werte auf (Listen-)Zugehörigkeit testen
- *in Mengen dürfen nur unveränderliche Typen enthalten sein!* (Zahlen, Strings, Booleans, ...)

Mengen - Methoden und Operatoren (1)

- `elem` hinzufügen: `set.add(elem)`
- `elem` entfernen:
  - `set.remove(elem)` (Fehler wenn `elem` nicht vorhanden)
  - `set.discard(elem)` (entfernt `elem` falls vorhanden)
- alle Elemente aus `set2` zu `set1` hinzufügen:
  `set1.update(set2)`
- Zugehörigkeit und Nicht-Zugehörigkeit:
  `elem in set` bzw. `elem not in set`

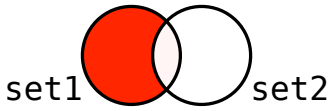
Mengen - Methoden und Operatoren (2)

Methoden können auch andere Sammeltypen als 2. Argument haben, Operatoren benötigen zwei Mengen.

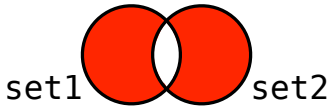
- Teilmenge / Obermenge (Rückgabe: True/False):
  - `set1.issubset(set2)` bzw. `set1.issuperset(set2)`
  - `set1 <= set2` bzw. `set1 >= set2`
- Vereinigungsmenge / Schnittmenge
  (Rückgabe: die neue Menge)
  - `set1.union(set2)` bzw. `set1.intersection(set2)`
  - `set1 | set2` bzw. `set1 & set2`

Mengen - Methoden und Operatoren (3)

- Differenzmenge (Rückgabe: neue Menge mit Elementen, die in `set1` aber nicht in `set2` sind)
  - `set1.difference(set2)`
  - `set1 - set2`



- Symmetrische Differenzmenge (Rückgabe: neue Menge mit Elementen, die entweder in `set1` oder in `set2`, aber nicht in beiden sind)
  - `set1.symmetric_difference(set2)`
  - `set1 ^ set2`



Mengen - Methoden und Operatoren (4)

- alle Mengen-Operationen gibt es auch als ‘update‘-Methode / Operator
- kein Rückgabewert, in `set1` wird das resultierende Set behalten:
  - `set1.difference_update(set2)`
    `set1 -= set2`
  - `set1.symmetric_difference_update(set2)`
    `set1 ^= set2`
  - `set1.intersection_update(set2)`
    `set1 &= set2`
  - `set1 |= set2`

Unveränderliche Mengen: *frozenset*

- es gibt eine unveränderliche Mengenvariante, das `frozenset`
- funktioniert wie set: `fs = frozenset(sammelt)`
- aber: alle Methoden, die Elemente hinzufügen, Löschen oder verändern sind verboten (`add`, `remove`, `discard`, alle `update`-Methoden)
- Alle anderen Methoden und Operatoren funktionieren wie bei `set` (und geben ggf. `frozenset` statt `set` zurück)

Initialisierung von Listen, Mengen etc.

- die Sammeltypen, die keine Wörterbücher sind (:)) kann man direkt ineinander konvertieren
- das geht mit `typname(sammel_instanz)` - siehe Mengen

```
> menge = set([1,2])
> liste = list(menge)
> tupel = tuple(menge)
> tupel2 = tuple(liste)
> menge2 = set(liste *5)
...
```

Wörterbücher: `dict` (Dictionaries, Maps)

- Wörterbücher bilden (eindeutige) Schlüssel auf Werte ab; Schlüssel müssen einen unveränderlichen Typ haben
- Zugriff auf Werte über die Schlüssel
- Beispiel: ein Telefonbuch

```
> tel = {‘Mueller‘: 7234, ‘Meier‘: 8093}
> tel[‘Meier‘]
8093
> tel[‘Schmidt‘] = 2104
> tel
{‘Mueller‘: 7234, ‘Meier‘: 8093, ‘Schmidt‘: 2104}
```


Wörterbücher als Literale

- `{}` ist ein leeres Dictionary (!), genau wie `dict()`
- einige Alternative Schreibweisen mit dem gleichen Ergebnis:

```
> tel = {‘Mueller‘: 7234, ‘Meier‘: 8093}
```

```
> tel = dict([‘Mueller‘, 7234], [‘Meier‘,8093]])
```

```
> tel = dict([(‘Mueller‘, 7234), (‘Meier‘, 8093)])
```

```
> tel = dict(Mueller=7234, Meier=8093)
```

nur mit gültigen Variablennamen als Schlüssel

Wörterbücher - Schlüssel (1)

- Schlüssel müssen unveränderliche Werte haben (siehe Mengen)
- `frozenset` ist demnach erlaubt (auch in Mengen)

```
> tel = {}
> tel[['Peter','Sophie']] = 7473
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type
```

```
> tel[frozenset(['Peter','Sophie'])] = 7473
> tel
{frozenset(['Peter','Sophie']):7473}
```


Wörterbücher - Schlüssel (2)

- Schlüsel, die beim Vergleich mit „==“ `True` ergeben, gelten als gleich
- wenn man einen Schlüssel belegt, der schon im Dictionary steht, kriegt er den neuen Wert (der alte wird gelöscht)

```
> tel[‘Peter‘] = 7473
> tel[‘Peter‘] = 9999
> tel
{‘Peter‘:9999}
```

- Anmerkung: `1` und `1.0` sind somit der gleiche Schlüssel

Wörterbücher - Methoden (1)

- Test, ob ein Schlüssel `key` in `dict` existiert:
  - `key in dict`
- Löschen eines Schlüssel-Wert-Paares (`key:value`):
  - `del dict[key]` (gibt nichts zurück)
  - `dict.pop(key)` (gibt `value` zurück)
- Setzen des Schlüssels `key` auf den Wert `value`, falls `key` noch nicht existiert:
  `dict.setdefault(key,value)`
  (wenn `key` existiert, wird der alte Wert von `key` zurückgegeben, sonst `value`)

Wörterbücher - Methoden (2)

- `dict1` mit Werten aus `dict2` ergänzen:
  `dict1.update(dict2)`
  (Doppelte Schlüssel bekommen den Wert aus `dict2`)
- „View“ aller Schlüssel: `dict.keys()`
- „View“ aller Werte: `dict.values()`
- „View“ aller Schlüssel-Wert-Paare: `dict.items()`

Vorsicht: die Reihenfolge ist hier nicht deterministisch! Einzige Garantie: zwei Aufrufe hintereinander auf dem gleichen System ohne Veränderung von `dict` liefern die gleiche Reihenfolge, korrespondierend bei Schlüssel und Werten.

Wörterbücher -“Views“

- Views sehen z.B. so aus:

```
>>> map = {'a': 1, 'l': 3, 'o': 4}
>>> map.keys()
dict_keys(['a', 'l', 'o'])
```

- sie spiegeln jeweils den aktuellen Zustand des Dictionaries
- wir betrachten sie als Sammeltypen, die wir nicht unmittelbar verändern können (nur durch Ändern des zugrunde liegenden Dictionaries)
- zur weiteren Verarbeitung kann man die Views in andere Sammeltypen konvertieren:

```
liste = list(map.keys())
```


Listen - Tupel - Mengen?

- feste Reihenfolge, alle Methoden zum Verändern von Elementen: `list`
- feste Reihenfolge, keine Methoden oder Manipulation (unveränderlich): `tuple`
- keine feste Reihenfolge, manipulierbar: `set` (viel effizienter für Zugehörigkeits-Tests als Listen)
- unveränderliche Mengen: `frozenset`
- unveränderliche Typen als Schlüssel (in `dict`) und Mengenelemente (in `set` und `frozenset`)