Datentypen in Python

Prof. Dr. Rüdiger Weis

Beuth Hochschule für Technik Berlin

Wintersemester 2018/2019

Datentypen

- Zahlen
- Strings
- Tupel
- Listen
- Dictonaries
- Dateien
- Mengen
- Frozen Sets

Dynamische Typisierung

Dynamische Typisierung (Wikipedia)

Bei der dynamischen Typisierung (engl. dynamic typing) erfolgt die Typzuteilung der Variablen zur Laufzeit eines Programmes durch das Laufzeitsystem, z.B. eine virtuelle Maschine.

Dynamische Typisierung in Python

- Das assignment statement
  - erzeugt ein Variable und
  - weist ihr einen Wert zu.
- Keine Deklaration notwendig.
- Wert bestimmt Variablen Typ.

Typ Konvertierung

`type` Funktion liefert Typ zurück.

- str
- list
- tupel
- int
- long
- float
- complex
- set
- frozenset

Mehrfachzuweisung

```
>>> a = b = c = 69
>>> print(a)
69
>>> print(b)
69
>>> print(c)
69
```

Python Keywords

Python Keywords

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def','del',
'elif', 'else', 'except', 'exec', 'finally', 'for', 'from',
'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or',
'pass', 'print', 'raise', 'return', 'try', 'while',
'with', 'yield']
>>> len(keyword.kwlist)
31
```

Zahlen

- Integers
- Long integers
- Floats
- Complex Numbers

Integers

- Ganzzahlen
- 0*x* für Hex
- 0 für Oktal
- Explizte Konversion mit Funktion int()

```
>>> 0xF
15
>>> 011
9
```

Long Integers

- Ganzzahlen mit beliebiger Länge
- Suffix L
- Explizte Konvertierung mit Funktion long()

Beliebig Lange Ganzzahlen

```
>>> 2 ** 64
18446744073709551616L
>>> 2 ** 128
340282366920938463463374607431768211456L
```

Floats

- Gleitkommazahlen
- E Schreibweise möglich
- Explizte Konvertierung mit Funktion float()

Komplexe Zahlen

- Komplexe Zahlen werden als zwei Fliekommazahlen dargestellt.
- Imaginäre Zahlen werden mit dem Suffix "j" oder "J" gekennzeichnet.
- `complex(real,imag)` ergibt real+imagJ
- `z.real` liefert den Real-Teil.
- `z.imag` liefert den Imaginär-Teil.

Komplexes Rechnen

```
>>> 1j ** 2
(-1+0j)
>>> (1 + 1j) * (2 + 1j)
(1+3j)
>>> (1 + 1j) * 2
(2+2j)
```

Strings

- **Unveränderlich**
- Sequenz von Zeichen

Strings

- Strings können mit einfachen ('...')
  oder doppelten (" ... ") Anführungszeichen angegeben werden.
- Der Backslash maskiert Anführungszeichen.
- Bei dreifache Anführungszeichen muss das Zeilenende (engl. EOL) nicht maskiert werden.

Dreifache Anführungszeichen

```
>>> mailsignature = """
"avenidas
avenidas y flores
flores
flores y mujeres
avenidas
avenidas y mujeres
avenidas y flores y mujeres y
un admirador"
"""
>>> print(mailsignature)

"avenidas
avenidas y flores
flores
flores y mujeres
avenidas
avenidas y mujeres
avenidas y flores y mujeres y
un admirador"
```

Excape Sequenzen

\ ...

Der Backslash ermöglicht die Eingabe von Escape-Sequenzen.

- \ ' : '
- \ " : "
- \n : newline
- \t : tab
- \\ : \
- ...

Raw Strings

- r'String'
- Escape-Sequenzen werden nicht intepretiert.

Reguläre Ausdrücke

Verwenden sie grundsätzlich raw Strings,
wenn sie mit regular expressions arbeiten.

String Operatoren

String Operatoren

- +: Konkatenation
- *: Wiederholung
- Indexing
- Slicing

Eingebaute String Funktionen

Built ins

- len: Länge
- in: Enthalten-Test
- $<, >$: Lexigraphische Vergleiche

String Indizierung

- Strings können analog zu C beginnend mit dem Index 0 indiziert werden.

```
>>> wort = "Wedding"
>>> wort[0]
'W'
>>> wort[3]
'd'
```

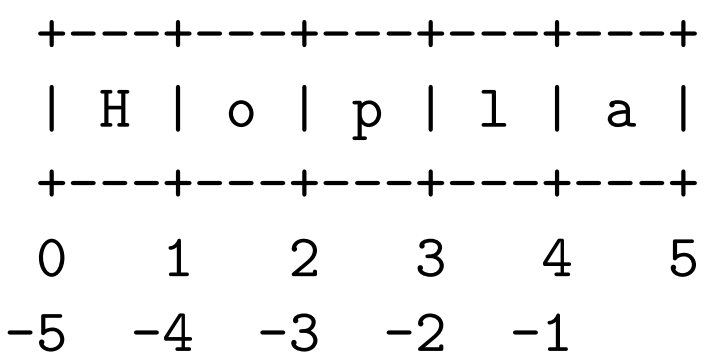
Teilbereichnotationen

```
>>> vorlesung = "Systemprogrammierung"
>>> vorlesung[4:6]
'em'
>>> vorlesung[:3]
'Sys'
>>> vorlesung[4:]
'emprogrammierung'
>>> len(vorlesung)
20
```

Negativ Indizierung

```
>>> w = "Hopla"
>>> w[-1]
'a'
>>> w[-2:]
'la'
>>> w[:-2]
'Hop'
```

Indizierung



Tupel

- **Unveränderlich.**
  - Achtung: Tupel können veränderliche Elemente enthalten.
- Sequenz mit beliebigene Elementen.
- Schachtelung möglich
- Einschluss durch ( )
- Trennung durch ,

Sonderfälle

- () : Leerer Tuppel
- (elment,) : Einelementiges Tuppel

Tupel Operatoren

Tupel Operatoren

- +: Konkatenation
- *: Wiederholung
- Indexing
- Slicing

Eingebaute Tupel Funktionen

Built ins

- len: Länge
- in: Enthalten-Test
- $<, >$: Lexigraphischer Vergleiche

Tupel Zuweisung

```
>>> a, b, c = 1, 2, 3 # Zuweisung der Reihe nach
>>> print(a, b, c)
1 2 3

>>> a, b = b, a # Vertauschen ohne Hilfsvariable
>>> print(a, b)
2 1
```

Listen

- **Veränderlich.**
- Sequenz mit beliebigene Elementen.
- Beliebige Länge
- Schachtelung möglich
- Einschluss durch [ ]
- Trennung durch ,

Listen Operatoren

Listen Operatoren

- +: Konkatenation
- *: Wiederholung
- Indexing
- Slicing

Eingebaute Listen Funktionen

Built ins

- len: Länge
- in: Enthalten-Test
- $<, >$: Lexigraphischer Vergleiche

Änderungen in Listen

Änderungen

- Indexierte Zuweisung
- Indexiertes Löschen mit `del`

Zahlenlisten mittels range

range

- `range(Ende)` liefert Zahlenliste von **0** bis **Ende-1** mit Schittweite 1.
- `range(Begin,Ende)` liefert Zahlenliste zwischen Begin und Ende-1 mit Schittweite 1.
- `range(Begin,Ende,Schrittweite)` liefert Zahlenliste zwischen Begin und Ende-1 mit angebebener Schittweite.

Listen Methoden: Elementprüfung

Elementprüfung

- `liste.index(x)` liefert den Index des ersten Elements in der Liste zurück, dessen Wert gleich x ist.
  Falls kein Element mit Wert x existiert wird ein

  `ValueError: list.index(x): x not in list.`

  geworfen.
- `liste.count(x)` liefert die Anzahl des Auftretens des Elements x in der Liste zurück.

Listen Methoden: Sortieren

Sortieren

- `liste.sort()` sortiere die Elemente der Liste.
- `liste.reverse()` invertiert die Reihenfolge der Listenelemente.

Listen Methoden: Einfügen und Löschen

Einfügen und Löschen

- `liste.insert(i, x)` fügt das Element x vor den Index i ein.
- `liste.append(x)` fügt das Element x am Ende der Liste an.
  `liste.append(x)` ist äquivalent zu `a.insert(len(a), x)`.
- `liste.remove(x)`
  - entferne das erste Element mit dem Wert x aus der Liste.
  - Falls kein Element mit Wert x existiert, liefert die remove Methode
    `ValueError: list.remove(x): x not in list`

Beispiel Einfügen und Löschen

```
>>> liste = [1, 2, 3]
>>> liste.insert(2,"Vier")
>>> liste
[1, 2, 'Vier', 3]
>>> liste.append(1)
>>> liste
[1, 2, 'Vier', 3, 1]
>>> liste.remove(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> liste.remove(1)
>>> liste
[2, 'Vier', 3, 1]
```

Listen in Tupel

Listen in Tupel bleiben veränderlich.

```
>>> tup = (1, ['Liste', 2], 'String')
>>> tup[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup[1][1] = 42
>>> tup
(1, ['Liste', 42], 'String')
```

Dictonaries

- Dictonaries sind Abbildungen von unveränderlichen Schlüssel zu beliebigen Werten.
- {Schluessel : Wert}
- Trennung durch Komma

Indizierung

- Indizierung durch unveränderliche Objekte
  - Integers
  - Tupel
  - Strings
- Kein Slicing.

Anlegen

```
>>> de2eng = {}
>>> de2eng['eins'] = 'one'
>>> de2eng['drei'] = 'three'
>>> de2eng[2] = 'two'
>>> de2eng
{'drei': 'three', 2: 'two', 'eins': 'one'}
```

Zugreifen und Löschen

```
>>> de2eng['eins']
'one'
>>> del de2eng[2]
>>> de2eng
{'drei': 'three', 'eins': 'one'}
>>> de2eng['drei'] = 3
>>> de2eng
{'drei': 3, 'eins': 'one'}
```

Dictonaries Methoden

```
>>> de2eng.keys()
['drei', 'eins']
>>> de2eng.values()
[3, 'one']
>>> de2eng.items()
[('drei', 3), ('eins', 'one')]
>>> de2eng.has_key('eins')
True
>>> de2eng.has_key('Eins')
False
```

Referenzsematik

Unterschiedliche Referenzsematik für

- Unveränderliche Objekte
  - Zahlen
  - Strings
  - Tupel
- Veränderliche Objekte
  - Listen
  - Dictories

Unveränderliche Objekte

- Zuweisung erzeugt ein neues Objekt.

```
>>> a = 1
>>> b = a
>>> a, b
(1, 1)
>>> a = a + 1
>>> a, b
(2, 1)
```

Veränderliche Objekte

- Zuweisung erzeugt Referenz.

```
>>> a = [1, 2, 3]
>>> b = a
>>> a, b
([1, 2, 3], [1, 2, 3])
>>> a[0] = 'neua'
>>> a, b
(['neua', 2, 3], ['neua', 2, 3])
>>> b[1] = 'neub'
>>> a, b
(['neua', 'neub', 3], ['neua', 'neub', 3])
```

Dictonaries

```
>>> d = {1:42, 2:23}
>>> d
{1: 42, 2: 23}
>>> a = d
>>> d[4] = 69
>>> d
{1: 42, 2: 23, 4: 69}
>>> a
{1: 42, 2: 23, 4: 69}
```

set, frozenset

`http://docs.python.org/lib/types-set.html`

A set object is an unordered collection of immutable values.

set	mutable
frozenset	immutable

set, frozenst Operations

- len(s)
  cardinality of set s
- x in s
  test x for membership in s
- x not in s
  test x for non-membership in s
- s.issubset(t)
  test whether every element in s is in t
- s.issuperset(t)
  test whether every element in t is in s

set, frozenset Operations (II)

- s.union(t)
  new set with elements from both s and t
- s.intersection(t)
  new set with elements common to s and t
- s.difference(t)
  new set with elements in s but not in t
- s.symmetric_difference(t)
  new set with elements in either s or t but not both
- s.copy()
  new set with a shallow copy of s

set Operations (I)

- s.update(t)
  update set s, adding elements from t
- s.intersection_update(t)
  update set s, keeping only elements found in both s and t
- s.difference_update(t)
  update set s, removing elements found in t
- s.symmetric_difference_update(t)
  update set s, keeping only elements found in either s or t but not in both
- s.add(x)
  add element x to set s

set Operations (II)

- s.remove(x)
  remove x from set s; raises KeyError if not present
- s.discard(x)
  removes x from set s if present
- s.pop()
  remove and return an arbitrary element from s;
  raises KeyError if empty
- s.clear()
  remove all elements from set s

Beispiel Mengen

```
>>> a = set([1, 2, 2, "Hallo"])
>>> a
set([1, 2, 'Hallo'])
>>> a.update('Python')
>>> a
set([1, 2, 'Hallo', 'h', 'o', 'n', 'P', 't', 'y'])
>>> b = set((1, 2, "t"))
>>> a.intersection_update(b)
>>> a
set([1, 2, 't'])
```

Datentyp file

- Öffnen
  - r : read
  - w : write
  - a : append
- read : Einlesen in String
- readlines : Zeilenweise einlesen in Liste von Strings

Öffnen und Schliessen

```
>>> f = open('datei.txt', 'w')
>>> print(f)
<open file 'datei.txt', mode 'w' at 0xb7df7338>
>>> f.close()
```

Lesen und Schreiben

```
>>> f = open('datei.txt', 'w')
>>> f.write("Python: ride the snake.")
>>> f.close()
>>> f = open('datei.txt')
>>> text = f.read()
>>> print(text)
Python: ride the snake.
>>> f.close()
```

Exkurs: Persistente Objekte

`http://docs.python.org/lib/persistence.html`

- pickle : Convert Python objects to streams of bytes and back.
- cPickle : Faster version of pickle, but not subclassable.
- copy_reg : Register pickle support functions.
- shelve : Python object persistence.
- marshal : Convert Python objects to streams of bytes and back (with different constraints).
- anydbm : Generic interface to DBM-style database modules.
- whichdb : Guess which DBM-style module created a given database.

Exkurs: Persistente Objekte (II)

- dbm : The standard “database” interface, based on ndbm.
- gdbm : GNU’s reinterpretation of dbm.
- dbhash : DBM-style interface to the BSD database library.
- bsddb : Interface to Berkeley DB database library
- dumbdbm : Portable implementation of the simple DBM interface.
- sqlite3 : A DB-API 2.0 implementation using SQLite 3.x.

pickle

pickle Modul

- import pickle
- pickle.dump()
- pickle.load()

pickle Beispiel

```
>>> x = ["Liste", "mit", ("Tupeln", "zum", "Beispiel")]
>>> import pickle
>>> f = open("picklebsp", "w")
>>> pickle.dump(x, f)
>>> f.close()
>>> g = open("picklebsp")
>>> y = pickle.load(g)
>>> y
['Liste', 'mit', ('Tupeln', 'zum', 'Beispiel')]
```

©opyleft

©opyleft

- Erstellt mit Freier Software
- © Rüdiger Weis, Berlin 2018
- unter der GNU Free Documentation License.