www.uni-frankfurt.de

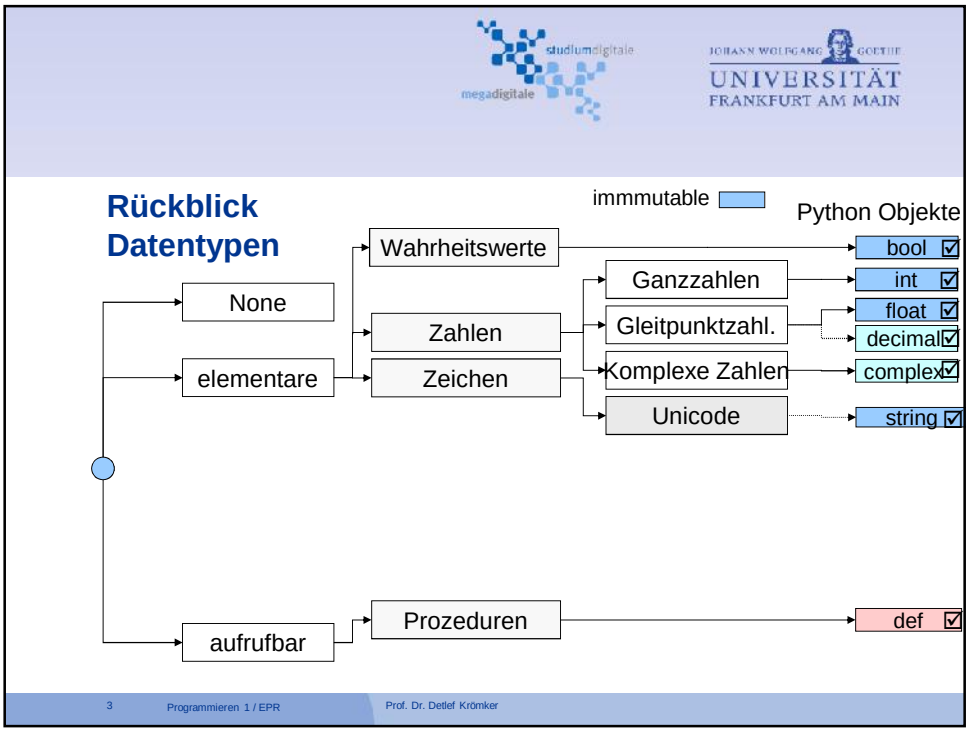
Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

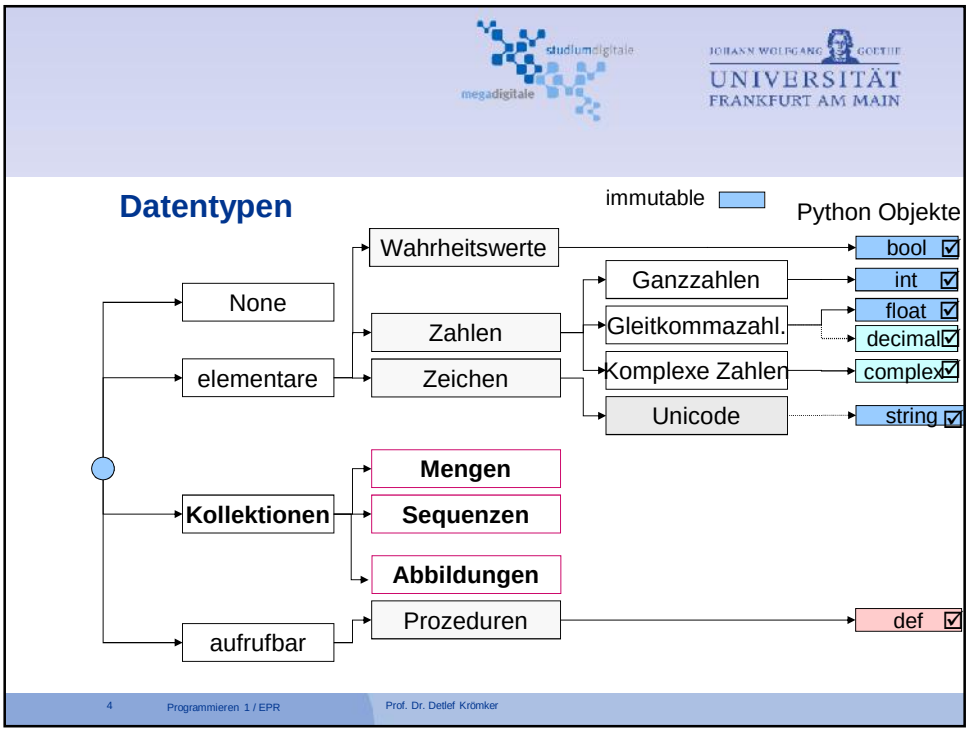
V09 Aggregierte Datentypen und Kopien

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

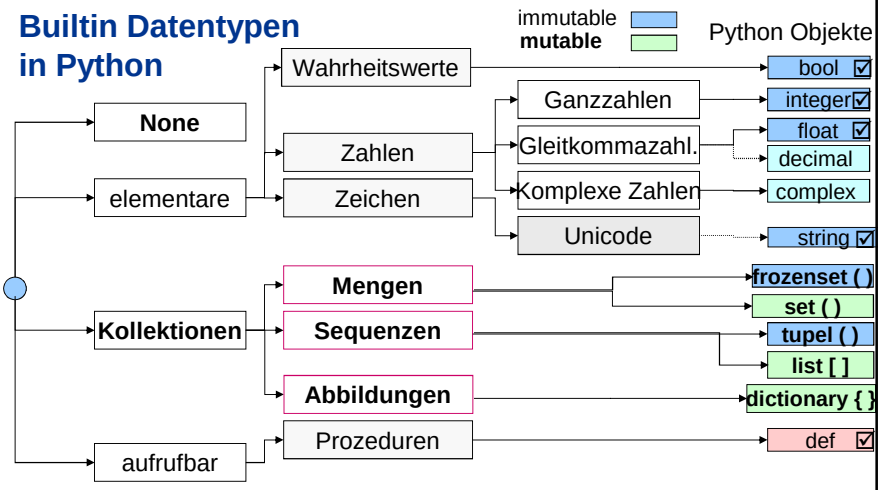
Rückblick: Datentypen – Abstrakte Datentypen

- Ein **Datentyp** (in der Informatik) ist die **Zusammenfassung von Objektmengen mit den darauf definierten Operationen**.
- Dabei werden durch den Datentyp unter Verwendung einer so genannten **Signatur** ausschließlich die **Namen dieser Objekt- und Operationsmengen** spezifiziert.
- Ein so spezifizierter Datentyp besitzt jedoch noch **keine Semantik**.
- Die Definition eines Abstrakten Datentyps (ADT) hält sich an folgendes Muster:
  - Typ/Wertebereich
  - Methoden - die **Syntax** wie mit dem Datentyp gearbeitet wird
  - Axiome - die die **Semantik** des Datentypen definieren


studiumdigitale
megadigitale
JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN
Rückblick
Datentypen
immmutable
Python Objekte
None
elementare
aufrufbar
Wahrheitswerte
Zahlen
Zeichen
Prozeduren
Ganzzahlen
Gleitpunktzahl.
Komplexe Zahlen
Unicode
bool
int
float
decimal
complex
string
def
3
Programmieren 1 / EPR
Prof. Dr. Detlef Krömker


studiumdigitale
megadigitale
JOHANN WOLFGANG GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN
Datentypen
immutable
Python Objekte
None
elementare
Kollektionen
aufrufbar
Wahrheitswerte
Zahlen
Zeichen
Mengen
Sequenzen
Abbildungen
Prozeduren
Ganzzahlen
Gleitkommazahl.
Komplexe Zahlen
Unicode
bool
int
float
decimal
complex
string
def
4
Programmieren 1 / EPR
Prof. Dr. Detlef Krömker

Builtin Datentypen in Python



Übersicht

- **Aggregierte Datentypen (Kollektionen) – Datenstrukturen**
  – (Objekte)
- **Realisierungen in Python**
  List – Tupel
  Set – Fozenset
  Dictionaries (und Hashing)
- **Kopien in Python**
  shallow copy
  deep copy
- **Zusammenfassung**

Datentypen vs. Datenstrukturen

- Zusammengesetzte (aggregierte) Datentypen sind Datenkonstrukte, welche aus einfacheren (elementaren) Datentypen oder zusammengesetzten Datentypen bestehen.
- **Datenstruktur:** Der Fokus der Diskussion liegt auf der Strukturierung/(der Kodierung und Anordnung) der gespeicherten Daten
- **Aggregierter Datentyp**: Der Fokus der Diskussion liegt auf den Operationen (im Sinne eines abstrakten Datentyps). Meist wird Wert auf **Kapselung** und **Information Hiding** gelegt. Der Datentyp ist allein durch die Schnittstelle bestimmt.

Datentypen vs. Datenstrukturen

- Die Definition von Datenstrukturen erfolgt im Allgemeinen durch die Angabe einer **Spezifikation zur Datenhaltung** (Widerspruch zur Abstraktion!)
- Die Definition erfolgt durch die Festlegung der O**perationen an der Schnittstelle**.
  Diese Spezifikation legt das allgemeine Verhalten der Operationen fest und **abstrahiert** damit von der konkreten Implementation der Datenstruktur.
- Von den meisten Datenstrukturen / Datentypen gibt es neben ihrer Grundform viele **Spezialisierungen**, die eigens für die Erfüllung ganz bestimmter Aufgaben spezifiziert wurden.

Aggregierte Datentypen

- stehen in (allen) Programmiersprachen zur Verfügung.
- Programmiersprachen können aggregierte Datentypen auf zwei Ebenen unterstützen:
  1. Sie stellt "Mechanismen" zur Verfügung, mit deren Hilfe einfache Datentypen (Integer, Float, …, andere aggregierte Datentypen) zusammengefasst werden können. Es stehen Funktionen zum Aufbau und zur Veränderung zur Verfügung.
  2. Sie verfügen über **vordefinierte aggregierte Datentypen** (builtins in Python).
- 1. ➔ bottom-up (historisch) – schauen wir uns kurz an
- 2. ➔ top-down

bottom-up - Mechanismen (1)

ARRAY (Reihung, Feld)

- alle Komponenten eines Arrays besitzen stets den **gleichen Datentyp**
- eine Komponente wird über einen Index bzw. eine Menge von Indizes angesprochen:

```
array_name [index] bzw.          # eindimensional
array_name [index_1]...[index_n] # n-dimensional
```

1, 2, 3, … dimensionale Tabelle mit gleichartigen Elementen, die unter einem Namen angesprochen werden kann.

bottom-up Mechanismen (2)

RECORD (Verbund, Struct)

- die Komponenten eines Verbundes können **unterschiedliche Datentypen** besitzen
- jede Komponente besitzt einen eigenen Namen. Angesprochen wird eine Komponente in einer Punktnotation

  record_name.komponenten_name

Zeiger oder Pointer ... eine Ergänzung

auch das gehört zur Geschichte

sind eine besondere Art von Variablen, die auf einen anderen Speicherbereich verweist (referenziert) ... eine Referenz

Der referenzierte Speicherbereich enthält entweder Daten (Objekt, Variable) oder Programmcode (ggf. sind auch mehrfache Referenzierungen zugelassen)

Array und Record sind **lineare Strukturen**: Durch Zeiger kann man dagegen beliebige verzweigte Strukturen realisieren.

Mit Zeigern kann man rechnen, sie inkrementieren, zuweisen, etc. Andererseits kann auf das verwiesene Element selbst zugegriffen werden. Dieses nennt man **De-Referenzierung**.

Zeiger = Indirektionen (Verweise)

kommen vor allem in maschinennahen Programmiersprachen wie z.B. Assembler, C oder C++ vor,

in **streng typisierten** Sprachen wie Modula-2 oder Ada **stark einschränkt.**

in Sprachen wie Java, Eiffel oder **Python** zwar intern vorhanden, aber für dem Programmierer verborgen.

(weil Programmierern bei der Arbeit mit Zeigern leicht schwerwiegende Programmierfehler unterlaufen, sicherlich die Hauptursache für „Pufferüberläufe“ und „Abstürze“ bei C oder C++ Programmen).

Sind etwas anderes als "Namens-Verweise" in Python … damit kann man **nicht** rechnen.

Zeiger – pros (1)

- Mit Feldern/Vektoren kann man durch Zeiger schnell innerhalb des Feldes springen und navigieren. Mittels Zeigerinkrement wird dabei durch ein Feld hindurchgelaufen (anstatt eines Index)
- Verweise auf Speicherbereiche können geändert werden, z.B. zur Sortierung von Listen, ohne die Elemente umkopieren zu müssen (dynamische Datenstrukturen).
- Mit Zeigern lassen sich Datenstrukturen wie „verkettete Listen“ effektiv und (fast natürlich) realisieren.

Zeiger (pros 2)

- Bei Funktionsaufrufen kann durch die Übergabe eines Zeigers auf eine Variable vermieden werden, die Variable selbst zu übergeben (vermeidet Zeit- und Platz-aufwändige Anfertigung einer Kopie)
- Anstatt Variablen jedes Mal zu kopieren und so jedes Mal erneut Speicherplatz zur Verfügung zu stellen, kann man in manchen Fällen einfach mehrere Zeiger auf ein und dieselbe Variable verweisen lassen.

Zeiger – cons (1)

- Der Umgang mit Zeigern ist relativ schwierig, kompliziert und **fehleranfällig**. (Auch bei erfahrenen Programmierern kommen Flüchtigkeitsfehler im Umgang mit Zeigern noch relativ häufig vor.)
- Programmierfehler bei der Arbeit mit Zeigern können schwerwiegende Folgen haben: z.B. Programmabstürze, unbemerkter Beschädigung von Daten oder gar Programmteilen, Pufferüberläufen, etc.
- **Die Effizienz des Prozessor-Caches leidet darunter, wenn eine Datenstruktur auf viele Speicherblöcke verweist, die im Adressraum weit auseinander liegen.**

Zeiger – cons (2)

ein „effizientes“ Programmieren mit Zeigern ist nur möglich, wenn man die Implementierung einer Datenstruktur kennt und

das steht im Widerspruch zu den Konzepten des abstrakten Datentyps oder der Objektorientierten Programmierung:

- die Komponenten eines abstrakten Datentyps sollen nach außen verborgen (gekapselt) werden
- auf die Komponenten eines abstrakten Datentyps darf nicht einzeln zugegriffen werden können, sondern nur mit Hilfe spezieller Methoden.

Aggregierte Datentypen - Zwei Sichten:

1. „Mechanismen“ zur benutzerdefinierten Definition aggregierter Datentypen
Die Bottom-up Sicht

2. Die Top-Down Sicht: Aus der Mathematik können wir hierfür, gewissermaßen **top-down**, wichtige Grundstrukturen zusammen mit typischen Grundoperationen übernehmen:

- **Mengen**
- **Tupel**
- **Relationen und Funktionen** (lesen Sie bitte im Skript den Anhang)

In Python spricht man von **Kollektionen.**

Mengen

- **Mengen:** *„Eine Menge ist eine Zusammenfassung bestimmter, wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens zu einem Ganzen. Diese Objekte heißen **Elemente** der Menge.“ (naive Definition von Cantor) … siehe auch DisMod…*
- ***Charakteristische Operationen sind:***

$$\in \quad \notin \quad \cup \quad \cap \quad \subset \quad \supset \quad \subseteq \quad \supseteq \quad \not\subset \quad \setminus \quad \Delta$$

- ***Eigenschaften: Jedes Element kommt höchstens einmal vor, ist also einzigartig!***

Tupel

- Ein **Tupel** ist eine endliche "Sammlung" von mathematischen Objekten. Dasselbe Objekt kann mehrfach vorkommen.
- Ist *n* die Länge der "Sammlung", dann spricht man von einem *n*-Tupel; 2-Tupel nennt man auch geordnete Paare (bei 2 Elementen), 3-Tupel auch Tripel, …
- Das an *i*-ter Stelle eines nichtleeren Tupels angegebene Objekt heißt seine *i*-te Komponente.
- Tupel sind durch das Gleichheitsaxiom charakterisiert:
  "*Zwei Tupel sind dann und nur dann gleich, wenn sie gleichlang sind und ihre entsprechenden Komponenten gleich sind.*"

(partielle) Funktionen / Abbildungen:

„Eine partielle Funktion ist eine rechtseindeutige Relation R.

R ist eine **Menge von *n*-Tupeln**. Objekte, die in der Relation *R* zueinander stehen, bilden ein *n*-Tupel, das Element von *R* ist.“

Sehr häufig kommen 2-Tupel (Paare) vor.

Angegeben durch 2-Tupel der Form (Schlüssel, Wert), geschrieben zum Beispiel als {Schlüssel:Wert}

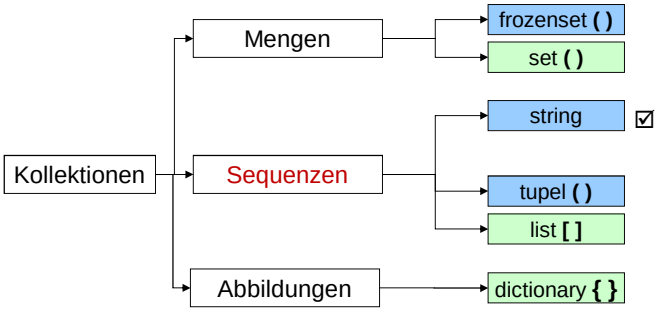
Schlüssel: aus der Menge des Definitionsbereiches (Schlüssel) und **Wert: Element** aus der Menge des Wertebereiches

Konstrukte aus der Mathematik

Struktur	Erhält eine vorhandene Ordnung	Einzigartigkeit der Werte der Elemente	Werte pro Element
Set / Frozen Set	nein	ja	1
List / Tupel	ja	nein	1
Map (Dictionary) = Funktion / Abbildung	nein	ja für das erste Element: sichert Rechtseindeutigkeit zu (also die Eigenschaft, eine partielle Funktion zu sein)	2

Python bietet **genau diese** Konstrukte an!

mutable - immutable



Bei "mutable"-Typen sind Änderungen "in place" möglich, z.B. `sort (liste)`

Sequenzen ermöglichen Index-Zugriff und Slicing

Listen (mutable)
Erzeugung von Literalen und einfache Zugriffe

- Listen werden als durch Komma getrennte Werte in **eckigen Klammern** notiert: [x , y, z]:

```
list_1 = [ ]         # eine leere Liste
list_1 = [1, 2, 3]   # eine Liste mit drei Integer-Elementen
a = liste [1]        # holt das Element 2 in der Liste

list_2 = [[1, 2, 3]] # Liste in Liste ==> wird 2-dimensional
B = list_2[0][1]     # holt das zweite Element aus der 0. Rei

list_3 = ['spam', [1, 2, 3], 3.141]  #eine verschachtelte
             # Liste: list_3 [1][0] holt das Element "1"
list_1 = list('spam') # erzeugt eine Liste durch Aufruf des
                      # Typkonstruktors list()
```

Tupel (immutable) - Erzeugung von Literalen und einfache Zugriffe … soweit, wie in einer Liste

Tupel werden als durch Komma getrennte Werte in **runden Klammern** notiert: (x , y, z).

`tupel = ( )`	ein leeres Tupel
`tupel = (1, 2, 3)`	ein Tupel mit drei Integer-Elementen
`tupel [1]`	Achtung: **eckige Klammern**, holt das Element 2, weil dies eine **Indexierung** ist.
`tupel = ('spam', (1, 2,3), 3.141)`	eine verschachteltes Tupel: tupel [1][0] holt das Element 1
`tupel = tupel('spam')`	erzeugt ein Tupel durch Aufruf des Typkonstruktors
`tupel = 1, 2, 3`	das gleiche Tupel wie oben, ist aber **in Funktionsaufrufen nicht** erlaubt, weil nicht zwischen Parametern und Tupel-Elementen unterschieden werden kann. Besser nicht verwenden!
`(1,)`	**ein Tupel mit einem Element** – kein geklammerter Ausdruck, wird durch das Komma deutlich gemacht

… aber es gibt auch Unterschiede!

Die folgenden Operationen gibt es nur bei Listen, weil sie Veränderungen "in place" vornehmen.

`L[i] = x`	Ersetzt das i-te Element in der Liste L durch x
`del L[i]`	Lösche das i-te Element in L
`L.append(x)`	Das Element x hinten an L anfügen
`L.clear()`	Leert die Liste L
`L.extend(x)`	Erweitert L um die Liste x
`L.insert(i, x)`	Fügt x am Index i in Liste L ein (und verschiebt den Rest von L nach hinten)
`L.remove(x)`	Entfernt alle x aus Liste L
`L.reverse()`	Dreht die Liste L um
`L.sort()`	Sortiert die Elemente der Liste

Beispiele

```
>>> L = [1, 2, 3]
>>> id(L)
55995648
>>> L[2] = 4
>>> L.append(3)
>>> L
[1, 2, 4, 3]
>>> id(L)
55995648
```

```
>>> L.reverse()
>>> L
[3, 4, 2, 1]
>>> id(L)
55995648
>>> L.sort()
>>> L
[1, 2, 3, 4]
>>> id(L)
55995648
>>>
```


Sequenz Operationen, hier help (list) … „help( ) ist immer eine gute Idee“

```
>>> help (list)
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __delitem__(...)
 |      x.__delitem__(y) <==> del x[y] |
```

```
>>> x = [1, 2, 3]
>>> y = [5, 4, 3]
>>> i = 1
>>> x + y
[1, 2, 3, 5, 4, 3]
>>> 3 in x
True
>>> del x[i]
>>> x
[1, 3]
>>>
```

help(list)

```
__eq__(...)
    x.__eq__(y) <==> x==y

__ge__(...)
    x.__ge__(y) <==> x>=y

__getattribute__(...)
    x.__getattribute__('name') <==> x.name

__getitem__(...)
    x.__getitem__(y) <==> x[y]

__gt__(...)
    x.__gt__(y) <==> x>y

__iadd__(...)
    x.__iadd__(y) <==> x+=y

__imul__(...)
    x.__imul__(y) <==> x*=y
```


help(list)

```
__init__(...)
    x.__init__(...) initializes x; see help(type(x)) for signature

__iter__(...)
    x.__iter__() <==> iter(x)

__le__(...)
    x.__le__(y) <==> x<=y

__len__(...)
    x.__len__() <==> len(x)

__lt__(...)
    x.__lt__(y) <==> x<y

__mul__(...)
    x.__mul__(n) <==> x*n
```

```
>>> x = [1, 2, 3]
>>> y = [5, 4, 3]
>>> i = 2
>>> x * y
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in <module>
    x * y
TypeError: can't multiply sequence by non-int of type 'list'
>>> x * i
[1, 2, 3, 1, 2, 3]
>>>
```

help(list)

```
|  __ne__(...)
|      x.__ne__(y) <==> x!=y
|  __repr__(...)
|      x.__repr__() <==> repr(x)
|
|  __reversed__(...)
|      L.__reversed__() -- return a reverse iterator over the list
|
|  __rmul__(...)
|      x.__rmul__(n) <==> n*x
|
|  __setitem__(...)
|      x.__setitem__(i, y) <==> x[i]=y
|
|  __sizeof__(...)
|      L.__sizeof__() -- size of L in memory, in bytes
```

```
>>> x = [1, 2, 3]
>>> x.__sizeof__()
32
>>>
```


help(list)

```
|  append(...)
|      L.append(object) -- append object to end
|
|  count(...)
|      L.count(value) -> integer -- return number of occurrences of value
|
|  extend(...)
|      L.extend(iterable) -- extend list by appending elements from the
   iterable
|
|  index(...)
|      L.index(value, [start, [stop]]) -> integer -- return first index of
   value.
|      Raises ValueError if the value is not present.
|
|  insert(...)
|      L.insert(index, object) -- insert object before index
|
```

help(list)

```
|  pop(...)
|      L.pop([index]) -> item -- remove and return item at index (default
  last).
|      Raises IndexError if list is empty or index is out of range.
|
|  remove(...)
|      L.remove(value) -- remove first occurrence of value.
|      Raises ValueError if the value is not present.
|
|  reverse(...)
|      L.reverse() -- reverse *IN PLACE*
|
|  sort(...)
|      L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```

```
>>> y = [5, 4, 3, 2]
>>> pop (y)
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    pop (y)
NameError: name 'pop' is not defined
>>> y.pop ()
2
```


help(list)

```
|  pop(...)
|      L.pop([index]) -> item -- remove and
  last).
|      Raises IndexError if list is empty or index is out of range.
|
|  remove(...)
|      L.remove(value) -- remove first occurrence of value.
|      Raises ValueError if the value is not present.
|
|  reverse(...)
|      L.reverse() -- reverse *IN PLACE*
|
|  sort(...)
|      L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```

```
>>> y = [5, 4, 2, 3]
>>> id (y)
42228472
>>> y.reverse ()
>>> y
[3, 2, 4, 5]
>>> id (y)
42228472
>>>
```

help (list)

```
>>> y = [5, 4, 2, 3]
>>> id (y)
42969688
>>> y
[5, 4, 2, 3]
>>> y.sort ()
>>> y
[2, 3, 4, 5]
>>> id (y)
42969688
>>>
```

```
|  pop(...)
|      L.pop([index]) -> item -- remove and
  last).
|      Raises IndexError if list is empty or
|
|  remove(...)
|      L.remove(value) -- remove first occurrence of value.
|      Raises ValueError if the value is not present.
|
|  reverse(...)
|      L.reverse() -- reverse *IN PLACE*
|
|  sort(...)
|      L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```


JA: So kann man sich beispielhaft ein neues Konstrukt erobern

- ALLE Operationen / Methoden ausprobieren
- Funktionieren diese wie erwartet (meistens JA)
- **Machen Sie dies auch für tupel (was ist anders als bei list) und set / frozenset und dictionary**
- `help` benutzen!

Mengen
Set – Frozenset

Mengenobjekte werden mit dem eingebauten Konstruktoren
set () (➔ mutable) oder
frozenset () (➔ uimmutable)
erzeugt.
In den Klammern () darf ein beliebiger **iterierbarer Typ** stehen, z.B. String, Liste, Tupel

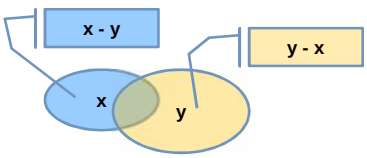
Nichtleere Mengenobjekte können durch **{**e1,e2,…**}** erzeugt werden.
Achtung:
Eine leere Menge wird NICHT durch {} erzeugt (!) ➔ das wäre ein leeres Dictionary!

```
>>> x = {'a', 'c', 'b'}
>>> y = set('bcde')
>>> x
{'a', 'c', 'b'}
>>> y
{'c', 'b', 'e', 'd'}
```


Mengenoperationen

```
>>> x = {'a', 'c', 'b'}
>>> y = set('bcde')
>>> 'b' in x
True
>>> x - y  # Differenz
{'a'}
>>> y - x  # Differenz
{'e', 'd'}
>>> x ^ y  # Symmetrische Differenz
{'a', 'e', 'd'}
```

Symmetrische Differenz



x ^ y = (x - y) | (y - x)

Mengen -- Operatoren, Funktionen und Methoden

`len(N)`	Kardinalität von N	Integer
`x in M`	(True/False)	Ist x Element von M?
`x not in M`		ist x nicht Element von M?
`N <= M  N < M`	M.issubset(N)	N istTeilmenge von M (True/False)
`N >= M  M > N`	M.isssuperset(N)	N ist Obermenge von M (True/False)
`M | N`	M.union(N)	Vereinigung von M und N
`M |= N`	M.update(N)	
`M & N`	M.intersection(N)	Schnittmenge von M und N
`M &= N`	M.intersection_update(N)	
`M - N`	M.difference(N)	Differenz von M und N
`M -= N`	M.difference_update(N)	
`M ^ N`		Symmetrische Differenz

Methoden (beachte die Punktnotation)

M.add(x)	Füge ein Element x zur Menge M hinzu. (Hat keine Wirkung, wenn x schon Element von M ist)
M.clear()	Erzeugt die Leere Menge M
M.pop()	Entfernt ein (beliebiges) Element von der Menge M
M. remove(x)	Entferne Element x von der Menge M (x muss ein Element von M sein, sonst Key error)

… siehe Python Docu

Nutzen von Mengen

Hauptnutzen ist:

- Testen auf Mitgliedschaft
- Duplikate entfernen
- Mengenoperationen wie Durchschnitt, Differenz berechnen!

Elemente müssen **hashable** sein, d.h. insbesondere inmutable und vergleichbar sein.

Alle primitiven Python-Typen sind inmutable, wie Integer, Float, String, …
Nicht: Liste, Menge
Vorgriff: Objekte, die Instanzen von user-defined Klassen sind, sind defaultmäßig hashable; sie sind ungleich zu allen anderen Objekten (und Instanzen) und ihr hash value ist ihre id.

Dictionaries (1)

Dictionaries implementieren partielle Funktionen

Hierzu benutzt man 2-Tupel (Paare) der Form (Schlüssel, Wert), geschrieben zum Beispiel als `{Schlüssel:Wert.}` Da der Wert wieder ein n-Tupel sein kann, aber auch eine Liste, etc. sind beliebige partielle Funktionen implementierbar.

Dictionaries sind in Python ein eingebauter Datentyp. Im Gegensatz zu Sequenzen, die mit einem Zahlen-Intervall indiziert werden, werden Dictionaries **über Schlüssel indiziert**, die **irgendeinen 'immutable' Typ** haben dürfen.

Strings und Zahlen können immer Schlüssel sein. Tupel können als Schlüssel verwendet werden, wenn sie nur Strings, Zahlen, Tupel, Frozensets enthalten.

Listen, Sets, andere Dictionaries können **nicht** als Schlüssel verwendet werden. **Warum???**

Dictionaries (2)

Ein Paar geschweiften Klammern erzeugt ein leeres Dictionary: `{}`.

Eine durch Kommata getrennte Liste von (`Schlüssel:Wert`)-Paaren innerhalb der Klammern fügt die initialen (Schlüssel:Wert)-Paare in das Dictionary ein. So werden Dictionaries auch ausgegeben.

Hauptoperationen auf einem Dictionary sind:

- das Speichern eines Wertes unter einem Schlüssel und das Abrufen dieses Wertes bei Angabe des Schlüssels.
- Es ist auch möglich, ein (`Schlüssel:Wert`)-Paar mit del zu löschen.
- Wird beim Speichern ein Schlüssel verwendet, der bereits existiert, so wird der alte damit assoziierte Schlüssel vergessen.
- Es erzeugt einen Fehler, einen Wert mit einem nicht existierenden Schlüssel abzurufen.

Nutzung von Dictionaries (dict)

- Immer dann, wenn ich nicht über einen Index ➔ Liste, sondern über einen Schlüssel (beliebiger immutable Datentyp: Zahlen, Strings, Tupel, …) auf eine Sammlung zugreifen will.
- Das häufigste Beispiel für die Nutzung eines Dictionaries ist das eines Wörterbuchs z.B. {deutsch:englisch}, eines Telefonbuchs {Name: Telefonnummer}, etc. {Schlüssel:Wert}- oder {key:value}-Paar
- Dictionaries kommen sehr häufig vor und sollten effizient implementiert sein, insbesondere:
  1. Einfügen eines neuen Wertes v mit einem Schlüssel k
  2. Finden eines Wertes v mithilfe seines Schlüssels k
  3. Das Löschen eines Schlüssels k (Zusammen miit dem Wert v)

Die wichtigsten Operationen auf Dictionaries

```
D.clear() -> None.  Remove all items from D.
D.copy() -> a shallow copy of D
fromkeys(iterable, value=None, /) from builtins.type
    Returns a new dict with keys from iterable and values equal to value.
D.get(k[,d]) -> D[k] if k in D, else d.  d defaults to None.
D.items() -> a set-like object providing a view on D's items
D.keys() -> a set-like object providing a view on D's keys
D.pop(k[,d]) -> v, remove specified key and return the corresponding
    value. If key is not found, d is returned if given, otherwise KeyError is raised.
D.popitem() -> (k, v), remove and return some (key, value) pair as a
    2-tuple; but raise KeyError if D is empty.
D.setdefault(k[,d]) -> D.get(k,d), also set D[k]=d if k not in D
D.update([E, ]**F) -> None.  Update D from dict/iterable E and F.
    If E is present and has a .keys() method, then does:  for k in E: D[k] = E[k]
    If E is present and lacks a .keys() method, then does:  for k, v in E: D[k] = v
    In either case, this is followed by: for k in F:  D[k] = F[k]
D.values() -> an object providing a view on D's values
```


Iterationen mit Dictionaries

- Es ist keine Reihenfolge der Items in Dictionaries definiert
- trotzdem kann über Elemente iteriert werden:
  - iteritems(): Iteration über Tupel (key, value)
  - iterkeys(): Iteration über Schlüssel
  - itervalues(): Iteration über Werte

Wie sind dict implementiert? – Antwort: Hashing Einfaches Beispiel für eine Hashtabelle

Index: e = h (key)	Eintrag = value
0	
1	11 oder 21 oder 'otto'
2	
3	
4	14 oder 24 oder 'marta'
5	
…	
n-1	

Eine sehr einfache Hashfunktion für Integer,
Hashfunktion: h(i) = i % 10 (= i modulo (n))

Prinzip von Hashverfahren

- Datensätze werden in einem Feld/Array mit Indexwerten 0 .. n-1 gespeichert
- Positionen werden auch als "Buckets" bezeichnet
- Eine Hashfunktion h bestimmt für den key den Index h(key)

Wie ist das implementiert? – Antwort: Hashing

… davon werden Sie in Algorithmen und Datenstrukturen noch sehr viel hören

Index = h (key)	Eintrag
0	
1	'otto'
2	
3	
4	'marta'
5	
…	
n-1	

Eine sehr einfache Hashfunktion für Integer,
Hashfunktion: h(i) = i % n (= i modulo (n))

Prinzip von Hashverfahren

- Die Hashfunktion h sollte die Elemente möglichst "gut", d.h. mit möglichst wenigen Kollisionen auf die Buckets verteilen
- **Kollision** liegt vor, wenn Hashfunktion mehrere Element auf denselben Bucket abbildet
- Varianten des Hashverfahrens unterscheiden sich bei der Behandlung von Kollisionen

Da müssen Sie sich mit vertraut machen. Sorry.

Bitte probieren Sie die genannten Funktionen vielleicht am Beispiel

```
d_de = {'eins': 'one', \
        'zwei': 'two', \
        'drei': 'three', \
        'vier': 'four'}
```

aus.

Schauen Sie sich auch die verschiedenen Möglichkeiten zur Erzeugung eines dict an.

Schauen Sie auch ins Python Tutorium.

Übersicht

- **Aggregierte Datentypen (Kollektionen) – Datenstrukturen – (Objekte)**
- **Realisierungen in Python**
  List – Tupel
  Set – Fozenset
  Dictionaries (und Hashing)
- **Kopien in Python**
  shallow copy
  deep copy

Tipp:
Noch einmal kurz wirklich aufpassen!

Zuweisung -- *(shallow)*copy -- deepcopy

- Wie Sie bereits wissen, wird in Python bei einer Zuweisung nur eine neue Referenz auf ein und dieselbe Instanz erzeugt, anstatt eine Kopie der Instanz zu erzeugen. Im folgenden Beispiel verweisen s und t auf dieselbe Liste, wie der Vergleich mit is offenbart:

```
>>> s = [1, 2, 3]
>>> t = s
>>> t is s
True
```

- Dieses Vorgehen ist **nicht immer** erwünscht, weil Änderungen (bei mutable Datentypen) an der von s referenzierten Liste auch auf t wirken: Sogenannte **„Seiteneffekte“**

Kopien durch das Modul copy
*(shallow)*copy -- deepcopy

- Alternativ: Das Modul **copy** benutzen, das dazu gedacht ist, echte Kopien einer Instanz zu erzeugen.
- Für diesen Zweck bietet copy zwei Funktionen an:
  - **copy.copy() und**
  - **copy.deepcopy().**
- Beide Methoden erwarten als Parameter die zu kopierende Instanz und geben eine Referenz auf eine Kopie von ihr zurück:
- [Natürlich kann eine Liste auch per Slicing echt kopiert werden. Das Modul copy erlaubt aber das Kopieren beliebiger Instanzen.]

*(shallow)*copy

```
>>> import copy
>>> s = [1, 2, 3]
>>> t = copy.copy(s)
>>> t
[1, 2, 3]
>>> t is s
False
```

Das Beispiel zeigt, dass t zwar die gleichen Elemente wie s enthält, aber trotzdem nicht auf dieselbe Instanz wie s referenziert, sodass der Vergleich mit is negativ ausfällt.

*(shallow)*copy -- deepcopy

- Der Unterschied zwischen copy und deepcopy besteht darin, wie mit Referenzen umgegangen wird, die die zu kopierenden Instanzen enthalten.
- Die Funktion `copy.copy` erzeugt bei Listen zwar eine neue Liste, aber die Referenzen innerhalb der Liste verweisen trotzdem auf dieselben Elemente. Mit `copy.deepcopy` hingegen wird die Instanz selbst kopiert und anschließend rekursiv auch alle von ihr referenzierten Instanzen.
- (Es gibt allerdings Datentypen, die sowohl von copy.copy als auch von copy.deepcopy nicht wirklich kopiert, sondern nur ein weiteres Mal referenziert werden. Dazu zählen unter anderem Modul-Objekte, Methoden, file-Objekte, socket-Instanzen und traceback-Instanzen.)
- Machen Sie sich hiermit vertraut!

Tipp

- Beim Kopieren einer Instanz mithilfe des copy-Moduls wird das Objekt ein weiteres Mal im Speicher erzeugt.
- Dies kostet ggf. erheblich mehr Speicherplatz und Rechenzeit als eine einfache Zuweisung.
- Deshalb sollten Sie copy wirklich **nur dann** benutzen, wenn Sie tatsächlich eine echte Kopie brauchen.

Zusammenfassung

Praktisch wichtige Datenstrukturen kennen gelernt

Felder (array)	**Listen (list)**
Mengen (set)	Stapel (stack)
(Warte-)Schlange (queue)	Verbund (struct)
Dictionaries	Weitere

Warum hat man eigentlich in Python keine arrays, kein struct, …?

Zuweisung -- *(shallow)*copy -- deepcopy

Realisierungen in Python
Nutzen Sie die help-Funktion!

Es gibt wieder zwei Quiz! -- Unbedingt machen!

Fragen und (hoffentlich) Antworten

Ausblick

Jetzt kommt eine Woche PS 2 Stoff

Montag: Testen
Freitag: noch mal Testen + (Iteration vs. Rekursion)

Und … Danke für Ihre Aufmerksamkeit