

Skriptsprachen: Python

Datenstrukturen



Jan Krüger, Alexander Sczyrba

Technische Fakultät
Universität Bielefeld

16. Februar 2020

Immutable vs. Mutable

Bisher bekannte Datentypen: `int` und `string`.

- Beide sind immutable → Aber was bedeutet das?
 - Werte sind fix und können nicht verändert werden
 - Neuweisung durch: `zahl = zahl + 1`
 - Bei Neuweisung wird neues Objekt mit gewünschtem Wert erstellt
- Welchen Nutzen haben also Immutablees?
 - Mutable (veränderbare Datentypen) verwenden Immutablees zur Adressierung
 - Werden auch als *hashable* bezeichnet
 - Gleiche Immutablees referenzieren (zum Teil) zur Speicheroptimierung auf dieselbe Speicheradresse

Datentypen in Python

Es existieren unterschiedliche „*built-in*“ Datentypen (*immutable*, *mutable*):

Wahrheitswerte `bool`

Zahlen `int`, (`long`), `float`, `complex`

Sequentielle Daten `string`, `tuple`, `list`

Mapping `dict`

Sets `set`, `frozenset`

Spezialfall: `None`.

- „Nichts“ wert
- Entspricht in:

Java `null`

Perl `undef`

Python

—

Wahrheit

Boolean, Wahrheitstests, Vergleichoperatoren

Was ist Wahrheit in Python

- Wahrheit ist bei `bool` klar definiert
 - `True`
 - `False`
- Zusätzlich ergibt jeder Wahrheitstest auf einem Objekt `True`

Was ist Wahrheit in Python

Als False wird folgendes angesehen:

None

False

0, 0L, 0.0, 0j Die Zahl 0

, ", (), [] Leerer sequentieller Datentyp

{ } Leeres Mapping

Sonderfall bei eigenen Klassen, die eine der folgenden Methoden implementieren und diese als Rückgabewert False oder 0 liefern würden:

- `__nonzero__()`

- `__len__()`

Boolsche Operatoren

Die folgenden Operatoren ersetzen die in anderen üblichen Kürzel `||`, `&&` respektive `!`, sortiert nach abnehmender Priorität.

`X or Y` Logisches *ODER*, als Funktion: Wenn `X == False`, dann `X`, sonst `Y`

`X and Y` Logisches *UND*, als Funktion: Wenn `X == True`, dann `X`, sonst `Y`

`not X` Logische Negierung, (kein `!`; Würde die Anweisung an die Kommandozeile weiterleiten)

Darüber hinaus sind sie spezielle Funktionen, die Eingabewerte als Rückgabe besitzen.

Vergleichsoperatoren

- Sind höher priorisiert als Boolsche Operatoren
- Können verkettet werden:
 $X < Y \leq Z$ wird zu $X < Y$ and $Y \leq Z$, wobei der letzte Ausdruck nur dann ausgewertet wird, wenn der erste True ergeben hat
- Es können alle Objekte miteinander verglichen werden
- Objekte unterschiedlichen Typs können niemals gleich sein, werden aber immer gleich sortiert → Immer Identische Sortierung heterogener Arrays gegeben
- Objekte desselben Typs müssen `__cmp__()` implementieren, um als gleich evaluiert werden zu können

Vergleichsoperatoren

<code><</code> kleiner als	<code><=</code> kleiner gleich
<code>></code> größer als	<code>>=</code> größer gleich
<code>==</code> gleich	<code>!=</code> ungleich
<code>is</code> Objektidentität	<code>is not</code> ungleiche Objektidentität

- `<`, `<=`, `>` und `>=` werfen bei Vergleichen mit komplexen Zahlen einen `TypeError`
- Bei sequentiellen Datentypen existieren noch zwei weitere Vergleichsoperatoren

`in` ist enthalten

`not in` ist nicht enthalten

Python

—

Zahlen

Integer, Float, Complex

Zahlen

Es existieren in Python 2.X vier unterschiedliche Typen von Zahlen:

Integer/Long Ganzzahlen

```
100          # Integer
100L         # Long
sys.maxint   # Maximale Groesse eins Ints,
              # bis es automatisch zu
              # Long wird
```

Float Fließkommazahlen, 1. oder -55.3

Complex Komplexe Zahlen, 1.5+2j

Zahlen – Besonderheiten & Operanden

- Ab Python 3 bzw. 2.6 wurden Integer und Long zusammengeführt
- Wenn die Range einer Integerzahl zu groß wird, wird sie automatisch zu einem Long
- `int <OPERAND>` `int = int` und `float <OPERAND>` `int = float`

`+, -, *, / ...` II

`//` ganzzahl Anteil der
Division

`%` Modulo

`-X` Negation von X

`+X` X unverändert

`Y**Z` Y^Z

Shorthandzuweisung: `x <OPERAND>= y` $\rightarrow$ `x = x <OPERAND> y`

Zahlen – Wichtige Funktionen

`abs()` Betragsfunktion

`long()` Ausgabe als long

`complex(r, i)` Komplexe Zahl,
(r)ealteil, (i)maginärteil

`divmod(x,y)` (x // y, x % y)

`int()` Ausgabe als int

`float()` Ausgabe als float

`c.conjugate()` Konjugation einer
komplexen Zahl

`math.*` Siehe `math`-Modul

Übung – Präzedenz der Operatoren

Berechne die folgenden Ausdrücke. Wie ist die Präzedenz der Operatoren?

> 4 * 3 + 6

> 4 + 3 * 6

> 100 / 5 % 3

> 5 * 6 ** 7

> res = 7.

> n = 5

> res //= -n

Python

—

Sequentielle Datentypen

String, Tuple, List

Sequentielle Datentypen

Sequentielle Datentypen können allgemein als Container für Abfolgen von Objekten bzw. ihrer Referenzen gesehen werden. Für Strings, Listen und Tupel gelten dieselben Standardzugriffsoperatoren.

```
> s = "Hallo_Du!"  
      012345678
```

```
> s[2:7]  
    'llo_D'
```

```
> s[0]  
    'H'
```

```
> list = [s, 54, 'Welt']  
          0  1  2
```

```
> list[1]  
    54
```

```
> list[2][0:2]  
    'We'
```


Sequentielle Datentypen – Operanden & Funktionen

`x in s` True, wenn `x` in `s`
vorkommt, sonst
False

`s*n, n*s` Konkatenation `n`
Kopien von `s`

`s[i]` Objekt an `i`'ter Stelle

`s[i:j]` Slice von `i` bis `j`

`len(s)` Länge von `s`

`s.index(i)` Erstes Auftreten von `i`
in `s`

`x not in s` False, wenn `x` in `s`
vorkommt, sonst
True

`s+t` Konkatenatio von `s`
und `t`

`s[i:j:k]` Slice von `i` bis `j` mit
Schrittweite `k`

`min(s)` Kleines Element in `s`

`max(s)` Größtes Element in `s`

`s.count(i)` Anzahl aller `i` in `s`

String

- Zeichenkette beliebiger Länge
- Zeichen stehen bei Instanziierung zwischen `"_"` oder `'_'`
- Zeichenindizes beginnen bei 0, letzter Index: `len(string)-1`
- *Immutable* → Veränderungen der Form `s[0] = "a"` nicht zulässig
- Escapezeichen: `„\“`
- Nur ASCII-Zeichen werden in 2.x problemlos verarbeitet
 - `String != Unicode`
 - Erst in Python 3.x wurden beide Typen zusammengeführt

```
s = "Ich_bin_ein_String_der_Laenge_32"
```

String – Wichtigste Methoden

`str(<INPUT>)` INPUT als String

`count(sub[, start[, end]])` Anzahl nicht überlappender Substrings sub

`endswith(suffix[, start[, end]])` True, wenn der String mit suffix endet, sonst False; Kann auch ein tuple mit Strings sein

`find(sub[, start[, end]])` Indexposition des ersten Auftretens von sub, sonst -1

`r/lstrip([chars])` Entfernen der Zeichen chars von links bzw. rechts, parameterlos oder bei None der Leerzeichen. Wenn beidseitig gestrippt werden soll, verwendet `strip([chars])`

- Angabe von optionalem start und end in *slice*-Notation

String – Wichtigste Methoden

`partition(sep)` String wird an `sep` getrennt; Rückgabe als 3-Tupel:
(`prefix`, `sep`, `suffix`), wenn `sep` nicht gefunden wurde
(`string`, '', '')

`split([sep[, maxsplit]])` Teilt den String nach jedem `sep`, wenn gegeben, sonst nach jedem Whitespace. `maxsplit` gibt die maximale Anzahl an Teilungen an

`capitalize()` Erster Buchstabe groß, Rest in Kleinbuchstaben

`swapcase()` Kleinbuchstaben werden zu Großbuchstaben und umgekehrt

`join(<ITERABLE>)` Fügt alle Elemente aus `ITERABLE` an den String an

Übung – String und Zahlen

Eine kleine Fingerübung zum Thema Typen, Konvertierung und Modifizierung:

1. Erstellt euch die Zahl `z` mit dem Wert `4353,3`
2. Konvertiert diese als `string` und splittet diese bei dem Punkt und speichert die Rückgabe in einer eigenen Variable `l`.
3. Addiert `1` auf den Vorkommateil und speichert ihn wieder als `string` in `l`
4. Überschreibt `z`, wobei hier nun der Nachkommateil vor dem Komma stehen soll und der modifizierte Vorkommateil aus `l` hinter das Komma kommen soll.

`z` und `l` sowie ihre Inhalte sollen immer vom selben Typ sein!

Tuple

- Nicht veränderbare Liste → *Immutable*
- Beliebige Länge
- Elemente können unterschiedlichen Typs sein
- Notation:
 - Inhalt steht zwischen runde Klammern (...)
 - Objekte werden kommasepariert aufgelistet

```
> s1 = (u"are" ,1)
> suffix = "tuple"
> _tuple = (s1, 1337, suffix)
> _tuple
((u"are", 1), 1337, "tuple")
```

List

- Veränderbare Liste → *Mutable*
- Beliebige Länge
- Elemente können unterschiedlichen Typs sein
- Notation:
 - Inhalt steht zwischen eckigen Klammern [...]
 - Objekte werden kommasepariert aufgelistet

```
> s1 = (u"are" ,1)
> suffix = "tuple"
> _list = [s1, 1337, suffix]
> _list[2] = "list"
> _list
[(u"are", 1), 1337, "list"]
```

Übung – (Im)mutable

1. Erstellt euch eine Variable `tuple` wie auf der Tuple-Folie zu sehen.
2. Instanziiert euch eine Variable vom Typ `list` mit den Inhalten aus `_tuple`. Verändert diese Liste, sodass diese mit `_list` inhaltsgleich ist.
3. Rückkonvertiert die erstellte Liste wieder zu einem Tuple und legt den Inhalt wiederum in eine neue Variable vom `tuple` ab.

Verwendet die Funktionen `list()` sowie `tuple()`.

Operationen auf sequentiellen *mutable* Datentypen

Referenzen ersetzen und entfernen:

$s[i] = x$ Neuzuweisung

$s[i:j] = t$ Sliceersetzung mit *Iterable* t

$s[i:j:k] = t$ Intervalersetzung, Schrittweite k , mit Elementen aus t

$s.insert(i, x)$ x an Index i einschieben $\rightarrow s[i:i] = [x]$

$del\ s[i:(j:k)]$ Elementlöschung, j und k jeweils optional

Operationen auf sequentiellen *mutable* Datentypen

Referenzen finden, Datenstruktur verändern:

`s.index(x[,i[,j]])` Index des Elements `x`

→ `s[k] == x` und `i ≤ k < j`

`s.remove(x)` `x` aus `s` löschen → `del s[s.index(x)]`

`s.extend(x)` Füge alle Elemente aus `x` an `s` an

Übung – Listen

Erstelle dir eine Liste mit diesen Elementen:

1, 2, 3, 4, 5, 6, 7, 8, 9

- Entferne das erste Element
- Entferne das erste Element und füge es hinten wieder an
- Entferne das zweite und dritte Element
- Ersetze das vierte Element mit der Länge der Liste
- Füge die Liste [10, 11, 12] elementweise hinten an
- Füge die Liste [10, 11, 12] zwischen 7. und 8. Element ein

Gib nach jedem Schritt die Liste aus.

Operationen auf sequentiellen *mutable* Datentypen

Liste als *Queue*:

`s.append(x)` x an s anfügen $\rightarrow s[\text{len}(s):\text{len}(s)] = [x]$

`s.pop([i])` Letztes Element zurückgeben und entfernen $\rightarrow$
`x = s[i]; del s[i]; return x`

```
> list = []  
> store = list.append  
> store('apple')  
> store(math.pi)  
> list  
['apple', 3.141592653589793]
```

Operationen auf sequentiellen *mutable* Datentypen

Objektreihenfolge ändern:

`s.reverse()` Elementreihenfolge umkehren

`s.sort([cmp[, key[, reverseP]]])` ...

- Beide Methode verändern die sequentielle Datenstruktur direkt
 - Kein Rückgabewert
 - Speichereffizient
- Besonderheit zu `sort(...)`: Details zur `cmp`-Funktion in *Erweiterte Datenstrukturen*

Übung – Strings umkehren

Überlegt euch ein kurzes Skript, dass einen gegebenen String mit den bisher gegebenen Mitteln umkehrt.

Aus `Here I am!` soll `!ma I ereH` werden.

Python

—

Mapping Datentyp

Dict[ictionary] (Hashmap)

Mapping Datentyp – dict

- dict bisher einziger Mapping-Datentyp in Python
- Datentyp ist *mutable*
- Besteht aus ungeordneten key : value-Paaren
 - key muss hashable sein → *immutable!!!*
 - value kann eines beliebigen Typs sein
 - Vorsicht bei Verwendung bei Zahlen als key
 - Äquivalente verweisen auf denselben value: 1 == 1.
 - float sehr ungeeignet (**floating**point number!!11ELF.)
- Standardnotation:
dict1 = {key0: value0, key1: value1, key2: value2}
- Angabe in Standardnotation oder über dict-Konstruktor

dictionary anlegen

dictionaries können über die Standardnotation oder durch Aufruf eines der folgenden Konstruktoren instantiiert werden:

```
class dict(**kwarg)
class dict(mapping, **kwarg)
class dict(iterable, **kwarg)
```

Beispiel:

```
> a = dict(one=1, two=2)
> b = {'one': 1, 'two': 2}
> c = dict(zip(['one', 'two'], [1, 2]))
> d = dict([('two', 2), ('one', 1)])
> e = dict({'one': 1, 'two': 2})
> a == b == c == d == e
```

True

dictionary auslesen und verändern

Auf values kann direkt per key zugegriffen werden (auslesen, ändern, neues Mapping hinzufügen):

```
> b = {1: 'on', 2: 'two'}
```

```
> b[1]  
'on'
```

```
> b[1] = 'one'
```

```
> b[1]  
'one'
```

```
> b[3] = 'three'
```

```
> b  
{2: 'two', 3: 'three', 1: 'one'}
```

Python

—

Set Datentypen

Mengen – Set, Frozenset

Set, Frozenset – Mengen

`set` und `frozenset` sind Container für eindeutige Elemente. Mathematisch können diese als Mengen angesehen werden.

- Menge an eindeutigen Elementen
- Jedes Objekt (Element) muss *immutable* sein
- Es existieren diverse Mengenspezifische Methoden: `union()`, `intersection()`, `issubset()`, `issuperset()`, `isdisjoint()`, ...

Es gibt zwei unterschiedliche Implementierungen für Mengen:

`set` Veränderbare Menge

`frozenset` Umveränderbare Menge → *immutable*

Set, Frozenset – Codebeispiel

```
> s = set('abc')
> s
{'a', 'b', 'c'}

> s.update('x', [8])
> s
{8, 'a', 'b', 'c', 'x'}

> s.intersection('ac8')
{'a', 'c'}
```

Set, Frozenset

- Konstruktoren: `class set([iterable]),`
`class frozenset([iterable])`
- Shorthand-Instanziierung ohne Konstruktor: `elem1, elem2, ...,`
`elemNelem1, elem2, ..., elemN`
- Bei Sets von Sets muss das referenzierte Set vom Typ `frozenset` sein

Weitere Methoden und Erläuterungen: [Python 2.7-Dokumentation](#)