Datenstrukturen

Datenstrukturen sind die Formen in denen die Daten gespeichert werden.

Python ist eine **nicht typisierte** Sprache. Das heißt man braucht einer Datenstruktur keinen Datentyp zuordnen. Das geschieht bei der Zuweisung von Daten dynamisch.

```
>>> a = 42
>>> a
42
>>> type(a)
<class 'int'>
>>> a = 3.14
>>> a
3.14
>>> type(a)
<class 'float'>
>>> a = "Hallo"
>>> a
'Hallo'
>>> type(a)
<class 'str'>
>>>
```

Datenstrukturen - Übersicht

Python kennt folgende Datenstrukturen:

- **Konstanten**
- **Variablen**
- **Listen**
- **Dictionaries**
- **Strings**
- **Sets**
- **Tuples**

Datenstrukturen - const

CPython kennt keine Konstanten.

In Micropython gibt es sie.

Konstanten dienen zur Speicherung von Werten die während des Progammablaufs nicht verändert werden.

Konstanten können kein float sein.

```
>>> MWST = const(19)
>>> MWST
19
>>> PI = const(3.14)
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: constant must be an integer
>>> TEXT = 'Hallo'
>>> TEXT
'Hallo'
>>>
```

Datenstrukturen - Variablen

Variablen sind einfache Schachteln in die man einen Wert hinein tun kann.

```
>>> a = 42
>>> a
42
>>> print(a)
42
```

Man kann den Wert wieder herausnehmen und zwar beliebig oft.

Man kann auch einen anderen Wert hinein tun. Der vorherige Wert ist dann verloren.

```
>>> a = 3.14
>>> a
3.14
>>> print(a)
3.14
>>>
```

Datenstrukturen - Variablen

Mehrfachzuweisungen

```
>>> a, b, c = 1, 2, 3
>>> a
1
>>> b
2
>>> c
3

>>> a = b = c = 1
>>> a
1
>>> b
1
>>> c
1
>>>
```

Datenstrukturen - Listen

Listen sind Datenstrukturen, die alle Datentypen aufnehmen können.

Die Reihenfolge der Elemente ist vorgegeben.

Werte dürfen mehrfach vorkommen.

```
>>> l = [42, 'Hallo', 3.14]
>>> l
[42, 'Hallo', 3.14]
>>> type(l)
<class 'list'>
>>> l = [1, 2, 3, 2, 1]
>>> l
[1, 2, 3, 2, 1]
>>>
```

Datenstrukturen - Listen

Listen dürfen auch Listen und andere Datenstrukturen enthalten.

```
>>> l = [[1, 2, 3], ['A', 'B', 'C']]
>>> l
[[1, 2, 3], ['A', 'B', 'C']]
>>>
```

Datenstrukturen - Listen

Auf Listen kann mit einem Index zugegriffen werden.

Der Index beginnt bei 0.

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[0]
1
>>> l[5]
6
>>>
```

Negative Indices beginnen hinten.

```
>>> l[-1]
6
>>> l[-6]
1
>>>
```

Datenstrukturen - Listen

Durch **Slicing** lassen sich Ausschnitte aus Listen erstellen.

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[0:3]
[1, 2, 3]
>>>
```

Es wird immer vom Startindex bis zum Endindex – 1 ausgegeben.

Das geht auch Rückwärts.

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l[-5:-1]
[2, 3, 4, 5]
```

Aber:

```
>>> l[-5:0]
[]
```

Datenstrukturen - Listen

list() ist eine Klasse und hat Methoden:

- l.append()
- l.clear()
- l.copy()
- l.count()
- l.extend()
- l.index()
- l.insert()
- l.pop()
- l.remove()
- l.reverse()
- l.sort()

```
>>> l = []
>>> type(l)
<class 'list'>
>>> dir(l)
['__class__', 'append', 'clear',
'copy', 'count', 'extend', 'index',
'insert', 'pop', 'remove', 'reverse',
'sort']
>>>
```

Datenstrukturen - Listen

l.copy() kopiert eine Sequenz. Es entstehen unterschiedliche Objekte.

= Hier entsteht nur ein neuer Verweis auf das selbe Objekt.

```
>>> l = [1,2,3]
>>> l1 = l.copy()
>>> l2 = l
>>> l1
[1, 2, 3]
>>> l2
[1, 2, 3]

>>> id(l)
1073493408

>>> id(l1)
1073495632

>>> id(l2)
1073493408
```

Datenstrukturen - Listen

l.append() fügt einen Wert oder eine Sequenz an eine vorhandene Liste an.

l.remove() entfernt einen Wert oder eine Sequenz.

l.clear() löscht den Inhalt einer Sequenz.

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l
[1, 2, 3, 4, 5, 6]

>>> l1 = l.copy()
>>> l1
[1, 2, 3, 4, 5, 6]

>>> l1.append(l)
>>> l1
[1, 2, 3, 4, 5, 6, [1, 2, 3, 4, 5, 6]]

>>> l1.remove(l)
>>> l1
[1, 2, 3, 4, 5, 6]

>>> l1.clear()
>>> l1
[]
```

Datenstrukturen - Listen

l.count() zählt wie oft ein Zeichen vorhanden ist.

l.extend() fügt die Elemente einer anderen Liste zur Liste hinzu.

```
>>> l = [1,2,3,4,5,2,3,1,1,7,8,9]
>>> l
[1, 2, 3, 4, 5, 2, 3, 1, 1, 7, 8, 9]
>>> l.count(1)
3

>>> l = [1, 2, 3, 4, 5, 6]
l1 = l.copy()
>>> l1
[1, 2, 3, 4, 5, 6]
>>> l.extend(l1)
>>> l
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
>>>
```

Datenstrukturen - Listen

l.index() gibt den Index des Zeichens zurück.

```
>>> l = [1,2,3,4,5,6]
>>> l
[1, 2, 3, 4, 5, 6]
>>> l.index(1)
0
>>> l.index(4)
3
```

l.insert(Index, Objekt**)**
Index beginnt bei 0!

```
>>> l.insert(3, 12)
>>> l
[1, 2, 3, 12, 4, 5, 6]
>>>
```

Datenstrukturen - Listen

l.pop() gibt das letzte Zeichen der Liste zurück und entfernt es (Stackprinzip).

```
>>> l
[1, 2, 3, 12, 4, 5, 6]
>>> l.pop()
6
>>> l
[1, 2, 3, 12, 4, 5]
>>>
```

l.reverse() kehrt den Inhalt der Liste um.

```
>>> l
[1, 2, 3, 12, 4, 5]
>>> l.reverse()
>>> l
[5, 4, 12, 3, 2, 1]
>>>
```

Datenstrukturen - Listen

* wiederholt eine Sequenz

l.sort() sortiert eine Sequenz

l.sort() gibt nichts zurück.

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> l
[1, 2, 3, 4, 5, 6]
>>> l1 = l * 3
>>> l1
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6,
1, 2, 3, 4, 5, 6]

>>> l1.sort()
>>> l1
[1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4,
5, 5, 5, 6, 6, 6]

>>> l2 = l * 3
>>> l3 = l2.sort()
>>> l3
```

Datenstrukturen - Listen

Funktionen für Listen:

len(iterable**)** liefert die Anzahl der Elemente in einer Liste.

```
>>> l = [1, 2, 3, 4, 5]
>>> l
[1, 2, 3, 4, 5]
>>> len(l)
5
```

max(iterable**)**

min(iterable**)**

```
>>> max(l)
5
>>> min(l)
1
>>>
```

Datenstrukturen - Dictionaries

Dictionaries (Wörterbücher) enthalten Einträge, die aus einem **Key:Value** (Schlüssel:Werte) Paar (Item) bestehen.

Der Zugriff auf den Wert erfolgt über den Schlüssel.

Die Reihenfolge der Einträge ist nicht festgelegt.

```
>>> d = {'eins':1, 'zwei':2, 'drei':3}
>>> d
{'eins': 1, 'zwei': 2, 'drei': 3}


>>> d['zwei']
2
>>>

>>> d['vier'] = 4
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3}
>>>
```

Datenstrukturen - Dictionaries

dict() ist eine Klasse und hat Methoden:

- **d.clear(),**
- **d.copy(),**
- **d.get(),**
- **d.items(),**
- **d.keys(),**
- **d.pop(),**
- **d.popitem(),**
- **d.setdefault(),**
- **d.update(),**
- **d.values()**
- **d.fromkeys()**

```
>>> d = {}
>>> d
{}
>>> type(d)
<class 'dict'>
>>>

>>> dir(dict)
['__class__', '__delitem__',
'__getitem__', '__name__',
'__setitem__', 'clear', 'copy', 'get',
'items', 'keys', 'pop', 'popitem',
'setdefault', 'update', 'values',
'__bases__', '__dict__', 'fromkeys']
>>>
```

Datenstrukturen - Dictionaries

d.clear()
d.copy()

```
# siehe dort
```

Verhalten sich wie bei den Listen.

Datenstrukturen - Dictionaries

d.get(key**)** gibt des Wert des Schlüssels key zurück.

Hier gibt es noch etwas zum Thema Unicode (UTF8) zu sehen.

```
>>> d = {'eins':1, 'zwei':2, 'drei':3,
'vier':4, 'fünf':5, 'sechs':6}
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'f\xfcnf': 5, 'sechs': 6}
>>> print(d)
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'f\xfcnf': 5, 'sechs': 6}
>>> print('fünf')
fünf
>>> print('f\xfcnf')
fünf
>>> print(d.keys())
dict_keys(['eins', 'zwei', 'vier',
'drei', 'f\xfcnf', 'sechs'])

>>> d.get('fünf')
5
```

Datenstrukturen - Dictionaries

d[key] vs. **d.get(key)**

Der Zugriff auf ein Element mit dem Index oder der Methode get liefert im Fehlerfall unterschiedliche Ergebnisse:

get(key) liefert None zurück.

d[key] liefert eine Fehlermeldung und bricht die Programmausführung ab.

```
>>> d = {'eins':1, 'zwei':2, 'drei':3,
'vier':4, 'fünf':5, 'sechs':6}
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'f\xfcnf': 5, 'sechs': 6}
>>> d.get('drei')
3
>>> d.get('sieben')
>>> a = d.get('sieben')
>>> a
>>> a is None
True
>>> a = d['sieben']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: sieben
>>>
```

Datenstrukturen - Dictionaries

d.items()

```
>>> d.items()
dict_items([('eins', 1), ('zwei', 2),
('vier', 4), ('drei', 3), ('f\xfcnf',
5), ('sechs', 6)])
```

d.keys()

```
>>> d.keys()
dict_keys(['eins', 'zwei', 'vier',
'drei', 'f\xfcnf', 'sechs'])
```

d.values()

```
>>> d.values()
dict_values([1, 2, 4, 3, 5, 6])
>>>
```

Datenstrukturen - Dictionaries

d.pop(key, default**)** gibt den Wert von key zurück und entfernt das Item aus dem Dictionary. Gibt es den Key nicht wird default zurückgegeben.

d.popitem() gibt ein beliebiges Item zurück und entfernt es.

```
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'f\xfcnf': 5, 'sechs': 6}
>>> d.pop('fünf')
5
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'sechs': 6}

>>> d.popitem()
('eins', 1)
>>> d
{'zwei': 2, 'vier': 4, 'drei': 3,
'sechs': 6}
>>>
```

Datenstrukturen - Dictionaries

d.setdefault(key, default**)**
Wird hier nicht behandelt.

d.update(d2**)** fügt dem Directory **d** das Directory **d2** hinzu. Vorhandene Schlüssel werden überschrieben.

```
>>> d
{'eins': 1, 'zwei': 2, 'vier': 4,
'drei': 3, 'f\xfcnf': 5, 'sechs': 6}
>>> d2 = {'sieben':7, 'acht':8,
'zwei':'two'}
>>> d2
{'zwei': 'two', 'sieben': 7, 'acht':
8}
>>> d.update(d2)
>>> d
{'f\xfcnf': 5, 'sieben': 7, 'acht': 8,
'sechs': 6, 'eins': 1, 'zwei': 'two',
'vier': 4, 'drei': 3}
>>>
```

Datenstrukturen - Tupel

Tupel sind den Listen sehr ähnlich. Sie können aber nicht verändert werden (immutable).

Tuple die nur eine Zahl enthalten erfordern ein Komma hinter der Zahl, sonst wird nur eine Zahl erkannt.

```
>>> t = (1, 2, 3, 4, 5, 6)
>>> t
(1, 2, 3, 4, 5, 6)
>>> type(t)
<class 'tuple'>

>>> t1 = (1)
>>> type(t1)
<class 'int'>

>>> t2 = (1,)
>>> type(t2)
<class 'tuple'>
>>>
```

Datenstrukturen - Tupel

Der Zugriff auf Tupel erfolgt wie bei Listen über den Index.

Tupel sind immutable, aber dieser Trick funktioniert:

```
>>> t = (1, 2, 3, 4, 5, 6)
>>> t[2]
3
>>>

>>> l = [1, 2, 3]
>>> t4 = (l,)
>>> t4
([1, 2, 3],)
>>> type(t4)
<class 'tuple'>

>>> l[0] = 5
>>> l
[5, 2, 3]
>>> t4
([5, 2, 3],)
>>>
```

Datenstrukturen - Sets

Sets dienen dem Umgang mit Mengen. Auch sie sind den Listen ähnlich, allerdings dürfen Werte in einem Set nur einmal vorkommen.

```
>>> s = {1, 2, 3, 4, 5, 6}
>>> type(s)
<class 'set'>
```

Die Einträge sind nicht sortiert.

```
>>> s
{6, 1, 2, 3, 4, 5}
>>>
```

Datenstrukturen - Sets

Auch Sets haben Methoden.

Einige sind von den Listen her bekannt.

Andere sind neu.

Diese wollen wir uns genauer ansehen:

```
>>> dir(set)
['__class__', '__name__', 'clear',
'copy', 'pop', 'remove', 'update',
'__bases__', '__contains__',
'__dict__', 'add', 'difference',
'difference_update', 'discard',
'intersection', 'intersection_update',
'isdisjoint', 'issubset',
'issuperset', 'symmetric_difference',
'symmetric_difference_update',
'union']
>>>
```

Datenstrukturen - Sets

s.add(Wert**)** fügt einen Wert zum Set hinzu.

```
>>> s1 = {1, 2, 3, 4, 5}
>>> s1
{5, 1, 2, 3, 4}

>>> s1.add(6)
>>> s1
{6, 1, 2, 3, 4, 5}
>>>

>>> s2 = {4, 5, 6, 7, 8, 9}
>>> s2
{6, 7, 8, 9, 4, 5}
>>>
```

Datenstrukturen - Sets

s.difference() liefert den Unterschied von 2 oder mehr Mengen zurück.

s.difference_update() entfernt alle Werte des anderen Sets aus s.

Es entspricht x = x-y.

```
>>> s1.difference(s2)
{1, 2, 3}
>>> s2.difference(s1)
{7, 8, 9}
>>>
```

Datenstrukturen - Sets

s.discard(element**)** entfernt element aus dem Set s.

Wenn es nicht existiert passiert nichts.

s.remove(element**)** wie discard() aber wenn das Element nicht existiert gibt es eine Fehlermeldung.

```
>>> s = {1, 2, 3}
>>> s
{3, 1, 2}

>>> s.discard(3)
>>> s
{1, 2}
>>>

>>> s.remove(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>>
```

Datenstrukturen - Sets

s.intersection(sx**)** oder **&** liefert die Schnittmenge von s und sx zurück.

```
>>> s1.intersection(s2)
{4, 5, 6}

>>> s1 & s2
{4, 5, 6}
>>>
```

s.isdisjoint(sx**)** gibt True zurück wenn die Schnittmenge leer ist.

```
>>> s = {1,2}
>>> sx = {3,4}
>>> s.isdisjoint(sx)
True
>>>
```

Datenstrukturen - Sets

s.issubset(sx**)** gibt True zurück, wenn s eine Untermenge von sx ist.

```
>>> s = {1, 2, 3, 4, 5}
>>> sx = {1, 5}

>>> s.issubset(sx)
False
>>> sx.issubset(s)
True
```

s.issuperset(sx**)** gibt True zurück, wenn s eine Obermenge von sx ist.

```
>>> s.issuperset(sx)
True
>>> sx.issuperset(s)
False
```

Datenstrukturen - Strings

Strings sind Zeichenketten, also eine Folge von Zeichen. Micropython codiert Zeichen als UTF8.

Strings werden in Python durch einfache oder doppelte Anführungsstriche begrenzt.

Sie können keinen Zeilenwechsel enthalten

```
>>> 'Ich bin auch ein String'

>>> "Ich bin ein String"

>>> 'Deshalb kann man auch "Dieses" machen!'
'Deshalb kann man auch "Dieses" machen!'
>>>
```

Datenstrukturen - Strings

Für lange Strings werden 3fache Anführungszeichen verwendet.

Dann ist auch ein Zeilenwechsel möglich.

Es können einfache (‘) oder doppelte (“) Anführungszeichen verwendet werden.

Dreifache Anführungszeichen werden auch für Docstrings verwendet.

```
>>> '''
Hallo,
Hallo
'''
'\nHallo,\nHallo\n'
>>>

>>> print("""
Hallo,
Hallo
""")

Hallo,
Hallo

>>>
```

Datenstrukturen - Strings

Auf einzelne Elemente eines Strings kann durch den Index zugegriffen werden.

Teilstrings können durch Sliceing erzeugt werden.

```
>>> s = 'Hallo Micropython'
>>> s
'Hallo Micropython'
>>> s[0]
'H'
>>> s[6]
'M'
>>> s[0:6]
'Hallo '
>>> s[6:]
'Micropython'
>>> s[6::2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NotImplementedError: only slices with
step=1 (aka None) are supported
>>>
```

Datenstrukturen - Strings

Die Klasse **str()** enthält eine ganze Reihe Methoden:

```
>>> dir(str)
['__class__', '__name__', 'count',
'endswith', 'find', 'format', 'index',
'isalpha', 'isdigit', 'islower',
'isspace', 'isupper', 'join', 'lower',
'lstrip', 'replace', 'rfind',
'rindex', 'rsplit', 'rstrip', 'split',
'startswith', 'strip', 'upper',
'__bases__', '__dict__', 'center',
'encode', 'partition', 'rpartition',
'splitlines']
>>>
```

Datenstrukturen - Strings

s.count(x**)** Anzahl des Zeichens x im String.
s.find(x**)** gibt den kleinsten Index des Zeichens in der Klammer zurück. Wenn nicht vorhanden gibt es -1 zurück.
s.index(x**)** wie find() aber Fehlermeldung
s.replace(x,y,anzahl**)** ersetzt x durch y.

```
>>> s = 'Hallo Micropython'
>>> s.count('n')
1
>>> s.count('o')
3

>>> s.find('lo')
3

>>> s.index('t')
13

>>> s.replace('o','a')
'Halla Micrapythan'
>>>
```

Datenstrukturen - Strings

Strings sind unveränderlich. Deshalb kann man auch keine Änderungen mit der Indizierung vornehmen.

```
>>> s.replace('a','o')
'Hollo Micropython'

>>> s[1] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't
support item assignment
>>>
```

Aber so geht es:

```
>>> s.replace('a','o')
'Hollo Micropython'

>>> s.replace('o', 'a', 0)
'Hallo Micropython'
>>>
```

Datenstrukturen - Strings

s.upper()

s.lower()

s.isalpha()
s.isdigit()
s.islower()
s.isspace()
s.isupper()

```
>>> s
'Hallo Micropython'

>>> su = s.upper()
>>> su
'HALLO MICROPYTHON'

>>> sl = s.lower()
>>> sl
'hallo micropython'
>>>

>>> sl.islower()
True
>>>
```

Datenstrukturen - Strings

s.split(sep) teilt den String bei sep.

s.rsplit(sep) beginnt am Ende

s.strip(chars) entfernt Zeichen am Anfang und Ende des Strings.

s.lstrip
s.rstrip
s.splitlines teilt bei Zeilenvorschub

```
>>> s = 'eins zwei drei'
>>> s
'eins zwei drei'

>>> ss = s.split(' ')
>>> ss
['eins', 'zwei', 'drei']

>>> sstrip = '   Hallo    '
>>> sstrip
'   Hallo    '

>>> sss  = sstrip.strip()
>>> sss
'Hallo'
>>>
```

Datenstrukturen - Strings

s.join(Liste) erstellt einen String aus den Werten der Liste, mit dem Trennzeichen in s.

```
>>> liste = ['Hallo', 'M5Stack',
'Micropython']
>>> liste
['Hallo', 'M5Stack', 'Micropython']

>>> ' '.join(liste)
'Hallo M5Stack Micropython'
>>>
```

Datenstrukturen - Strings

s.format(wert1, wert2**)** fügt die Werte in die Platzhalter „{}“ ein.

```
>>> 'Micropython ist {} und
{}'.format('klar', 'lesbar')

'Micropython ist klar und lesbar'
>>>
```

s.center(Spaltenbreite**)**

```
>>> s = 'Micropython'
>>> len(s)
11
>>> s.center(19)
'    Micropython    '
>>>
```

Datenstrukturen - Strings

s.startswith(str**)** prüft ob s mit str beginnt.

s.endswith(str**)** prüft ob s mit str endet.

```
>>> s = 'Hallo Micropython'
>>> s
'Hallo Micropython'
>>> s.startswith('H')
True
>>> s.startswith('x')
False
>>> s.startswith('Ha')
True
>>> s.endswith('n')
True
>>> s.endswith('x')
False
>>> s.endswith('thon')
True
>>>
```

Datenstrukturen - Strings

s.partition(sep**)** trennt s bei sep auf.

```
>>> s.partition(' ')
('Hallo', ' ', 'Micropython')

>>> s.partition('M')
('Hallo ', 'M', 'icropython')
>>>
```

s.rpartition(sep) wie partition() beginnt aber am Ende.

```
# probiert selbst aus wie das
funktioniert.
```

Datenstrukturen - Strings

s.encode() codiert den Python Unicode String in einen anderes Kodierverfahren.

Wird hier nicht weiter behandelt.

Datenstrukturen - Strings

Escape Sequenzen für Steuerzeichen:

\n → neue Zeile

\t → Tabulator

\\ → \

```
>>> 'Hallo\nMicropython'
'Hallo\nMicropython'

>>> print('Hallo\nMicropython')
Hallo
Micropython
>>>
```