Technische Informatik 3 – Programmieren

Kapitel 1: Datentypen

Prof. Dr. Benjamin Kormann
Fakultät für Elektro- und Informationstechnik
12.10.2021

Erstes Python Programm

Hash-Bang: Wird unter Linux und Unix verwendet, um das Python Programm direkt starten zu können durch:
`$ ./myAppl.py`
anstatt
`$ python3 myAppl.py`

Definition einer Funktion mit zwei Parametern

Einrückung (Pflicht) um 4 Zeichen (Vorsicht: Tabulator)

Funktions- und Variablendefinition

Hauptprogramm (Programmstart)

```
#!/usr/bin/python3

# Funktion zur Addition von zwei Zahlen
def add(a, b):
    return a + b

# Funktion zur Subtraktion von zwei Zahlen
def sub(a, b):
    return a - b

# Start des Programms

# Aufruf von Addition
s = add(3, 7)
print("Summe von 3 und 7 ist " + str(s))

# Aufruf von Subtraktion
d = sub(17, 11)
print("Differenz von 17 und 11 ist {}".format(d))
```

Rückgabewert

Funktionsaufruf

Umwandlung der Zahl s in Datentyp String (Zeichenkette)

Ausgabe auf der Konsole

Einsetzen von Zahl d in Platzhalter {} der Zeichenkette

Datentypen und Datenstrukturen

Abstrakte Datentypen

- WAS: Festlegung der Funktionalität von Operationen (Semantik)
- Die konkrete Implementierung bleibt verborgen (Definition der Schnittstelle)

Definition Datenstruktur ist ein formalisiertes Objekt zur

- Eine Datenstruktur ist ein formalisiertes Objekt zur Speicherung, Verwaltung und dem Zugriff auf Daten.
- Die Strukturierung und Speicherung der Daten wird dabei ebenfalls festgelegt
- Eine Datenstruktur ist die Implementierung eines abstrakten Datentyps

Beispiel

- `PriorityQueue` mit Operationen als Schnittstelle wie bspw. `insert(), pull()`
- `BinaryHeap` als typische Implementierung der `PriorityQueue`

Variablen in Python

Variablen besitzen immer einen Bezeichner

- Name der Variable ist für den Zugriff erforderlich
- Variablen haben eine Identität zugeordnet
  - Kennzeichnung des verwiesenen Objekts
  - Identität entsprechen Adressstelle im Speicher
- Eine Variable ist eine Bindung an ein Objekt
- Objekte besitzen einen Datentyp

Datentypen legen die Speicherorganisation fest

- Alle Daten in Python sind Objekte
- Python nutzt die dynamische Typisierung
- Variablendeklaration ist nicht notwendig

```python
# Definition einer Variablen
variable = 17
# Ermittlung der ID (Identität)
ident = id(variable)
# Ausgabe der Identität (bspw. 4532767536)
print(ident)
```

```python
# Bindung von Objekt 17 an Variable number
number = 17
# Typ von number (hier: <class 'int'>)
print(type(number))

# Bindung von Objekt "hello" an Variable text
text = "hello"
# Typ von text (hier: <class 'str'>)
print(type(text))

# Bindung von Objekt 17.7 an Variable number
number = 17.7
# Typ von number (hier: <class 'float'>)
print(type(number))
```

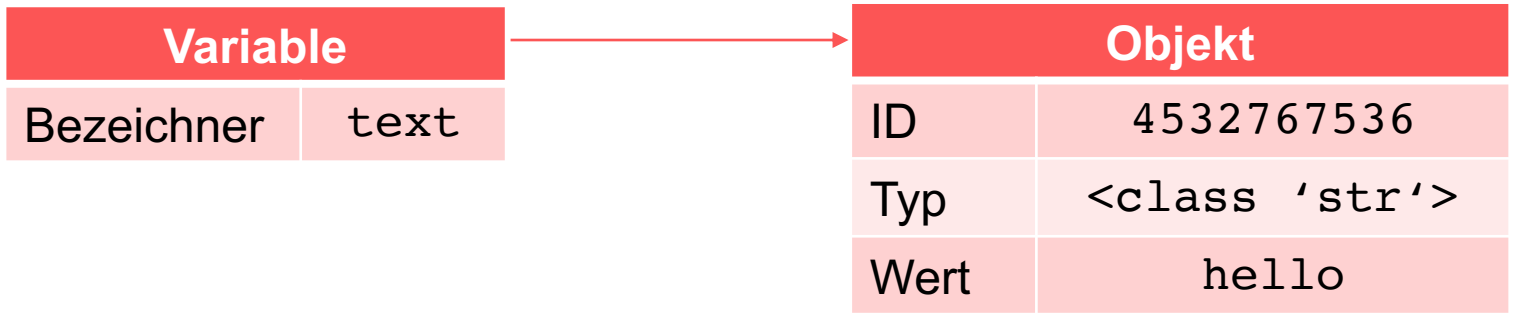
Variablen in Python
Veränderbare und unveränderbare Objekte

Veränderbare Objekte

- Werte können sich während der Laufzeit verändern
- Folgende Python Objekte sind veränderbar
  - `list`: Liste, deren Elemente `item` genannt werden
  - `dict`: Dictionary, um `key:value` Paare zu speichern
  - `set`: Sammlung von mehreren `items` in einer Variablen
  - `bytearray`: Liste von Bytes (Zahlen von 0-255)

Unveränderbare Objekte

- Werte können sich während der Laufzeit nicht verändern
- Folgende Python Objekte sind unveränderbar
  - `int`: Primitiver Datentyp für Ganzzahlen
  - `float`: Primitiver Datentyp für Gleitkommazahlen (IEEE 754)
  - `str`: Zeichenkette mit unbegrenzter Länge
  - `tuple`: Liste von `items` unterschiedlicher Datentypen
  - `frozenset`: Menge von Objekten (jedes Objekt genau einmal)



Die dynamische Typisierung in Python

Unterschied statischer zu dynamischer Typisierung

- Eine statisch typisierte Variable erhält bei Deklaration einen Typ und dieser kann sich über die Laufzeit nicht mehr verändern.
- Python nutzt eine dynamische Typisierung, d.h. der Typ kann sich verändern.

Python: Alles in ein Objekt

- Selbst Zahlen wie 14, 17 etc. werden in Python als Objekte dargestellt und verwendet
- Durch den Typ der Objekte sind die darauf anwendbaren Funktionen (Operationen) festgelegt, wie bspw. eine Addition oder Subtraktion von Ganzzahlen

```
# Anlegen von Variablen
my_var1 = 14
my_var2 = 15
my_var3 = my_var1
my_var4 = 17.2

# Ausgabe der IDs
print( id(my_var1) )
print( id(my_var2) )
print( id(14) )
print( id(my_var3) )
print( id(my_var4) )

# Dynamische Typisierung (Ausgabe von Typ und ID)
print(type(my_var2))
my_var2 = my_var4
print( id(my_var2) )
print(type(my_var2))
```

Ausgabe

```
4425226960
4425226992
4425226960
4425226960
4945936240
<class 'int'>
4945936240
<class 'float'>
```

Die dynamische Typisierung in Python
Interna bei veränderbaren und unveränderbaren Typen

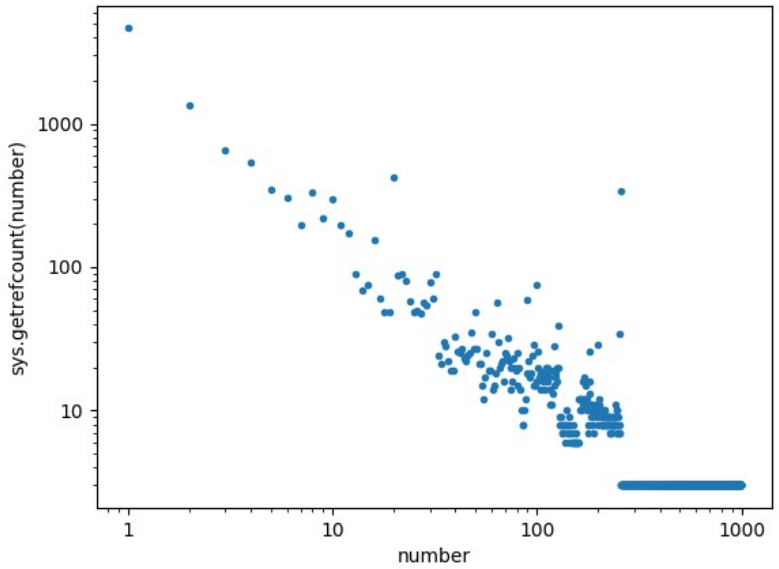
Die dynamische Typisierung erfolgt in drei Schritten

- Anlegen der Variable mit dem dazugehörigen Bezeichner
- Setzen der Referenz in der Variable auf das Objekt (bei veränderbaren Typen wird stets ein neues Objekt erzeugt)
- Erhöhen des Referenzzählers des Objektes (der Verweis auf identische Objekte spart Rechnerressourcen)

```
# Standardmodul sys einbinden
import sys

# Ausgabe der Referenzzähler
print(sys.getrefcount(1))
print(sys.getrefcount(2))
print(sys.getrefcount(17))

# Zuweisungen
var = 17
var2 = var
print()

# Ausgabe der Referenzzähler
print(sys.getrefcount(1))
print(sys.getrefcount(2))
print(sys.getrefcount(17))
```

Ausgabe

```
165
111
40

165
111
42
```



https://groverlab.org/hnbfpr/2017-06-22-fun-with-sys-getrefcount.html

Die dynamische Typisierung in Python
Vergleich von Objekten und Inhalten

Test auf Inhaltsgleichheit mit ==

- Anwendbar für alle Typen, die diesen Vergleichsoperator unterstützen bzw. definieren
- Bewertet, ob die Inhalte zweier Objekte gleich sind

```
# Anlegen von Objekten
s1 = "Hello World"
s2 = "hello world"
l1 = list(s1)
l2 = list(s1)

# Test auf Inhaltsgleichheit
print(s1 == s2)
print(s1 == s1)
print(l1 == l2)

# Ausgabe der Identitäten
print( id(s1) )
print( id(s2) )
print( id(l1) )
print( id(l2) )
```

Ausgabe

```
False
True
True
4945841136
4933630576
4946402496
4946401984
```

Test auf Adressgleichheit mit `is`

- Anwendbar für alle Typen, da ein Objektvergleich auf Basis der Identität der Objekte erfolgt (in Python ist alles ein Objekt)
- Ungleichheit kann mit `is not` geprüft werden

```
# Anlegen von Objekten
s1 = "Hello World"
s2 = "hello world"
l1 = list(s1)
l2 = list(s1)

# Test auf Adressgleichheit
print(s1 is s2)
print(s1 is s1)
print(l1 is l2)

# Ausgabe der Identitäten
print( id(s1) )
print( id(s2) )
print( id(l1) )
print( id(l2) )
```

Ausgabe

```
False
True
False
4945841136
4933630576
4945906880
4945953472
```

Adressgleichheit entspricht Inhaltsgleichheit

Übersicht der wichtigsten Datentypen in Python

Klasse	Beschreibung	numerisch[a]	veränderbar[b]	größenbestimmbar[c]	sequentiell[d]	iterierbar[e]	anordenbar[f]
• *elementare Typen*							
– *numerische Typen*							
int	ganze Zahlen	x					x
float	Fließkommazahlen	x					x
complex	komplexe Zahlen	x					
– *sonstige elementare Typen*							
bool	logische Werte (wahr/falsch)						x
NoneType	enthält nur das Symbol "nichts"						
• *Kollektionen*							
– *allgemeine Sequenzen*							
list	Liste		x	x	x	x	x
tuple	Folge (Tupel)			x	x	x	x
– *spezielle Sequenzen*							
str	String (Zeichenkette)			x	x	x	x
bytes	Tupel von Bytes			x	x	x	x
bytearray	Liste von Bytes		x	x	x	x	x
range	ganzzahliger Wertebereich			x		x	
– *Mengen*							
set	Menge		x	x		x	x
frozenset	unveränderbare Menge			x		x	x
– *Abbildungen*							
dict	Dictionary		x	x		x	

Skript TI3-Programmieren, Schöttl, Tasin

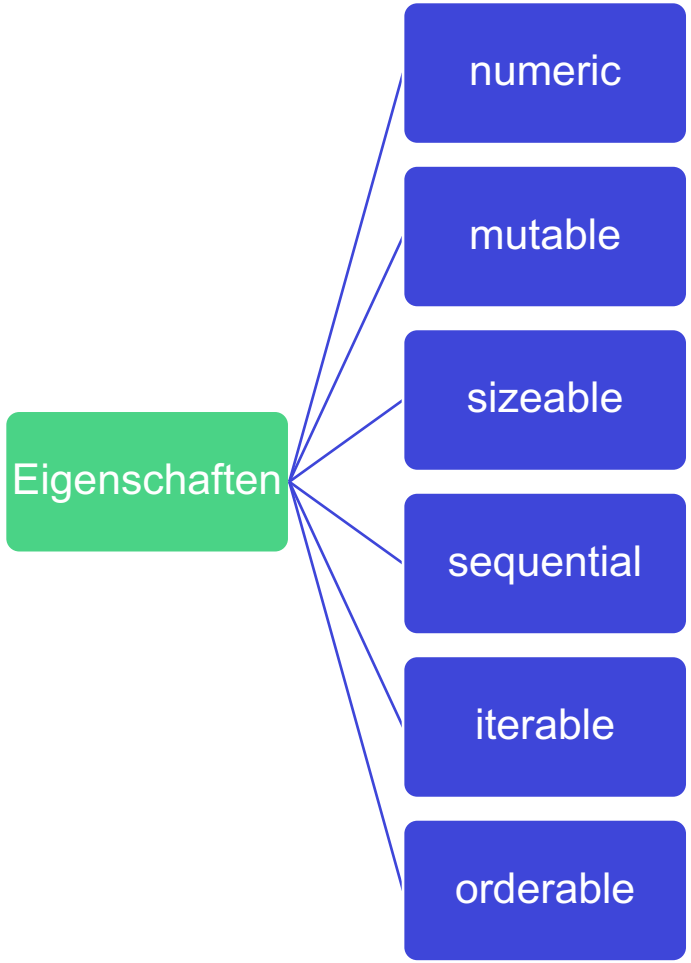
Unterscheidung der Datentypen

- Elementare Typen: Primitive Datentypen (bspw. Zahl)
- Collections: Sammlung mehrerer Daten

Beschreibung der Eigenschaften

- `numeric`: Mit diesen Objekten kann gerechnet werden
- `mutable`: Werte der Identität können sich ändern
- `sizeable`: Die Anzahl der Elemente können ermittelt werden
- `sequential`: Die Reihenfolge der Elemente ist relevant
- `iterable`: Alle Elemente können durchlaufen werden
- `orderable`: Es gibt eine Ordnung (kleiner, größer etc.)

Übersicht der wichtigsten Datentypen in Python
Zusätzliche Beschreibung der Eigenschaften



Mit diesen Objekten kann gerechnet werden

- Beispielhafte Rechenoperatoren: `+, -, *, /, //, %, **`

Werte der Identität können sich ändern

- Bei unveränderbaren Typen wird eine neue Identität erstellt

Die Anzahl der Elemente können ermittelt werden

- Mit `len(x)` kann die Anzahl der Elemente von `x` bestimmt werden

Die Reihenfolge der Elemente ist relevant

- Der Elementzugriff kann mit dem Index-Operator (`[]`) erfolgen

Alle Elemente können durchlaufen werden

- Diese Eigenschaft wird in Schleifen (bspw. `for`) genutzt

Es gibt eine Ordnung (kleiner, größer etc.)

- Beispielhafte Vergleiche: `==, <, >, <=, >=, !=`

Übersicht der wichtigsten Datentypen in Python
Heterogene und homogene Datenstrukturen

Heterogene Datenstrukturen können Elemente mit unterschiedlichem Typ aufnehmen

- Bsp.: `list, tuple, set`

```
# Heterogen: Aufnahme unterschiedlicher Typen
my_list = [1, 'two', 3, False]
my_tuple = (True, 'abc', b'\x01')
my_set = {'a', 'b', 3, False}
print(my_list)
print(my_tuple)
print(my_set)
```

Ausgabe

```
[1, 'two', 3, False]
(True, 'abc', b'\x01')
{False, 'a', 'b', 3}
```

Homogene Datenstrukturen können nur Elemente des selben Typs aufnehmen

- Bsp.: `str, bytearray`

```
# Homogen: Aufnahme identischer Typen
my_str = 'Hochschule München'
my_barr = bytearray(b'hallo') + bytearray(b'welt')
print(my_str)
print(my_barr)
```

Ausgabe

```
Hochschule München
bytearray(b'hallowelt')
```

Übersicht der wichtigsten Datentypen in Python
Elementare Datentypen - `int`

Der numerische Datentyp `int`

- ist ein unveränderbarer Datentyp
- stellt positive und negative Ganzzahlen dar

Verwendung / Operationen

- Anwendung der Grundrechenarten (`+,-,*,/`)
- Ganzzahlige Division (`//`)
- Exponent (`**`)
- Rest einer Division: Modulo (`%`)
- Konvertierung von Zeichenkette: `int(string, base)`
- Binärdarstellung: `bin(number)`
- Oktaldarstellung: `oct(number)`
- Hexadezimaldarstellung: `hex(number)`

```python
# Anlegen eines Integers (ID bspw.: 4400249264)
a = 5
print(id(a))

# Neuzuweisung Integer (neues Objekt, ID bspw.: 4400249328)
a = 7
print(id(a))

# Divison zweiter Integer (neues Objekt, ID bspw.: 4920578960)
b = 7/5
print(b)
print(id(b))

# Ganzzahlige Division
b = 7//5
print(b)
```

Ausgabe

```
4400249264
4400249328
1.4
4920578960
1
```

Übersicht der wichtigsten Datentypen in Python
Elementare Datentypen - `float`

Der numerische Datentyp `float`

- ist ein unveränderbarer Datentyp
- stellt positive und negative Gleitkommazahlen dar
- Rundungsfehler können entstehen

Verwendung / Operationen

- Anwendung der Grundrechenarten (`+,-,*,/`)
- Ganzzahlige Division (`//`)
- Exponent (`**`)
- Rest einer Division: Modulo (`%`)
  - Hinweis: Modulo auf Gleitkommazahlen wird nicht von allen Programmiersprachen unterstützt

```python
# Anlegen von und rechnen mit Gleitkommazahlen
a = 11.1
b = a + 2.2
print(b)

# Potenzrechnung
c = 2.5
e = 7
print(c**7)

# Modulo mit Gleitkommazahlen
f = 7.5 % 2.4
print(f)
```

Ausgabe

```
13.3
610.3515625
0.30000000000000027
```

Übersicht der wichtigsten Datentypen in Python
Elementare Datentypen - `float`

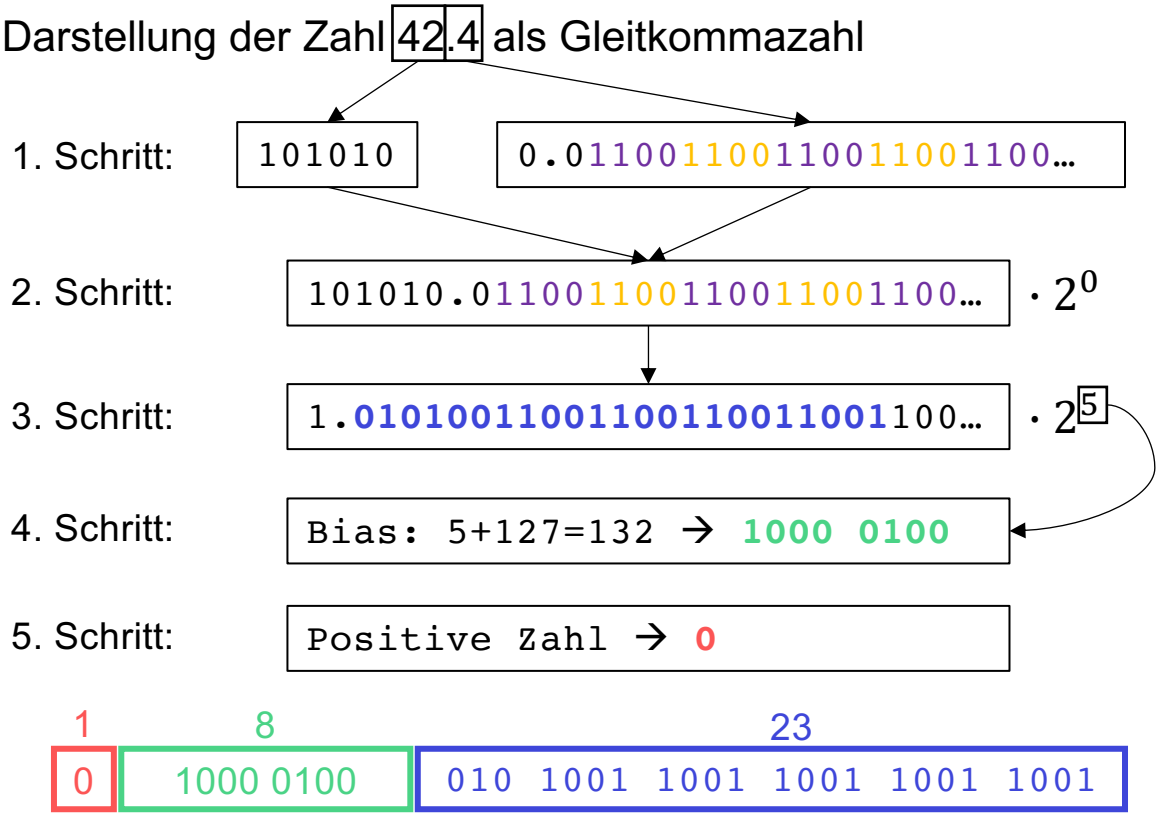
Gleitkommazahlen nach IEEE 754

- Ganzzahlen werden durch Zweierkomplement im Computer abgebildet
  - Die Zahl 42 kann binär als 32 Bit Zahl dargestellt werden: 0000 0000 0000 0000 0000 0000 0010 1010
- Gleitkommazahlen erfordern eine Codierung gemäß $z = m \cdot b^e$
  - Mantisse m, Basis b (2 für die Computerdarstellung), Exponent e und Vorzeichen v
  - Codierung für 32 Bit Gleitkommazahl

  - Die Zahl 42.4 kann somit mehrdeutig codiert werden durch bspw. $4.24 \cdot 10^1$ oder $424 \cdot 10^{-1}$
  - Eine Normalisierung liegt vor, wenn die Mantisse vor dem Komma genau eine Ziffer ≠ 0 hat

Codierung einer Gleitkommazahl nach IEEE 754

- Darstellung der Zahl 42.4 als Gleitkommazahl



- Dezimalwert: 42.40000152587890625
- Fehler durch Codierung: 0.00000152587890625

Numerische Funktion in Python
Auswahl häufig benötigter Funktionen

Operation	Beschreibung
`abs(x)`	Absolutbetrag von x.
`divmod(x, y)`	Bestimmt das Paar (x // y, x % y).
`pow(x, e)`	Berechnet den Wert x^e.
`math.trunc(x)`	Schneidet x auf einen Integer ab.
`round(x[, n])`	Rundet x auf n Zeichen mit n=0 als default.
`math.floor(x)`	Der größte Integer <= x.
`math.ceil(x)`	Der kleinste Integer >= x.
`math.sqrt(x)`	Berechnet die Quadratwurzel von x.
`math.log(x[, base])`	Natürlicher Logarithmus von x zur Basis e, wenn kein base Parameter angegeben wird.
`math.cos(x)`	Gibt den Kosinus von x im Bogenmaß zurück.
In `math` sind viele weitere Operationen vorhanden. Die Konstante π kann mit `math.pi` genutzt werden.	

Übersicht der wichtigsten Datentypen in Python
Elementare Datentypen - `complex`

Der numerische Datentyp `complex`

- ist ein unveränderbarer Datentyp
- stellt komplexe Zahlen mit Real- und Imaginärteil dar
- Genauigkeit von `float`

Verwendung / Operationen

- Anwendung der Grundrechenarten (`+,-,*,/`)
- Ganzzahlige Division (`//`)
- Exponent (`**`)
- Der Imaginärteil (`j`) muss vorliegen

```python
# Anlegen von und rechnen mit komplexen Zahlen
a = 5.1 + 1j
print(a)
a += 2j
print(a)

# Division
b = a / 2
print(b)
```

Ausgabe

```
(5.1+1j)
(5.1+3j)
(2.55+1.5j)
```

Übersicht der wichtigsten Datentypen in Python
Elementare Datentypen - `bool`

Der numerische Datentyp `bool`

- ist ein unveränderbarer Datentyp
- stellt einen logischen Wert (`True`, `False`) dar
- damit kann nicht gerechnet (`+,-,*,/`) werden

Verwendung / Operationen

- Logische Verknüpfungen gemäß der boolschen Aussagenlogik (`and`, `or`, `not`)
- Operatorenpriorität: `not` vor `and` vor `or`
- Empfehlung: Klammerung zur expliziten Priorität

```python
# Anlegen von boolschen Variablen
a = True
b = False

# Einfache Verknüpfungen
print(a or b)
print(a and b)
print(not b)

# Komplexere Verknüpfungen (Priorität)
print(b or b and a)
print((b or a) and not b)
```

Ausgabe

```
True
False
True
False
True
```

Übersicht der wichtigsten Datentypen in Python
Collections - `list`

Der allgemeine Collections Typ `list`

- ist ein veränderbarer Datentyp
- stellt eine Liste von Elementen dar
- sizeable, sequential, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Reihenfolge ist von Bedeutung
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Selektion (`[]`)
- Konkatenation (+)
- Anfügen (`append()`), Entfernen (`pop()`)
- Einfügen (`insert()`), Verbinden (`zip()`)

```python
# Anlegen einer list
l = ['Python', 'ist', 'eine', 'Programmiersprache']
print(l)
print(l[2])

# Einfügen von Elementen
l.insert(3, 'tolle')
print(l)

# Konkatinieren, Entfernen und Anfügen von Elementen
l[2] = 'kein'
l[3] = l[3] + 's'
print(l)
l.pop()
print(l)
l.append('Haustier')
print(l)

# Verbinden von zwei Paaren zu einer Liste
n = list( zip([1, 2, 3, 4, 5], l) )
print(n)
```

Ausgabe

```
['Python', 'ist', 'eine', 'Programmiersprache']
eine
['Python', 'ist', 'eine', 'tolle', 'Programmiersprache']
['Python', 'ist', 'kein', 'tolles', 'Programmiersprache']
['Python', 'ist', 'kein', 'tolles']
['Python', 'ist', 'kein', 'tolles', 'Haustier']
[(1, 'Python'), (2, 'ist'), (3, 'kein'), (4, 'tolles'),
(5, 'Haustier')]
```

Übersicht der wichtigsten Datentypen in Python
Collections - `tuple`

Der allgemeine Collections Typ `tuple`

- ist ein unveränderbarer Datentyp
- stellt eine Liste von Elementen dar
- sizeable, sequential, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Reihenfolge ist von Bedeutung
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Selektion (`[]`), Konkatenation (+), Verbinden (`zip()`)

Unterschied zu `list`

- `tuple` ist unveränderbar und somit effizienter

```python
# Anlegen eines tuple
t1 = ('e10', False, 1.199)
t2 = ('e5', True, 1.219)
print(t1)
print(t2[1])

# Verbinden von zwei Paaren
desc = ('Sorte', 'Eignung', 'Preis')
n = list( zip(desc, t1) )
print(n)

# Einfaches tuple
a = 1
b = 2
t3 = a,b
print(t3)
```

Ausgabe

```
('e10', False, 1.199)
True
[('Sorte', 'e10'), ('Eignung', False), ('Preis', 1.199)]
(1, 2)
```

Übersicht der wichtigsten Datentypen in Python
Collections (spezielle Sequenzen) - `str`

Die spezielle Sequenz `str`

- ist ein unveränderbarer Datentyp
- stellt eine Zeichenkette (String) dar
- sizeable, sequential, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Reihenfolge ist von Bedeutung
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Selektion (`[ ]`), Konkatenation (+)
- Sonderzeichen: `\n` (new line), `\t` (tab)
- Texte mit ‘ ‘ oder “ “ (“ kann in ‘ vorkommen)
  - Mehrzeilige Zeichenketten mit “““

```python
# Anlegen von Zeichenketten
s1 = 'Das ist ein Text'
s2 = 'Das ist ein "wichtiger" Text'
s3 = """Das ist ein
     mehrzeiliger Text"""
s4 = 'Hier ist ein \nZeilenumbruch integriert.'
print(s1)
print(s2)
print(s3)
print(s4)

# Konkatinieren und Selektieren von Zeichenketten
s5 = 'Das ist'
s6 = 'ein Text'
s7 = s5 + ' ' + s6
print(s7)
print(s7[2])
```

Ausgabe

```
Das ist ein Text
Das ist ein "wichtiger" Text
Das ist ein
     mehrzeiliger Text
Hier ist ein
Zeilenumbruch integriert.
Das ist ein Text
s
```

Übersicht der wichtigsten Datentypen in Python
Collections (spezielle Sequenzen) - `bytes`

Die spezielle Sequenz `bytes`

- ist ein unveränderbarer Datentyp
- stellt ein `tuple` von bytes dar (Werte 0 bis 255)
- sizeable, sequential, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Reihenfolge ist von Bedeutung
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Selektion (`[ ]`), Konkatenation (+)
- Eingabe durch vorangestelltem Buchstaben b
- `bytes()` konvertiert `int-tuple/list` in `bytes`

```
# Anlegen eines byte tuple
b1 = b'hallo'
print(b1)
print(b1[2])

# Konkatenieren von byte tuple
b2 = b'welt'
b3 = b1 + b' ' + b2
print(b3)
```

Ausgabe

```
b'hallo'
108
b'hallo welt'
```

Übersicht der wichtigsten Datentypen in Python
Collections (spezielle Sequenzen) - `bytearray`

Die spezielle Sequenz `bytearray`

- ist ein veränderbarer Datentyp
- stellt eine Liste von bytes dar (Werte 0 bis 255)
- Anwendung für hardwarenahe Programmierung
- sizeable, sequential, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Reihenfolge ist von Bedeutung
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Selektion (`[ ]`), Konkatenation (+)
- Wird durch `bytearray()` aus `bytes` erzeugt

```
# Anlegen eines bytearray
b1 = b'hallo'
barr1 = bytearray(b1)
print(barr1)
print(barr1[2])

# Konkatenieren und Verändern von bytearray
b2 = b'welt'
barr2 = bytearray(b2)
barr3 = barr1 + bytearray(b' ') + barr2
barr3[0] = 72
barr3[6] = 87
print(barr3)
```

Ausgabe

```
bytearray(b'hallo')
108
bytearray(b'Hallo Welt')
```

Übersicht der wichtigsten Datentypen in Python
Collections (spezielle Sequenzen) - `range`

Die spezielle Sequenz `range`

- ist ein unveränderbarer Datentyp
- Definiert einen ganzzahligen Wertebereich (Startwert, Endwert, Schrittweite)
  - Der Endwert ist nicht mehr Teil des Wertebereichs
- sizeable und iterable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Elemente können durchlaufen (Iterator) werden

Verwendung / Operationen

- Liste mit Werten (`list()`)
- Iterator in einer `for`-Schleife (`for i in range(42):`)

```python
# Wertebereich von [3, 7) --> 3, 4, 5, 6
r1 = range(3, 7)
print(r1)

# Wertebereich von [3, 13) --> 3, 5, 7, 9, 11
r2 = range(3, 13, 2)
print(r2)

# Wertebereich von [13, 3) --> 13, 11, 9, 7, 5
r3 = range(13, 3, -2)
print(r3)

# Werte als list
l = list(r3)
print(l)

# Wertebereich von [0, 5] --> 0, 1, 2, 3, 4
r4 = range(5)
print(list(r4))
```

Ausgabe

```
range(3, 7)
range(3, 13, 2)
range(13, 3, -2)
[13, 11, 9, 7, 5]
[0, 1, 2, 3, 4]
```

Übersicht der wichtigsten Datentypen in Python
Collections (Mengen) - `set`

Die Menge `set`

- ist ein veränderbarer Datentyp
- nimmt eine Menge von Objekten auf
- jedes Objekt kann maximal einmal enthalten sein
- sizeable, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Vereinigungsmenge (`|`), Schnittmenge (`&`)
- Differenz (`-`), symmetrische Differenz (`^`)
- Leere Menge (`set()`)

```python
# Anlegen von set
m1 = {'b', 11, 'k', 19}
m2 = {'k', 17, 'm', 6, 'k'} # Duplikat
print(m1)
print(m2)

# Vereinigung
print(m1 | m2)
# Schnittmenge
print(m1 & m2)
# Differenz
print(m1 - m2)
# Symmetrische Differenz
print(m1 ^ m2)
```

Ausgabe

```
{'b', 11, 19, 'k'}
{'m', 17, 'k', 6}
{'m', 6, 'b', 11, 'k', 17, 19}
{'k'}
{'b', 19, 11}
{'m', 6, 'b', 11, 17, 19}
```

Übersicht der wichtigsten Datentypen in Python
Collections (Mengen) - `frozenset`

Die Menge `frozenset`

- ist ein unveränderbarer Datentyp
- nimmt eine Menge von Objekten auf
- jedes Objekt kann maximal einmal enthalten sein
- sizeable, iterable und orderable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Elemente können durchlaufen (Iterator) werden
  - Eine Sortierung ist möglich

Verwendung / Operationen

- Vereinigungsmenge (`|`), Schnittmenge (`&`)
- Differenz (`-`), symmetrische Differenz (`^`)
- Leere Menge (`frozenset()`)

Unterschied zu `set`

- `frozenset` ist unveränderbar, aber gleich effizient

```python
# Anlegen von frozenset
m1 = frozenset({'b', 11, 'k', 19})
m2 = frozenset({'k', 17, 'm', 6, 'k'}) # Duplikat
print(m1)
print(m2)

# Vereinigung
print(m1 | m2)
# Schnittmenge
print(m1 & m2)
# Differenz
print(m1 - m2)
# Symmetrische Differenz
print(m1 ^ m2)
```

Ausgabe

```
frozenset({'b', 11, 19, 'k'})
frozenset({'m', 17, 'k', 6})
frozenset({'m', 6, 'b', 11, 'k', 17, 19})
frozenset({'k'})
frozenset({'b', 19, 11})
frozenset({'m', 6, 'b', 11, 17, 19})
```

Übersicht der wichtigsten Datentypen in Python
Collections (Mapping) - `dict`

Das Mapping `dict`

- ist ein veränderbarer Datentyp
- nimmt `key:value` Paare auf (Mapping)
- der `key` (Schlüssel) muss unveränderbar sein
- sizeable und iterable
  - Die Anzahl der Elemente ist unbegrenzt
  - Die Elemente können durchlaufen (Iterator) werden

Verwendung / Operationen

- Selektion (`[]`)
- Hinzufügen (`[]`)
- Schlüssel abfragen (`keys()`)
- Alle Elemente als `key:value` Paare (`items()`)

```python
# Anlegen eines dict
d = {'Pizza Marinara': 4.50, 'Pizza Napoli': 4.50}
print(d['Pizza Marinara'])

# Element hinzufügen
d['Spaghetti Carbonara'] = 6.00
print(d)

# Ein Element löschen
del d['Pizza Marinara']
print(d)

# Schlüssel abfragen
print(d.keys())

# Alle Elemente abfragen
print(d.items())

# Elemente im dict löschen
d = {}
print(d)
```

Ausgabe

```
4.5
{'Pizza Marinara': 4.5, 'Pizza Napoli': 4.5, 'Spaghetti Carbonara': 6.0}
{'Pizza Napoli': 4.5, 'Spaghetti Carbonara': 6.0}
dict_keys(['Pizza Napoli', 'Spaghetti Carbonara'])
dict_items([('Pizza Napoli', 4.5), ('Spaghetti Carbonara', 6.0)])
{}
```

Elementprüfung bei iterierbaren Klassen
Der `in` Operator

Überprüfung, ob ein Element Teil einer Struktur bzw. in der Struktur enthalten ist

- Beispiel mit `list`, `tuple`, `str` und `dict` (`in` bezieht sich auf den `key` der `key:value` Paare)

```
# Anlegen von list, tuple, str und dict
my_list  = [47, 11, 8, 15]
my_tuple = ('abc', 123, True, False, b'\x01\x02\x03')
my_str   = 'Hochschule München - Fakultät 04'
my_dict  = {'e5': 1.459, 'diesel': 1.259}

# Elementprüfungen
print(47 in my_list)
print(99 not in my_list)
print()

print('abc' in my_tuple)
print('True' not in my_tuple)
print(bytes([1,2,3]) in my_tuple)
print()

print('04' in my_str)
print('-' in my_str)
print()

print('e5' not in my_dict)
print(1.459 in my_dict)
print('e10' in my_dict)
```

Ausgabe

```
True
True

True
True
True

True
True

False
False
False
```

Das Python Datenmodell

Das Datenmodell beschreibt die Eigenschaften von Typen in Form von sog. Protokollen

- So müssen in Python iterierbare Objekte das `iterator protocol` erfüllen.
  - Dazu müssen die Methoden `__iter__()` und `__next__()` (`special methods`) implementiert werden.
  - Mehr Informationen zum `iterator protocol` unter https://wiki.python.org/moin/Iterator
- Übersicht zum Datenmodell unter https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy

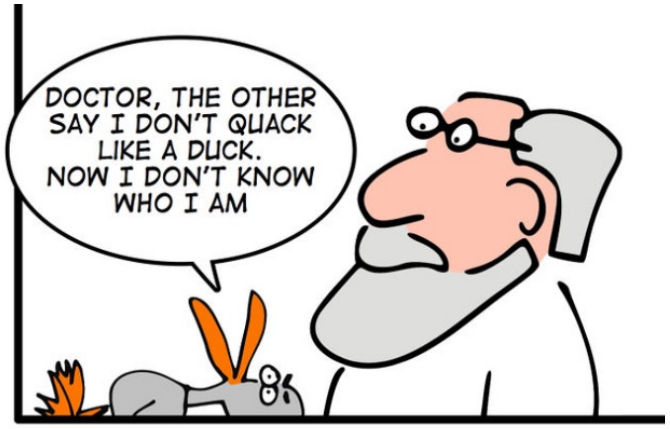
Protokolle werden in Python über sog. Abstract Base Classes (ABC) bereitgestellt

- Typen mit der Eigenschaft **`sizeable`** müssen die ABC **`Sized`** erfüllen, welche die Methode **`__len__()`** repräsentiert
  - Größe einer Liste `l1`: `len(l1)`
  - Größe eines Sets `s1`: `len(s1)`
- Übersicht zu den ABCs für Container unter https://docs.python.org/3/library/collections.abc.html

Duck Typing, LBYL und EAFP
Grundkonzept hinter Duck Typing

Duck Typing ist ein Konzept der Objektorientierung

- Bei Programmiersprache mit statischer Typisierung können Typfehler bereits zur Compilezeit ermittelt werden
  - Ada hat eine starke Typisierung und wird verstärkt im sicherheitskritischen Bereich eingesetzt (bspw. Luft-/Raumfahrt)
- Duck Typing kommt bei Sprachen mit dynamischer Typisierung zum Einsatz
  - Die Typ-Entscheidung erfolgt nicht aufgrund der Strukturbeschreibung (Klasse), sondern durch die Existenz von Methoden
  - Das Prinzip wird im Kapitel Klassen und OOP nochmal aufgegriffen



UNTYPED DUCK
https://mythinkpond.com/post/great-article-on-duck-typing-in-javascript/

Namensgebung von Duck Typing durch Gedicht von James Whitcomb Rileys

- „When I see a bird that walks like a duck and swims
  like a duck and quacks like a duck, I call that bird a duck.“

Duck Typing, LBYL und EAFP
Der (Non-)Pythonic Ansatz

LBYL vs. EAFP

- „Look before you leap“: Bei diesem Programmierstil werden Vorbedingungen vor einem Aufruf explizit überprüft.
- „It‘s easier to ask for forgiveness than permission“: Dieser Python typische Programmierstil unterstellt, dass Vorbedingungen erfüllt sind und fängt sog. Exceptions (späteres Kapitel) im Fehlerfall ab.

LBYL (Non-Pythonic)

```
# Dictionary anlegen
my_dict  = {'e5': 1.459, 'diesel': 1.259}

# Der Non-Pythonic Ansatz (LBYL)
if 'e10' in my_dict:
    print('Preis für E10: ' + str(my_dict['e10']))
else:
    print('kein Preis für E10 gefunden')
```

Ausgabe

```
kein Preis für E10 gefunden
```

EAFP (Pythonic)

```
# Dictionary anlegen
my_dict  = {'e5': 1.459, 'diesel': 1.259}

# Der Pythonic Ansatz (EAFP)
try:
    print('Preis für E10: ' + str(my_dict['e10']))
# Falls der Schlüssel 'e10' nicht vorhanden ist
# wird die Exception KeyError abgefangen (forgiveness)
except KeyError:
    print('kein Preis für E10 gefunden')
```

Ausgabe

```
kein Preis für E10 gefunden
```

Zusammenfassung

Variablen besitzen immer einen Bezeichner

- Name der Variable ist für den Zugriff erforderlich
- Variablen haben eine Identität zugeordnet
- Eine Variable ist eine Bindung an ein Objekt
- Objekte besitzen einen Datentyp
- Dynamische Typisierung, Duck Typing, LBYL, EAFP

Die wichtigsten Datentypen in Python

- Fünf elementare Datentypen: `int, float, complex, bool, NoneType`
- Neun Collections: `list, tuple, str, bytes, bytearray, range, set, frozenset, dict`
- Auswahl der Datentypen je nach Eigenschaft: `numeric, mutable, sizeable, sequential, iterable, orderable`