Python – Datentypen und Kontrollstrukturen

Stephan Karrer

Autor: Stephan Karrer

Ein Programmbeispiel

```
# Sample Comment

# Variablen
string1 = "Stephan"
string2 = 'Karrer'
numbers = range(1 , 10)

# Schleife
for i in numbers:
    # Anweisungen
    print("processing number:", i)

    # Fallunterscheidung
    if i % 2 == 0:
        print("even", i)
    else:
        print("odd", i)

# Anweisung nach der Schleife
print(string1, string2, "finished")
```

- Kommentar geht bis zum Ende der Zeile
- Jede Anweisung üblicherweise auf eigener Zeile (oder Komma-Separiert)
- Variablen ohne Typ-Angabe
- Groß/Klein-Schreibung ist relevant
- Bezeichner beginnen mit einem Buchstaben oder dem Unterstrich und dürfen neben diesen Zeichen auch Ziffern enthalten
- Einrückungen definieren die Anweisungsblöcke
- Köpfe von Schleifen und bedingten Anweisungen werden durch Doppelpunkt abgeschlossen

Variablen

```
a = 'Zeichenkette'   # str
print(a)
a = [1, 2, 3]       # list
print(len(a))

(a,b) = ('abc', 10)   # Tupel-Zuweisung
print(a,b)

(a,b) = (b, a)      # Werte-Tausch
print(a,b)

a = b = 'otto'      # Mehrfachzuweisung
print(a,b)
```

- Typ der Variablen ergibt sich zur Laufzeit anhand der Zuweisung
- Variablen sind nicht Typ-gebunden
- Natürlich muss der Typ zur Laufzeit zu den auszuführenden Operationen passen
- Mehrfachzuweisung und Tupel-Verwendung sind möglich

Ein/Ausgabe

```
eingabe = input("Bitte Eingabe: ")
print("Typ der Eingabe:", type(eingabe))  # <class 'str'>

zahl_als_string = input("Bitte Zahl: ")
zahl = int(zahl_als_string)     # Cast
print("Typ von zahl", type(zahl))        # <class 'int'>
print(zahl – 1)        # Subtraktion nur für Zahlen möglich
```

- Python bringt eine Vielzahl von eingebauten Funktionen mit https://docs.python.org/3/library/functions.html
- Für einfache Ein- und Ausgaben sind wichtig:
  - input — fordert den Benutzer zur Eingabe auf (Zeichenkette)
  - print — gibt die Argumente durch Leerzeichen getrennt als Zeichenketten aus
  - type — liefert zur Laufzeit den Typ
  - int — macht aus Zeichenkette eine Ganzzahl (Objekt-Initialisierer), gibt es analog auch für andere Typen

Builtin-Datentypen

- Wie bei den Funktionen bringt Python auch einige eingebaute Standard-Datentypen mit https://docs.python.org/3/library/stdtypes.html
- Zu den wichtigsten gehören:
  - Wahrheitswerte (**bool**)
  - Zahlen (**int, float, complex**)
  - Zeichenketten (**str**)
  - Sequenzen (**list, tuple, range**)
  - Mengen (**set, frozenset**)
  - Maps (**dict**)
- Diese werden ergänzt durch Datentypen (Klassen) aus den Bibliotheken und potentiell durch uns programmierte Datentypen (Klassen)

Ganzzahlen

Bezeichner	int	
Literale	1, -2, 123456789123455 0o12, 0o677 0x12, 0x67FF 0b110011111, 0b1001	dezimal oktal hexadezimal binär

Beispiele	i = 1; a = 0xFF;
	k = 100_000 # Unterstrich wird durch Compiler ignoriert
	b = 0b1001_0001
	
	# Führende 0 ist nicht erlaubt
	~~x = 012~~	

- Wertebereich ist nur den vorhandenen Speicher begrenzt
- Unterstrich ist nur innerhalb des Ziffergebirges erlaubt
- Führende 0 ist nicht erlaubt

Gleitpunktzahlen

Bezeichner	float
Literale	1.1, 0.23, 123., 1., 0023.2 dezimal 1.0e3 # (represents 1.0×10^{3}, or 1000.0) 1.166e-5 # (represents 1.166×10^{-5}, or 0.00001166) 6.02214076e+23 # (represents 6.02214076×10^{23}, # or 602214076000000000000000.)

Beispiele	i = 96_485.332_123 # Unterstrich wird durch Compiler ignoriert
	j = 3.14_15_93

- Wertebereich ist abhängig von den zugrundeliegenden C-Typ
- Unterstrich ist nur innerhalb des Ziffergebirges erlaubt
- Führende 0 ist erlaubt

Operationen mit Zahlen

Operation	Ergebnis
- x	Negation
+ x	x bleibt unverändert
x + y	Addition
x - y	Subtraktion
x * y	Produkt
x / y	Division – liefert float
x // y	Ganzzahldivision (Division mit Rest)
x % y	Rest bei der Division (Modulo)
x ** n	n-te Potenz von x
pow(x,n)	n-te Potenz von x
abs(x)	Absolutbetrag
round(x [,n])	Rundung
int(x)	Cast (Objektinitalisierung)
float(x)	Cast (Objektinitalisierung)

- Es gelten die üblichen Vorrangregeln, ansonsten Klammerung mit (...)
- Weitere stehen via Modul math zur Verfügung

Decimal als Datentyp

```
print(0.1 + 0.1 + 0.1 - 0.3)        # 5.551115123125783e-17
print(0.1 + 0.1+ 0.1+ 0.1+ 0.1+ 0.1+ 0.1+ 0.1+ 0.1+ 0.1)   #
0.9999999999999999
print(round(2.5))     # 2
print(round(3.5))     # 4

import decimal

print(decimal.getcontext())
decimal.getcontext().rounding=decimal.ROUND_DOWN
x =  decimal.Decimal("0.1")
print(x + x + x - decimal.Decimal("0.3"))   # 0.0
print(round(decimal.Decimal("2.5")))     # 2
print(round(decimal.Decimal("3.5")))     # 4
print(round(x+x+x+x+x))     # 0 - Rundung bezieht sich auf
                            # das Ergebnis von Arithmetik
```

- Für kaufmännische Anwendungen ist float wenig geeignet!
- Python bringt mit Decimal aus der Bibliothek einen geeigneten Typ mit https://docs.python.org/3/library/decimal.html#

Zeichenketten

Bezeichner	str
Literale	'a', 'Das', "Hallo", """Dies ist ein mehrzeiliger String"""
Operatoren	+ Konkatenation * Vervielfachung

Beispiele	s = "Hallo"
	s = 'Say "Hello", please.'
	char = 'B\u00E4h' # Bäh
	x = s + ' ' + char
	("spam " "eggs") == "spam eggs"
	line = '*' * 50

- Zeichen können auch als Unicode-Sequenz (z.B. '\u00E4') eingegeben werden und es gibt weitere Möglichkeiten:
  https://docs.python.org/3/reference/lexical_analysis.html#strings
- Vorsicht: In Ausdrücken werden einzelne literale Zeichenketten automatisch konkateniert

Escape-Squenzen

Escape Sequence	Meaning
`\<newline>`	Ignored end of line
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\ooo`	Octal character
`\xhh`	Hexadecimal character
`\N{name}`	Named Unicode character
`\uxxxx`	Hexadecimal Unicode character
`\Uxxxxxxxx`	Hexadecimal Unicode character

Prefixed-Strings

```
# Raw String: verhindert die Interpretation des \
r'\d{4}-\d{2}-\d{2}'
R'\d{4}-\d{2}-\d{2}'

# Strings mit Format-Spezifikationen
who = 'nobody'
nationality = 'Spanish'
f'{who.title()} expects the {nationality} Inquisition!'
```

Strings als Sequenzen

```
word = 'Python'

# Zugriff via Index
word[0]           # character in position 0: P
word[5]           # character in position 5: n

word[-1]          # last character: n
word[-2]          # second-last character: o

# Slicing
word[0:2]     # characters from position 0 (included) to 2 (excluded)
word[2:5]     # characters from position 2 (included) to 5 (excluded)
word[:2]      # character from the beginning to position 2 (excluded)
word[4:]      # characters from position 4 (included) to the end
word[-2:]     # characters from the second-last (included) to the end
```

- Strings sind wie einige andere (Listen, Tupel, …) Sequenz-Typen und weisen die allen Sequenzen gemeinsame Funktionalität auf

Methodik von Strings

```
wort = 'Python'
wort[0] = 'B'
```

```
Liefert zur Laufzeit Fehler:
Traceback (most recent call last):
  File "E:\stephan\workspaces\pycharm\PythonProject\Zahlen.py",
line 74, in <module>
    wort[0] = 'B'
    ~~~~^^^
TypeError: 'str' object does not support item assignment
```

- Strings können aber nicht direkt manipuliert werden (sind immutable)
- Verarbeitung geht nur über die Methodik der str-Klasse
  https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str

Gemeinsame Sequenz-Operationen

Operation	Result
`x in s`	`True` if an item of *s* is equal to *x*, else `False`
`x not in s`	`False` if an item of *s* is equal to *x*, else `True`
`s + t`	the concatenation of *s* and *t*
`s * n` or `n * s`	equivalent to adding *s* to itself *n* times
`s[i]`	*i*th item of *s*, origin 0
`s[i:j]`	slice of *s* from *i* to *j*
`s[i:j:k]`	slice of *s* from *i* to *j* with step *k*
`len(s)`	length of *s*
`min(s)`	smallest item of *s*
`max(s)`	largest item of *s*

Listen (Klasse list)

```
# Deklaration von Listen: via []
squares = [1, 4, 9, 16, 25]
mixed = ["Hallo", 2, 4, 6, 8]

# Deklaration von Listen: via [x for x in iterable]
liste = [x for x in "Hallo"]
print(liste)

# Deklaration von Listen: via Konstruktor list() or list(iterable)
liste = list("Hallo")
print(liste)
```

- Listen benutzen eckige Klammern
- Listen müssen nicht den gleichen Inhaltstyp haben
- Listen sind Sequenzen und haben deshalb wie Strings die gemeinsamen Sequenz-Operationen (Index, Slicing, ...)

Listen sind veränderbar

```
cubes = [1, 8, 27, 65, 125]   # something's wrong here
cubes[3] = 64                 # replace the wrong value
cubes.append(216)            # add the cube of 6

# Zuweisung via Slice
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
letters[2:5] = ['C', 'D', 'E']      # ['a', 'b', 'C', 'D', 'E', 'f', 'g']
letters[2:5] = []                   # ['a', 'b', 'f', 'g']
letters[:] = []                     # []
```

- Listen und andere veränderbare Sequenztypen haben zusätzliche gemeinsame Funktionalitäten

Gemeinsame Operatoren von veränderbaren Sequenztypen

Operation	Result
`s[i] = x`	item *i* of *s* is replaced by *x*
`del s[i]`	removes item *i* of *s*
`s[i:j] = t`	slice of *s* from *i* to *j* is replaced by the contents of the iterable *t*
`del s[i:j]`	removes the elements of `s[i:j]` from the list (same as `s[i:j] = []`)
`s[i:j:k] = t`	the elements of `s[i:j:k]` are replaced by those of *t*
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list
`s += t`	extends *s* with the contents of *t* (for the most part the same as `s[len(s):len(s)] = t`)
`s *= n`	updates *s* with its contents repeated *n* times

- Zusätzlich gibt es noch gemeinsame Methodik
  https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types

Verkürzte Zuweisungen

Bezeichner	str
=	a = b ;
+=	a += b ; // entspricht: a = a + b ;
-=	a -= b ; // entspricht: a = a - b ;
*=	a *= b ; // entspricht: a = a * b ;
/=	a /= b ; // entspricht: a = a / b ;
%=	a %= b ; // entspricht: a = a % b ;
…	

- Sofern möglich, sollten immer die verkürzten Varianten benutzt werden, da der Python-Interpreter in diesem Fall den 2-mal referenzierten Wert nur einmal auswertet (ziemlich schwache Leistung)

Tupel (Klasse tuple)

```
# Deklaration von Tupeln: via ()
squares = (1, 4, 9, 16, 25)
mixed = ("Hallo", 2, 4, 6, 8)

# Deklaration von Tupeln: via Konstruktor tuple() or tuple(iterable)
tupel = list("Hallo")
print(liste)
```

- Tupel benutzen runde Klammern
- Tupel sind nicht veränderbare Sequenzen
- Tupel müssen nicht den gleichen Inhaltstyp haben
- Tupel sind Sequenzen und haben deshalb wie Strings und Listen die gemeinsamen Sequenz-Operationen (Index, Slicing, …) bzgl. immutable Sequenzen

Noch ein immutable Sequenz-Typ: Ranges

```
list(range(10))             # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
list(range(1, 11))          # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
list(range(0, 30, 5))       # [0, 5, 10, 15, 20, 25]
list(range(0, 10, 3))       # [0, 3, 6, 9]
list(range(0, -10, -1))     # [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
list(range(0))              # []
list(range(1, 0))           # []
sum(range(1,10))            # 45
```

- Liefern fortlaufende Ganzzahlen (oder auch dynamische Sequenzen von anderen Typen, sofern die Typklassen entsprechende Methodik vorsehen.
- Die Schrittweite (auch negativ) kann durch den optionalen 3. Parameter vorgegeben werden.
- Brauchen wenig Speicher, da die Werte fortlaufend generiert werden

Wahrheitswerte

Bezeichner	**bool**
Literale	**True, False**
Standardwert	**False**
Operatoren (absteigende Priorität)	**not x** Negation **x and y** Und mit partieller Auswertung **x or y** Oder mit partieller Auswertung Niedrigere Priorität als alle anderen Operatoren: ***not x == y*** entspricht ***not (x == y)*** und ***x == not y*** liefert Syntax-Fehler!

Vergleichsoperatoren

Operatoren	==	gleich
	!=	ungleich
	>	größer
	>=	größer oder gleich
	<	kleiner
	<=	kleiner oder gleich
	is	ist identisch
	is not	ist nicht identisch

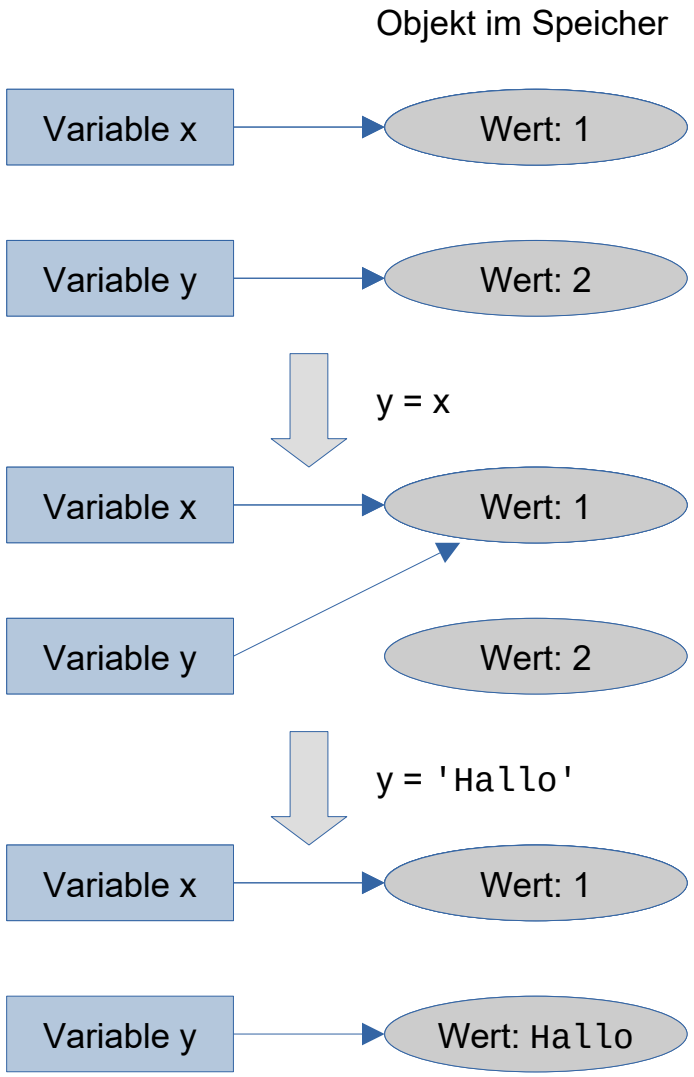
- Haben alle gleiche Priorität
- Können verknüpft werden mit partieller Auswertung: x < y <= z
- Können nur verwendet werden, wo es Sinn macht
  (Datentypen, sprich Klassen, können entsprechende Vergleichs-Methoden anbieten)

Variablen und Werte – Teil 1

```
x = 1
print(id(x))   # 140703929562024
print(id(1))   # 140703929562024
y = 2
print(id(y))   # 140703929562056
print(id(2))   # 140703929562056

y = x
print(id(x))   # 140703929562024
print(id(y))   # 140703929562024

y = 'Hallo'
print(x, y)    # 1 Hallo
print(id(x))   # 140703929562024
print(id(y))   # 2906441908896
```



- Zuweisung einer Variablen: Kopie der Referenz
- Dynamische Typisierung: Typkonvertierung (Cast) von Zahl zu Zeichenkette ist erstmal kein Problem

Variablen und Werte – Teil 2

- Python kopiert keine Werte (Objekte) im Speicher !
- Das nennt sich Referenz-Semantik
- Sofern ein Objekt bei komplexeren Szenarien nicht mehr benötigt wird, wird sich die automatische Speicherbereinigung kümmern (Garbage Collector)

```
x = 1
print(id(x))  # 140703931528104
y = x
print(id(y))  # 140703931528104
y = 2
print(id(y))  # 140703931528136
print(id(x))  # 140703931528104
z = 2
print(id(z))  # 140703931528136

x = [1, 2, 3]  # Liste
print(id(x))  # 2478157701952
y = x
y[1] = 4
print(id(y))  # 2478157701952
print(y)      # [1, 4, 3]
```

Referenz-Semantik:
Das gleiche ist nicht unbedingt dasselbe

- `==` testet auf wertmäßige Gleichheit
- `is` testet, ob identisch

```
x = 1
y = 1.0

print(x == y)  # True
print(x is y)  # False

print(id(x))   # 140703777715112
print(id(y))   # 2703769907600
```

Bedingte Anweisungen

```
if a==3:
   print("a ist 3")
elif a==4:
   print("a ist 4")
elif a==4:
   print("a ist 5")
else:
   print("weder 3 noch 4 noch 5")
```

- elif und else sind optional
- Die Einrückung bestimmt den Block, Klammern sind nicht nötig

Bedingte Zuweisung (Ternäres if)

```
if a > b:
   max=a
else:
   max=b

# kann auch geschrieben werden als
max = a if (a > b) else b

# und direkt in Ausdrücken verwendet
   werden
a = 17
b = 50
result = (21 if a < b else 7) * 2
```

Bedingte Schleifen

while (*Bedingung*) { *Anweisungen* } //abweisend

```
i = 1
summe = 0
while i < 101:
    summe += i
    i += 1
else:
    print("jetzt ist i > 100")
print("Summe der Zahlen von 1 bis 100:", summe)
```

- Bis auf den optionalen ELSE-Zweig übliches Konstrukt in Programmiersprachen
- Selbstverständlich können Schleifen geschachtelt werden

Break und Continue

```
while ... :
  ...
  if … :
    break;
  ...
  while … :
     ...
    if …:
      continue ;
    ...
```

- Mit break kann die jeweilige Schleife verlassen werden, wobei dann kein eventuell vorhandener else-Zweig ausgeführt wird
- Mit continue kann der jeweilige Durchlauf abgebrochen werden, und der nächste Durchlauf gestartet werden
- Es gibt aber keine Sprungmarken für geschachtelte Schleifen

for-Schleife

for <target> in <iterable>:
    Anweusungen
[else:
    Anweisungen]

```
summe = 0
for i in range(101):
    summe += i
else:
    print("jetzt ist i > 100")
print("Summe der Zahlen von 1 bis 100:", summe)

# oder noch einfacher
print("Summe der Zahlen von 1 bis 100:", sum(range(1,101)))
```

- Schleife basiert intern darauf, dass die Wertemenge einen Iterator liefert
- Es ist in dieser Form nur Lesen ohne Schrittkontrolle möglich