

KAPITEL 2: EINFÜHRUNG IN PYTHON

Eine ganz kurze Geschichte von Python



Quelle: Wikipedia

Guido van Rossum, Entwickler von Python:

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office (a government-run research lab in Amsterdam) would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus).

Python entstand Anfang der 90er Jahre am Zentrum für Mathematik in Amsterdam als Nachfolger der Lehrsprache ABC:

- 1991: Python v. 0.9
- 1994: Python v. 1.0
- 2000: Python v. 2.0
- 2008: Python v. 3.0

The Zen of Python

```
>>> import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than **right** now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!



Daniel Greenfeld
pydanny.com / @pydanny

Eigenschaften von Python

- einfach und minimalistisch
- Open Source
- Hochsprache
- portierbar
- interpretiert
- objektorientiert
- erweiterbar und einbettbar
- umfangreiche Bibliotheken

Vor- und Nachteile von Python

Vorteile:

- schnell erlernbar
- gut lesbarer Code
- weite Verbreitung
- aktive „Community“ im Web (Tutorials, Diskussionsforen etc.)
- umfangreiche Standardbibliothek
- viele nützliche Pakete, insbesondere für Data Analytics
- geeignet sowohl für Forschung/Prototyping als auch für Produktivsysteme

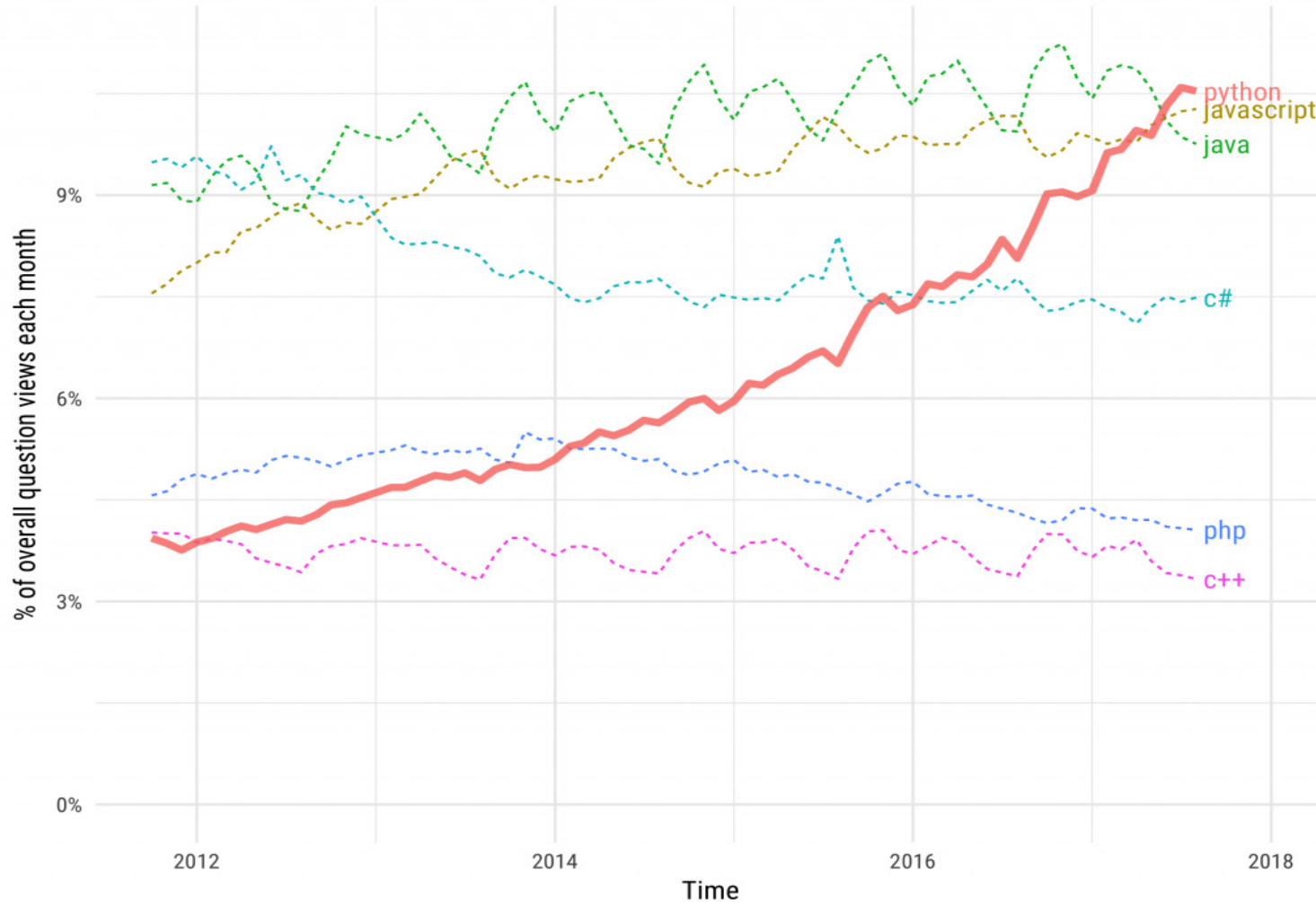
Nachteile:

- Geringere Ausführungsgeschwindigkeit als bei Compiler-Sprachen
- keine gute Wahl für Multi-Threading (GIL)
- Dynamische Typisierung führt ggf. zu späterer Erkennung von Fehlern

Verbreitung von Python

Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries



Quelle:
stackoverflow

Warum Python für Data Analytics?

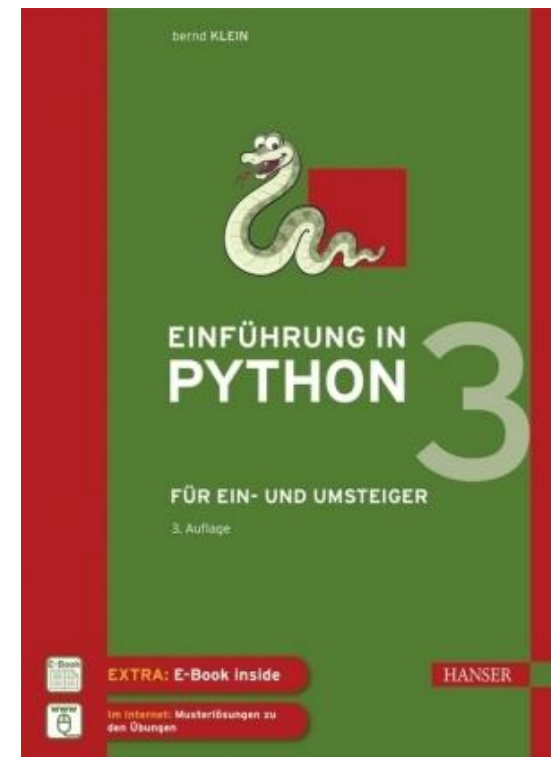
- Jupyter-Notebook als interaktive Entwicklungsumgebung
- Nützliche Pakete und Datenstrukturen für z.B.
 - Datenmanipulation
 - Datenanalyse
 - Visualisierung
 - Machine Learning
 - Deep Learning
 - Statistik
 - Natural Language Processing
 - Web-Applications
 - u.v.m.

- Open-Source-Distribution für die Sprachen Python und R
- Kommandozeileninterpreter IPython
- Jupyter-Notebooks
- Entwicklungsumgebung Spyder
- einfache Installation und Paketverwaltung
- Download unter <https://www.anaconda.com/distribution/> für Windows, macOS und Linux
- Anaconda ist in den EDV-Laboren 124 und 224 installiert



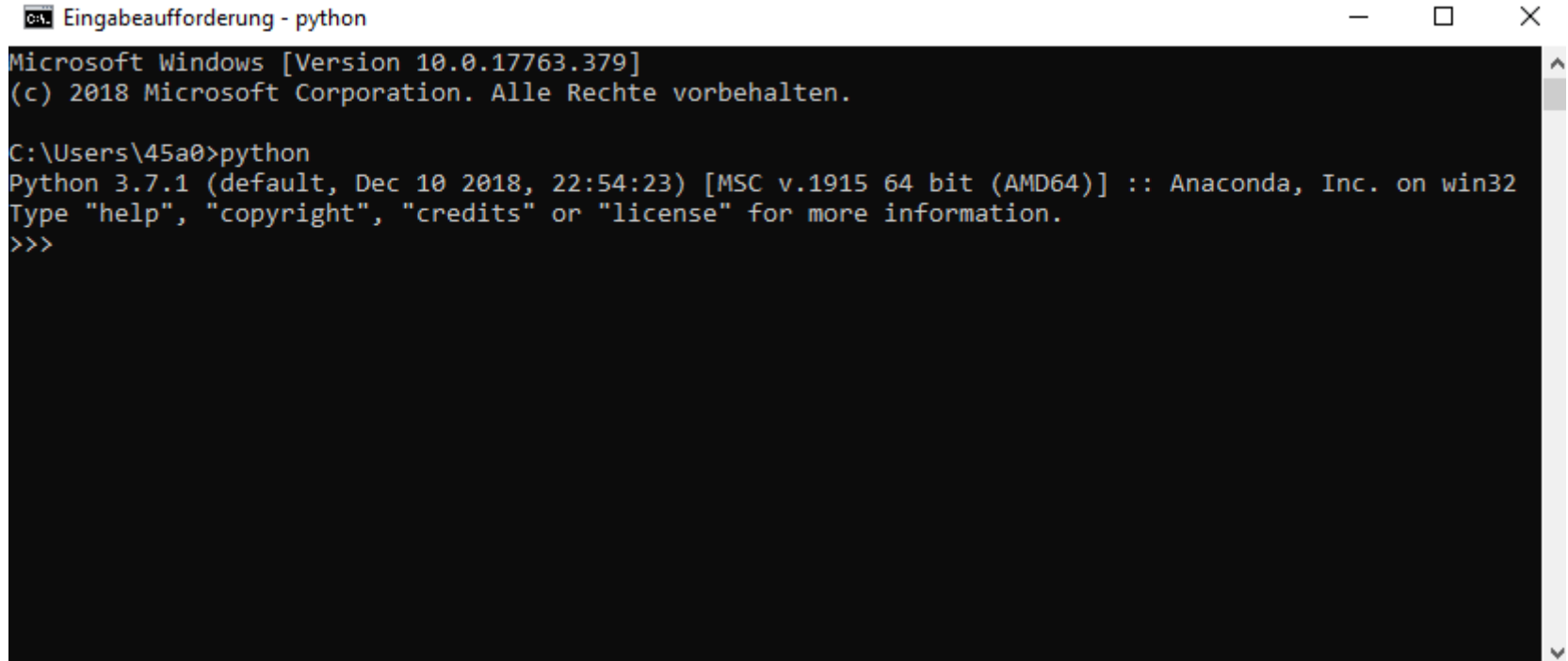
Hinweis

Die nachfolgende Einführung in Python ist angelehnt an das Buch „Einführung in Python 3“ von B. Klein.



Erste Schritte in der interaktiven Shell

Durch Eingabe von „python“ in der shell startet man den Python-Interpreter:



```
C:\Users\45a0>python
Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\45a0>python
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Demonstration in der Shell

- Python als Taschenrechner verwenden
- Strings ausgeben
- `print()`
- Variablen anlegen und verwenden
- mehrzeilige Anweisungen
- Die Variable `_` („Unterstrich“)
- Befehls-Stack
- Kommentare
- Anzeigen aller Variablen mit `dir()`
- Verlassen der Python-Shell

Programme schreiben

Python

```
hallo_welt.py  
print("Hallo Welt!")
```

C++

```
hallo_welt.cpp  
  
#include <iostream>  
using namespace std;  
  
int main(){  
    cout << "Hallo Welt!" << endl;  
    return 0;  
}
```

Datentypen und Variablen



Definition von Variablen

- Variablen sind Verweise auf Objekte
- Objekte haben einen Datentyp
- Definition bzw. Zuweisung erfolgt mit „=“

```
>>> x = 123  
>>> type(x)  
<class 'int'>
```

Variablennamen:

- gültige Variablennamen müssen mit einem Buchstaben oder Unterstrich beginnen
- Konventionen für Variablennamen:
 - Kleinschreibung
 - Unterstrich als Separator
 - CamelCase unüblich

Integer

- können beliebig groß bzw. beliebig klein werden
- Dezimalzahlen dürfen nicht mit einer 0 beginnen

```
>>> x = 1232352390847230947130132947129
>>> x*x
1518692415226886265873366234599355705700941786254479109342641
```

- Literale für Binärzahlen beginnen mit 0b

```
>>> x = 0b10100
20
>>> bin(12)
'0b1100'
```

- Literale für Hexadezimalzahlen beginnen mit 0x oder 0X

```
>>> x = 0x10A
>>> x
266
```

Fließkommazahlen

- Datentyp float
- Speicherung als Binärbruch führt zu Approximationsfehlern

```
>>> x = 0.123
>>> type(x)
<class 'float'>
>>> 0.1+0.2
0.30000000000000004
```

- Für exakte dezimale Darstellung kann bei Bedarf das Modul decimal verwendet werden.

```
>>> from decimal import Decimal
>>> Decimal('0.1')+Decimal('0.2')
Decimal('0.3')
```


Boolsche Werte

- Datentyp mit den zwei Werten „True“ und „False“
- Entspricht den Werten „1“ und „0“, deshalb kann man damit „numerisch“ rechnen:

```
>>> x = True
>>> type(x)
<class 'bool'>
>>> x*4
4
>>> not(x)
False
>>> not x
False
>>> y = False
>>> x and y
False
>>> x or y
True
>>> x and not y
True
```

Komplexe Zahlen

- Datentyp „complex“
- imaginäre Einheit: j
- für uns nicht relevant

```
>>> y = 2+3j
>>> type(y)
<class 'complex'>
>>> y*y
(-5+12j)
```

Statische Typdeklaration

- In vielen Sprachen (C, C++, Java) muss man einer Variablen bei der Deklaration einen Typ zuordnen.
- Der Typ ist zur Laufzeit nicht mehr veränderbar, lediglich der Wert.

Dynamische Typdeklaration

- In Python wird der Datentyp automatisch von Python erkannt und zugeordnet.
- Es kann sowohl der Wert als auch der Typ einer Variablen während der Laufzeit verändert werden.
- Dies führt dazu, dass die Variable auf ein anderes Objekt zeigt.

Datentyp ermitteln

- Der aktuelle Typ kann mit `type()` ausgegeben werden (s. oben).
- Um auf einen bestimmten Typ zu prüfen, kann die Funktion `isinstance()` verwendet werden.

Typen konvertieren:

- `int()`
- `float()`
- `str()`
- `hex()`
- `oct()`

Arithmetische und bitweise Operatoren

Operation	Bedeutung
$x + y$	Summe
$x - y$	Differenz
$x * y$	Produkt
x / y	Quotient
$x // y$	Ganzzahlige Division
$x \% y$	Modulo-Division
$\text{abs}(x)$	Absolutbetrag
$x ** y$	Potenz

Operator	Bedeutung
$\&$	bitweises AND
$ $	bitweises OR
\sim	bitweises NOT
\wedge	bitweises XOR
\gg	bitweiser right shift
\ll	bitweises left shift

Zuweisungsoperatoren

Operator	Bedeutung	Äquivalent
=	$x = 5$	$x = 5$
+=	$x += 5$	$x = x + 5$
-=	$x -= 5$	$x = x - 5$
*=	$x *= 5$	$x = x * 5$
/=	$x /= 5$	$x = x / 5$
%=	$x \% = 5$	$x = x \% 5$
//=	$x //= 5$	$x = x // 5$
**=	$x ** = 5$	$x = x ** 5$
&=	$x \& = 5$	$x = x \& 5$
=	$x = 5$	$x = x 5$
^=	$x \wedge = 5$	$x = x \wedge 5$
>>=	$x >> = 5$	$x = x >> 5$
<<=	$x << = 5$	$x = x << 5$

Identitätsoperatoren

Operator	Bedeutung	Äquivalent
is	True falls die Operanden auf dasselbe Objekt referenzieren	x is True
is not	True falls die Operanden nicht auf dasselbe Objekt referenzieren	x is not True

```
>>> x = [1,2,3]
>>> y = [1,2,3]
>>> x == y
True
>>> x is y
False
>>> id(x)
2288073133192
>>> id(y)
2288068515976
```

Sequentielle Datentypen



Strings

- Für Zeichenketten gibt es den Datentyp String:

```
>>> s = "Data Analytics"
>>> t = 'ist spannend'
>>> s
'Data Analytics'
>>> t
'ist spannend'
>>> type(s)
<class 'str'>
```

- Definition von Strings über mehrere Zeilen:

```
>>> s = "Dieser String ist \
... ueber zwei Zeilen definiert"
>>> print(s)
Dieser String ist ueber zwei Zeilen definiert
```

- Zeilenumbrüche in Strings:

```
>>> s = """Dieser String besteht aus
... zwei Zeilen"""
>>> print(s)
Dieser String besteht aus
zwei Zeilen
```


Strings: Escape-Zeichen

Sequenz	Beschreibung
\\	Backslash
\'	Hochkomma
\“	Anführungszeichen
\b	Backspace
\NNAME	Unicode-Zeichen NAME
\t	Horizontaler Tabulator
\n	Zeilenumbruch
\uXXXX	16 bit Unicode Zeichen
\uXXXXXXXXXX	32 bit Unicode Zeichen
\v	Vertikaler Tabulator
\ooo	ASCII-Zeichen oktoal
\xhh	ASCII-Zeichen hexadezimal

Listen und Tupel

Listen

- Listen sind sequenzielle Anordnungen beliebiger Datentypen.
- Sie werden durch eckige Klammern generiert:

```
>>> x = [1,2,3,"hallo"]
>>> print(x)
[1, 2, 3, 'hallo']
>>> type(x)
<class 'list'>
```

Tupel

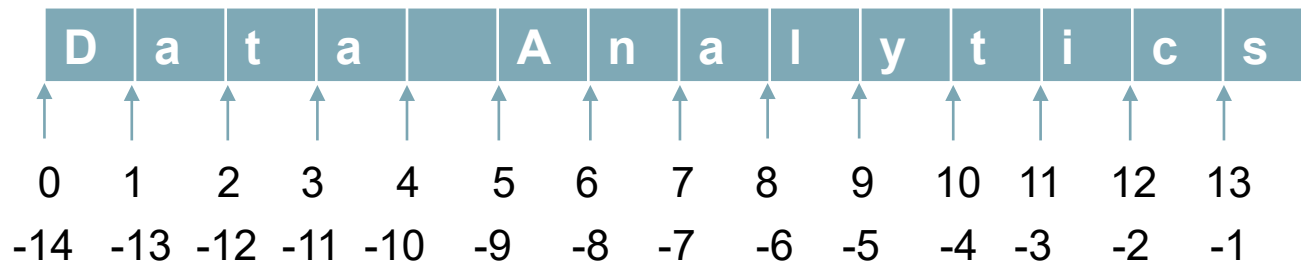
- Tupel unterscheiden sich von Listen nur durch die Art der Klammerung.
- Sie sind jedoch nicht mehr veränderbar (immutable).

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

Indizierung

- Sequenzielle Typen (String, Listen, Tupel) können mit eckigen Klammern indiziert werden.
- Für die Indizierung vom Ende werden negative Indizes unterstützt:

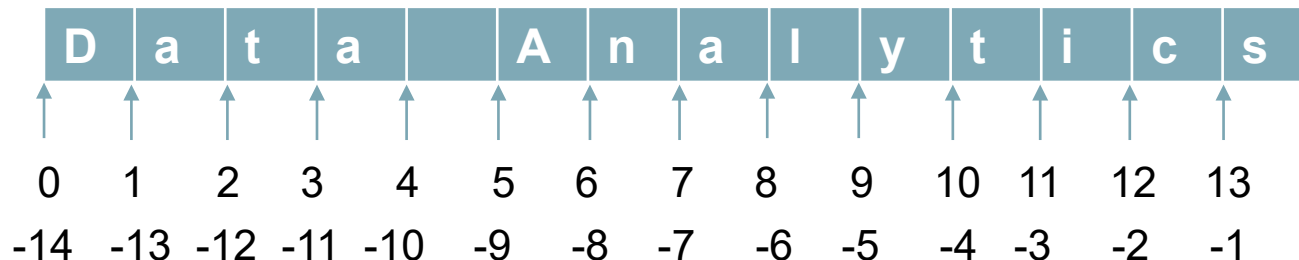
```
>>> text = "Data Analytics"  
>>> text[3]  
'a'  
>>> text[-1]  
's'  
  
>>> l = [1,2,3,[5,6]]  
>>> l[0]  
1  
>>> l[3]  
[5, 6]  
>>> l[3][0]  
5
```



Teilbereiche auswählen („slicing“)

- Man kann Teilbereiche eines sequentiellen Datentyps auswählen.
- Im Falle eines Strings erhält man wieder einen String. Im Falle einer Liste wieder eine Liste.
- Lässt man den Anfangswert weg, beginnt das Slicing am Anfang. Lässt man den Endwert weg, wird alles bis zum Ende übernommen:

```
>>> text = "Data Analytics"  
>>> text[0:4]  
'Data'  
>>> text[-4:-2]  
'ti'  
>>> text[:4]  
'Data'  
>>> text[5:]  
'Analytics'
```



Teilbereiche auswählen („slicing“)

- Der Slicing-Operator funktioniert auch mit drei Argumenten. Das dritte Argument (Inkrement) gibt dann an, das wievielte Argument jeweils genommen werden soll:

```
>>> s = [1,2,3,4,5,6,7,8,9]
>>> s[0:8]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> s[0:8:2]
[1, 3, 5, 7]
>>> s[1:8:2]
[2, 4, 6, 8]
```

- Es ist auch möglich, einen negativen Wert für das Inkrement verwenden:

```
>>> s[5:0:-1]
[6, 5, 4, 3, 2]
>>> s[5:0:-2]
[6, 4, 2]
>>> s[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Die len - Funktion

- Die Anzahl der Elemente eines sequentiellen Datentyps kann man mit der **len**-Funktion bestimmen:

```
>>> text = "Data Analytics"  
>>> len(text)  
14
```

Membership Operatoren

Prüfung, ob ein Element/ein Wert in einer list, str, tuple, set enthalten ist:

Operator	Bedeutung
in	True falls ein Wert/eine Variable in einer Sequenz enthalten ist
not in	True falls ein Wert/eine Variable nicht in einer Sequenz enthalten ist

```
>>> name = "Franz"
>>> "a" in name
True
>>> l = [1,2,3]
>>> 2 in l
True
>>> A = {"A", "B", "C"}
>>> "C" in A
True
```

Mehr zu Strings

- **s.split()** spaltet den String s auf:

```
>>> s = "Ein einfacher junger Mensch reiste im Hochsommer von Hamburg,\n... seiner Vaterstadt, nach Davos-Platz im Graubündischen.\n... Er fuhr auf Besuch für drei Wochen."
>>> print(s)
Ein einfacher junger Mensch reiste im Hochsommer von Hamburg,
seiner Vaterstadt, nach Davos-Platz im Graubündischen.
Er fuhr auf Besuch für drei Wochen.
>>> s.split()
['Ein', 'einfacher', 'junger', 'Mensch', 'reiste', 'im', 'Hochsommer', 'von',
'Hamburg,', 'seiner', 'Vaterstadt,', 'nach', 'Davos-Platz', 'im',
'Graubündischen.', 'Er', 'fuhr', 'auf', 'Besuch', 'für', 'drei', 'Wochen.']
```

- Es werden standardmäßig Teilstrings, die nur aus Whitespaces, Tabs und Newlines bestehen, zu einer Trennstelle zusammengefasst.
- **s.splitlines()** zerlegt einen Text mit Zeilenbegrenzern (z.B. `\\n`) in eine Liste von Zeilen.

Mehr zu Strings

- Suchen von Teilstrings:
 - Operator „in“
 - Methode find()
- **s.count(substring)**: zählt, wie oft der Teilstring substring in s vorkommt.
- **s.replace(str1, str2)**: gibt einen String aus, in dem alle Vorkommen von str1 in s durch str2 ersetzt wurden.
- **s.upper()**: gibt s in Großbuchstaben aus
- **s.strip()**, **s.lstrip()**, **s.rstrip()**: Entfernen von unerwünschten Zeichen am Anfang und/oder Ende von s.
- **s.join(it)** liefert als Ergebnis die Konkatination der Elemente des iterierbaren Objekts it und fügt jeweils den String „s“ dazwischen ein.

Beispiele

```
>>> print(s)
Ein einfacher junger Mensch reiste im Hochsommer von Hamburg,
seiner Vaterstadt, nach Davos-Platz im Graubündischen.
Er fuhr auf Besuch für drei Wochen.
>>> "Hamburg" in s
True
>>> s.find("Hamburg")
53
>>> s.count("im")
2
>>> s.replace("Hamburg", "München")
'Ein einfacher junger Mensch reiste im Hochsommer von München,
seiner Vaterstadt,nach Davos-Platz im Graubündischen.
Er fuhr auf Besuch für drei Wochen.'
>>> s.upper()
'EIN EINFACHER JUNGER MENSCH REISTE IM HOCHSOMMER VON HAMBURG,
SEINER VATERSTADT,NACH DAVOS-PLATZ IM GRAUBÜNDISCHEN.
ER FUHR AUF BESUCH FÜR DREI WOCHEN.'
>>> "-".join("abc")
'a-b-c'
>>> "    Hallo \n".strip()
'Hallo'
>>> "    Hallo \n".lstrip()
'Hallo \n'
```

Listen manipulieren

- `s.append(x)` hängt `x` an das Ende der Liste `s` an
- `s.pop(i)` gibt das `i`-te Element von `s` zurück und entfernt es
- `s.pop()` gibt das letzte Element von `s` zurück und entfernt es
- `s.extend(t)` hängt das iterierbare Objekt `t` an `s` an
- `s.remove(x)` entfernt das erste Vorkommen des Werts `x`

```
>>> s = [1,2,3,4,5,6]
>>> s.pop(2)
3
>>> s
[1, 2, 4, 5, 6]
>>> s.append(7)
>>> s
[1, 2, 4, 5, 6, 7]
>>> s.pop(2)
4
>>> s
[1, 2, 5, 6, 7]
>>> s.extend([9,10,11])
>>> s
[1, 2, 5, 6, 7, 9, 10, 11]
```

,+'-Operator und append()

- Außer **append** gibt es noch weitere Möglichkeiten, Elemente an eine Liste anzuhängen. So kann man beispielsweise ein oder mehrere Elemente mit dem ,+'-Operator anhängen:

```
>>> x = x + [4,5]
>>> x
[1, 2, 3, 4, 5]
```

- Eine weitere Möglichkeit besteht in der Verwendung des += Operators

```
>>> x = [1,2,3]
>>> x += [4,5]
>>> x
[1, 2, 3, 4, 5]
```

Positionsbestimmung

- `s.index(x[,i,j])` prüft, ob das Element `x` in `s` enthalten ist und liefert den Index zurück. Geprüft wird, sofern angegeben, zwischen den Indizes `i` und `j`.

```
>>> farben = ["rot", "grün", "gelb", "rot"]
>>> farben.index("gelb")
2
>>> farben.index("rot")
0
>>> farben.index("rot",2)
3
```

List comprehension

- Die list comprehension ist eine einfache Methode um Listen zu erzeugen.
- Sie kommen bei der Erzeugung von Listen zum Einsatz, bei denen jedes Element durch Anwendung verschiedener Operationen aus einem Element einer anderen Liste oder eines Iterators erzeugt wird.
- Syntax: Die list comprehension ist von eckigen Klammern umrahmt. Nach der öffnenden Klammer steht ein Ausdruck, der von einem oder mehreren for-Ausdrücken und ggf. von einer if-Bedingung gefolgt wird.

```
>>> squares = []
>>> for i in range(1, 10):
...     squares.append(i**2)
...
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> squares = [i**2 for i in range(1,10)]
>>> squares
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Dictionaries



Was ist ein Dictionary?

- Dictionaries sind assoziative Felder und bestehen aus Schlüssel-Wert-Paaren
- Jedem Schlüssel-Objekt ist ein Wert-Objekt zugeordnet.
- Die Schlüssel-Wert-Paare sind nicht geordnet.
- Dictionaries können leicht verändert werden und während der Laufzeit beliebig wachsen oder schrumpfen.
- Die Indizierung funktioniert wie bei Listen mit eckigen Klammern.

```
>>> raum = {"Brunner": 229, "Dekanat": 125}
>>> raum["Brunner"]
229
>>> raum["Dekanat"]
125
>>> raum["Pirkl"] = 124
>>> raum
{'Brunner': 229, 'Dekanat': 125, 'Pirkl': 124}
>>> raum["Heckmann"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Heckmann'
```


Indizierung eines Dictionaries

- Die Indizierung funktioniert wie bei Listen mit eckigen Klammern.

```
>>> raum = {"Brunner": 229, "Dekanat": 125}
>>> raum["Brunner"]
229
>>> raum["Dekanat"]
125
>>> raum["Pirkl"] = 124
>>> raum
{'Brunner': 229, 'Dekanat': 125, 'Pirkl': 124}
>>> raum["Heckmann"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Heckmann'
```

- Alternativ kann die Methode **get()** verwendet werden:

```
>>> raum.get("Brunner")
229
>>> raum.get("Heckmann")
```

Zulässige Werte für Schlüssel und Werte

- Als Schlüssel dürfen nur immutables verwendet werden (int, float, String, Tupel):

```
>>> coords = {(49.4403198, 11.8633445) : "Amberg", (49.674364, 12.148934) :  
... "Weiden"}  
>>> coords[(49.4403198, 11.8633445)]  
'Amberg'  
>>> dict = {[1,2]: 42}  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unhashable type: 'list'
```

- Die Typen für Schlüssel und Werte brauchen nicht einheitlich zu sein:

```
>>> dict = {'abc': 0.4, 12.5: [1, 2, 3]}  
>>> dict["abc"]  
0.4  
>>> dict[12.5]  
[1, 2, 3]
```

- Bestehen die Schlüssel aus Strings, die den Konventionen für Variablennamen genügen, kann ein Dictionary auch einfacher definiert werden:

```
>>> raum = dict(Brunner=229, Dekanat=125)  
>>> raum  
{'Brunner': 229, 'Dekanat': 125}
```

- Dictionaries können auch verschachtelt werden:

```
>>> de_fr = {"Apfel" : "pomme", "Birne" : "poire"}
>>> de_en = {"Apfel" : "apple", "Birne" : "pear"}
>>> lang_dict = {"de_fr" : de_fr, "de_en" : de_en}
>>> lang_dict["de_en"]["Birne"]
'pear'
>>> lang_dict["de_fr"]["Apfel"]
'pomme'
```

- Mit dem Schlüsselwort „in“ kann überprüft werden, ob ein Schlüssel im Dictionary enthalten ist:

```
>>> "Apfel" in de_fr
True
>>> "Kirsche" in de_fr
False
```

- Mittels len() kann die Länge des Dictionaries ausgegeben werden:

```
>>> de_fr
{'Apfel': 'pomme', 'Birne': 'poire'}
>>> len(de_fr)
2
```

Methoden auf Dictionaries

- Die Zuweisung eines Dictionaries an eine Variable entspricht noch keiner Kopie. Wäre im folgenden Beispiel d eine Kopie, hätte sich der Wert von d nicht ändern dürfen:

```
>>> en_de = {"red" : "rot"}
>>> d = en_de
>>> en_de["yellow"] = "gelb"
>>> en_de
{'red': 'rot', 'yellow': 'gelb'}
>>> d
{'red': 'rot', 'yellow': 'gelb'}
```

- copy():** Erstellung einer (flachen) Kopie:

```
>>> en_de = {"red" : "rot"}
>>> d = en_de.copy()
>>> en_de["green"] = "grün"
>>> en_de
{'red': 'rot', 'green': 'grün'}
>>> d
{'red': 'rot'}
```

Methoden auf Dictionaries

- **from_keys(S[,v])** liefert ein Dictionary zurück mit Werten des unveränderlichen sequentiellen Datentyps S als Schlüssel und einem optionalen Wert „v“ als Wert, der jedem Schlüssel zugeordnet wird:

```
>>> food = ("Apfel", "Birne", "Pflaume")
>>> D = dict.fromkeys(food, "Obst")
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst', 'Pflaume': 'Obst'}
```

- **D.items()** liefert ein Mengen-ähnliches Objekt vom Typ „dict_item“ zurück, was einer View der Schlüssel-Werte-Paare von D entspricht:

```
>>> D.items()
dict_items([('Apfel', 'Obst'), ('Birne', 'Obst'), ('Pflaume', 'Obst')])
```

- **D.pop(k[,d])** entfernt den Schlüssel „k“ zusammen mit seinem Wert aus dem Dictionary und liefert D[k] zurück. Existiert „k“ nicht, wird „d“ zurückgegeben:

```
>>> p = D.pop("Pflaume")
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst'}
>>> p
'Obst'
```

Methoden auf Dictionaries

- **D.popitem()** liefert ein beliebiges Schlüssel-Wert-Paar zurück, das dann aus dem Dictionary D entfernt wird.

```
>>> item = D.popitem()
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst'}
>>> item
('Pflaume', 'Obst')
```

- **D.setdefault(k[,d])** setzt D[k] auf den Wert d, falls der Schlüssel k noch nicht in D enthalten ist. Falls k bereits in D enthalten ist, wird das Dictionary nicht verändert:

```
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst'}
>>> D.setdefault("Pfirsich" , "Obst")
'Obst'
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst', 'Pfirsich': 'Obst'}
>>> D.setdefault("Pfirsich", "Steinobst")
'Obst'
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst', 'Pfirsich': 'Obst'}
```

Methoden auf Dictionaries

- **D.update()** fügt einem Dictionary ein anderes hinzu und überschreibt ggf. die Werte von bereits vorhandenen Schlüsseln:

```
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst', 'Pfirsich': 'Obst'}
>>> D2 = {"Kirsche" : "Obst", "Pfirsich" : "Steinobst"}
>>> D.update(D2)
>>> D
{'Apfel': 'Obst', 'Birne': 'Obst', 'Pfirsich': 'Steinobst', 'Kirsche': 'Obst'}
```

Dictionaries aus Listen erzeugen

- **zip()** kann auf eine beliebige Folge von iterierbaren Objekten angewendet werden, z.B. String, Listen, Tupel, Dictionaries
- Rückgabewert ist ein Iterator, der Tupel zurückliefert:

```
>>> vornamen = ["Peter", "Ingrid"]
>>> nachnamen = ["Müller", "Meier"]
>>> for i in zip(vornamen, nachnamen):
...     print(i)
...
('Peter', 'Müller')
('Ingrid', 'Meier')
>>> list(zip(vornamen, nachnamen))
[('Peter', 'Müller'), ('Ingrid', 'Meier')]
```

- Definition eines Dictionaries aus einem zip-Objekt:

```
>>> namen = dict(zip(vornamen, nachnamen))
>>> namen
{'Peter': 'Müller', 'Ingrid': 'Meier'}
```

- Definition eines Dictionaries aus einer Liste von Listen:

```
>>> namen = dict([vornamen, nachnamen])
>>> namen
{'Peter': 'Ingrid', 'Müller': 'Meier'}
```


Mengen



Der Datentyp „set“

- Ein „set“ enthält eine ungeordnete Sammlung von einmaligen und unveränderlichen Elementen.
- Jedes Element kann nur einmal vorkommen.
- Die Definition erfolgt wie in der gewohnten mathematischen Schreibweise.
- Ein „set“ ist nicht indizierbar.

```
>>> staedte = {"Amberg", "Weiden", "Regensburg"}
>>> type(staedte)
<class 'set'>
>>> len(staedte)
3
>>> "Amberg" in staedte
True
>>> "Nürnberg" in staedte
False
>>> staedte[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

Der Datentyp „set“

- Ein „set“ kann mit Hilfe einer Liste erzeugt werden. Doppelte Einträge werden entfernt:

```
>>> list_names = ["Max", "Tim", "Ina", "Max"]
>>> set_names = set(list_names)
>>> set_names
{'Max', 'Ina', 'Tim'}
```

- Die Werte einer „set“ sind unveränderlich, sie selbst aber nicht. Es können beispielsweise mittels der Funktion **add()** neue Werte eingefügt werden:

```
>>> set_names.add("Leo")
>>> set_names
{'Max', 'Leo', 'Ina', 'Tim'}
```

- Für unveränderliche sets steht der Typ „frozenset“ zur Verfügung:

```
>>> set_names_new = frozenset(set_names)
>>> set_names_new.add("Paul")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Operationen auf „set“-Objekten

- **add(elem)** fügt das Element „elem“ einer Menge hinzu:

```
>>> staedte = {"Amberg"}  
>>> staedte.add("Weiden")  
>>> staedte  
{'Weiden', 'Amberg'}
```

- **clear()** entfernt alle Elemente

```
>>> staedte = {"Amberg", "Weiden"}  
>>> staedte.clear()  
>>> staedte  
set()
```

- **copy()** erzeugt eine (flache) Kopie einer Menge

```
>>> staedte = {"Amberg", "Weiden"}  
>>> staedte_backup = staedte  
>>> staedte_backup = staedte.copy()  
>>> staedte.clear()  
>>> staedte  
set()  
>>> staedte_backup  
{'Weiden', 'Amberg'}
```

Operationen auf „set“-Objekten

- Vereinigung zweier Mengen:

```
>>> set1 = {"a","b"}  
>>> set2 = {"a","c","d"}  
>>> set1.union(set2)  
{'d', 'c', 'a', 'b'}
```

- Schnitt zweier Mengen:

```
>>> set1.intersection(set2)  
{'a'}
```

- Differenzmenge:

```
>>> set1.difference(set2)  
{'b'}  
>>> set2.difference(set1)  
{'d', 'c'}  
>>> set2-set1  
{'d', 'c'}
```

Operationen auf „set“-Objekten

- **remove(elem)** entfernt das Element „elem“. Falls es nicht vorhanden ist, wird ein Fehler ausgegeben.
- **discard(elem)** entfernt das Element „elem“, falls es vorhanden ist. Ansonsten passiert nichts

```
>>> staedte = {"Amberg\","Weiden"}
>>> staedte
{'Weiden', 'Amberg'}
>>> staedte.remove("Amberg")
>>> staedte
{'Weiden'}
>>> staedte.remove("Amberg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Amberg'
>>> staedte.discard("Weiden")
>>> staedte
set()
>>> staedte.discard("Weiden")
>>>
```

Operationen auf „set“-Objekten

- **issubset()** überprüft, ob eine Menge Teilmenge der anderen ist.
- **issuperset()** überprüft, ob eine Menge eine Obermenge der anderen ist.

```
>>> A = {1,2,3,4,5}
>>> B = {3,4,5}
>>> B.issubset(A)
True
>>> A.issuperset(B)
True
```


Verzweigungen



Anweisungsblöcke und Einrückungen

- Die Strukturierung von Programmen in Anweisungsblöcke erfolgt in Python durch Einrückungen (Leerzeichen oder Tabulatoren):

```
anweisungskopf1:
    anweisung1
    anweisung2

    anweisungskopf2:
        anweisung3
        anweisung4
    anweisung5
anweisung6
```

- Programmierer werden hinsichtlich der Blockzugehörigkeit gezwungen, übersichtlichen und unmissverständlichen Code zu schreiben.
- Es sollte vermieden werden, Leerzeichen und Tabulatoren zu mischen.

if-else Anweisung

- Eine if-else Anweisung sieht in Python so aus:

```
if bedingung:  
    anweisung1  
    anweisung2  
else:  
    anweisung3  
anweisung4
```

- Die Anweisungen anweisung1 und anweisung2 werden nur ausgeführt, falls die Bedingung erfüllt ist.
- Die Anweisung anweisung3 wird ausgeführt, falls die Bedingung nicht erfüllt ist.
- Die Anweisung anweisung4 wird nach Abarbeiten des if-else-Blocks ausgeführt.

```
einmaleins.py  
  
antwort = int(input("Was ist 7 mal 8?\n"))  
  
if antwort == 56:  
    print("Korrekt!")  
else:  
    print("Falsch!")
```

Vergleichsoperatoren

Operator	Erklärung	Beispiel	Ergebnis
==	Prüfung auf Gleichheit	42 == 42	True
!=	Prüfung auf Ungleichheit	42 != 42	False
<	Prüfung auf „kleiner“	5 < 3	False
<=	Prüfung auf „kleiner gleich“	3 <= 4	True
>	Prüfung auf „größer“	4 > 2	True
>=	Prüfung auf „größer gleich“	4 >= 4	True

Die Vergleichsoperatoren funktionieren auch für Strings und überprüfen Zeichen für Zeichen die lexikographische Reihenfolge:

```
>>> "Tisch" < "Tischbein"
True
>>> "a" < "b"
True
>>> "A" < "a"
True
```

Logische Verknüpfungen

Operator	Erklärung
and	Logisches und
or	Logisches oder
not	Negation
	Bitweises oder
&	Bitweises und

```
>>> x = [1,2,3]
>>> (2<3) or (x[4]>0)
True
>>> (2<3) | (x[4]>0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Wahrheitswerte für Objekte

- Prinzip: Alles, was nicht False ist, ist True.
- Es muss also noch festgelegt werden, was False ist
 - numerische Null-Werte: 0, 0.0, 0.0+0.0j
 - der boolsche Wert False
 - leere Zeichenketten
 - leere Liste, leere Tupel
 - leere Dictionaries
 - der spezielle Wert None

```
>>> name = "Harald"
>>> if name:
...     print(name)
...
Harald
```

Schleifen



while-Schleife

- Eine while-Schleife hat die folgende Form:

```
while bedingung:  
    statement1  
    statement2  
statement3
```

- statement1 und statement2 werden so oft ausgeführt bis die Bedingung bedingung nicht mehr erfüllt ist.

```
>>> i = 0  
>>> while i<4:  
...     print(i)  
...     i+=1  
...  
0  
1  
2  
3
```

for-Schleife

- In vielen Programmiersprachen enthält der Schleifenkopf einer for-Schleife eine Startanweisung, eine Schleifenbedingung und ein Inkrement:

```
for (i=0; i<10; ++i){  
    ...  
}
```

- In Python wird bei einer for-Schleife stets über eine Sequenz iteriert.

```
for Variable in Sequenz:  
    anweisung1  
    anweisung2  
else:  
    anweisung3
```

- Beispiel:

```
>>> countries = ["Germany", "France", "Netherlands"]  
>>> for country in countries:  
...     print(country)  
...  
Germany  
France  
Netherlands
```


range-Objekt

- Mit **range(start, stop, step)** kann man ein iterierbares Objekt erzeugen, das eine Folge von Integers von start (inklusive) bis stop (exklusive) liefert.
- `range(start, stop)` erzeugt `start, start+1, ..., stop-1`
- `range(stop)` erzeugt `0, 1, ..., stop-1`
- Das optionale Argument `step` gibt das Inkrement an.
- Das `range`-Objekt generiert nicht alle enthaltenen Elemente auf einmal, sondern bei Bedarf wenn darüber iteriert wird.

```
>>> x = range(5)
>>> x
range(0, 5)
>>> type(x)
<class 'range'>
>>> list(x)
[0, 1, 2, 3, 4]
>>> for i in x:
...     print(i)
...
0
1
2
3
4
```

break und continue, else-Teil

- Mittels **break** kann eine Schleife vorzeitig verlassen werden
- Mittels **continue** wird der Rest einer Iteration übersprungen und es wird mit der nächsten Überprüfung der Bedingung fortgefahren.
- Nach einer Schleife kann ein **else**-Teil folgen, der ausgeführt wird, wenn die Bedingung im Schleifenkopf nicht erfüllt ist.

kein_spinat.py

```
essbares = ["Schinken", "Speck", "Spinat", "Nuesse"]
for gericht in essbares:
    if gericht == "Spinat":
        print("Mag ich nicht")
        break
    print("Lecker, " + gericht)
else:
    print("Glueck gehabt, kein Spinat dabei gewesen")
print("Bin satt!")
```

```
Lecker, Schinken
Lecker, Speck
Mag ich nicht
Bin satt!
```

Formatierte Ausgabe und Stringformatierung



print()

- Das Aufrufverhalten der **print()**-Funktion ist wie folgt definiert:

```
print(value1, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- Beispiele:

```
>>> print("hallo", "Welt")
hallo Welt
>>> print("hallo", "Welt", sep="")
halloWelt
>>> print("hallo\n", "Welt")
hallo
    Welt
>>> for i in range(4):
...     print(i, end=" ")
...
0 1 2 3 >>>
```

String-Formatierung im C-Stil

- Man kann bei der String-Formatierung ähnlich wie mit den Funktionen printf und sprintf in C formatierte Strings erzeugen.
- Es erfolgt kein Abschneiden von Werten, sondern Rundung.

```
>>> x = 10.34127
>>> "%10.3e" % x
' 1.034e+01'
>>> "%10.3E" % x
' 1.034E+01'
>>> "%10.3f" % x
'    10.341'
>>> "Wert Nr. %d: %3.2f" % (1, 12.3)
'Wert Nr. 1: 12.30'
```

Platzhaltersymbole

Symbol	Bedeutung
d	vorzeichenbehaftete Ganzzahl (Integer, dezimal)
i	vorzeichenbehaftete Ganzzahl (Integer, dezimal)
o	vorzeichenbehaftete Ganzzahl im Oktalformat
u	obsolet, ansonsten wie d
x	vorzeichenlose Ganzzahlen (hexadezimal)
X	Uppercase-Variante von x
e	Fließkommazahl im Exponentialformat
E	wie e, aber Großbuchstabe für Exponenten „e“
f	Gleitpunktdarstellung
F	wie f, außer dass nan als NAN dargestellt wird
g	entspricht entweder e oder f. Dies wird abhängig von der Größe des Wertes und der gegebenen Präzision entschieden
G	analog zu g, aber es entspricht entweder E oder F
s	String (konvertiert beliebiges Objekt mittels str())

String-Formatierung mit format()

- In Python bevorzugt zu verwendende String-Formatierung. Syntax:

```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

- Beispiel:

```
>>> "Name: {}, Geburtsjahr: {}".format("Angela Merkel", 1954)
'Name: Angela Merkel, Geburtsjahr: 1954'
```

- In geschweiften Klammern kann man Positionsparameter und Formatierungen angeben:

```
>>> "Gewicht={0:3.2f} Alter: {1:d}".format(70.5, 40)
'Gewicht=70.50 Alter: 40'
```

- Anstelle von Positionsparametern können auch Schlüsselwortparameter verwendet werden:

```
>>> "Gewicht={g:3.2f} Alter: {a:d}".format(g=70.5, a=40)
'Gewicht=70.50 Alter: 40'
```


Flaches und tiefes Kopieren



Kopien von Objekten erstellen

- Wir hatten bereits gelernt, dass Variablen Referenzen auf Objekte sind.
- Für unveränderliche Datentypen kann man mittels einfacher Zuweisung eine Kopie erstellen:

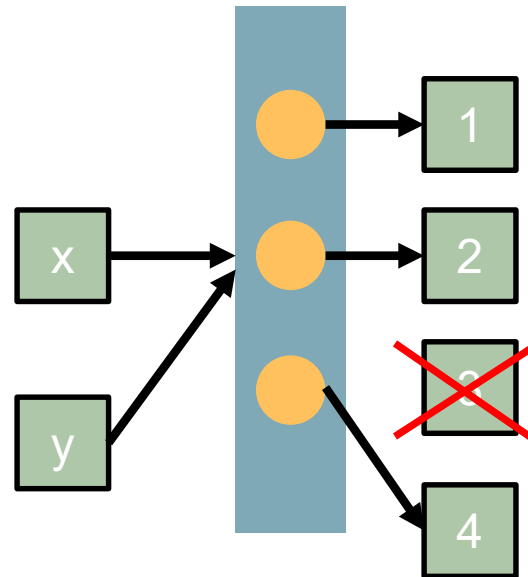
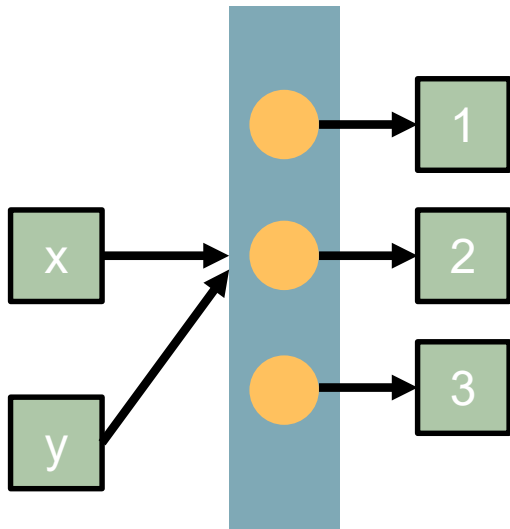
```
>>> x = 3
>>> y = x
>>> id(x)
140727572280208
>>> id(y)
140727572280208
>>> x = 1
>>> y
3
```

- Für veränderliche Datentypen funktioniert das nicht mehr:

```
>>> x = [1,2,3]
>>> y = x
>>> x[2] = 4
>>> y
[1, 2, 4]
```

Kopien von Objekten erstellen

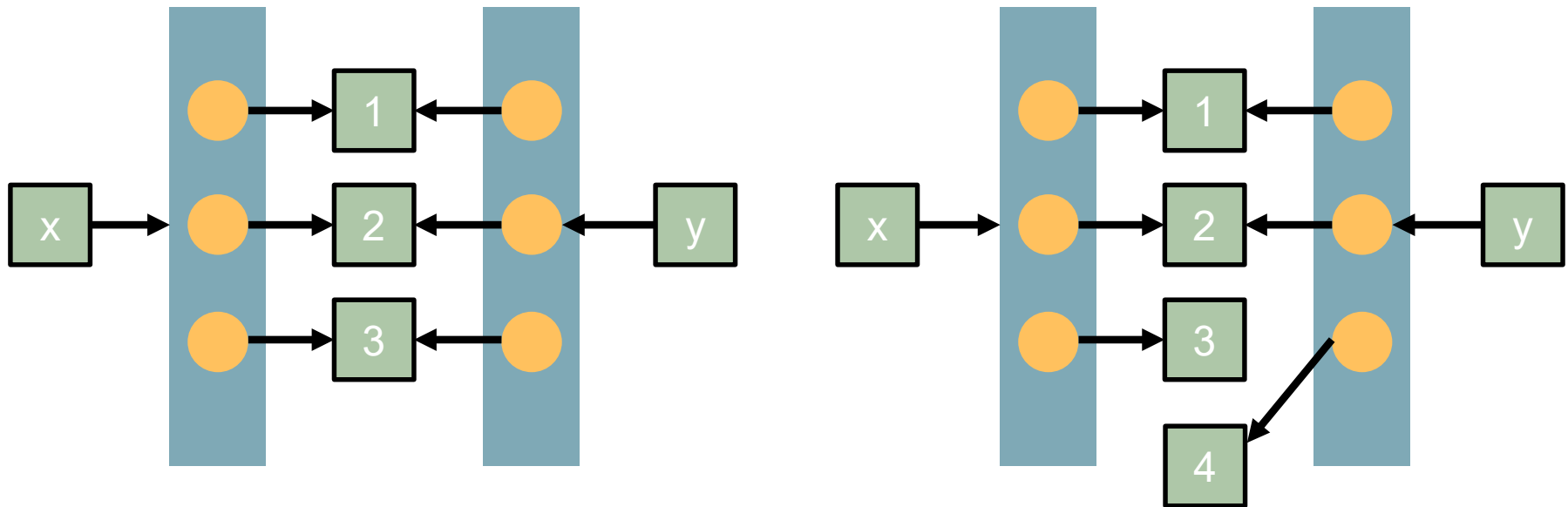
- Die Zuweisung $y=x$ bewirkt, dass y auf dasselbe Objekt im Speicher zeigt wie x .
- Wird die Liste verändert, so ändert sich nicht die Speicherposition. Die Änderung schlägt also auf x und y durch.



Flache Kopien

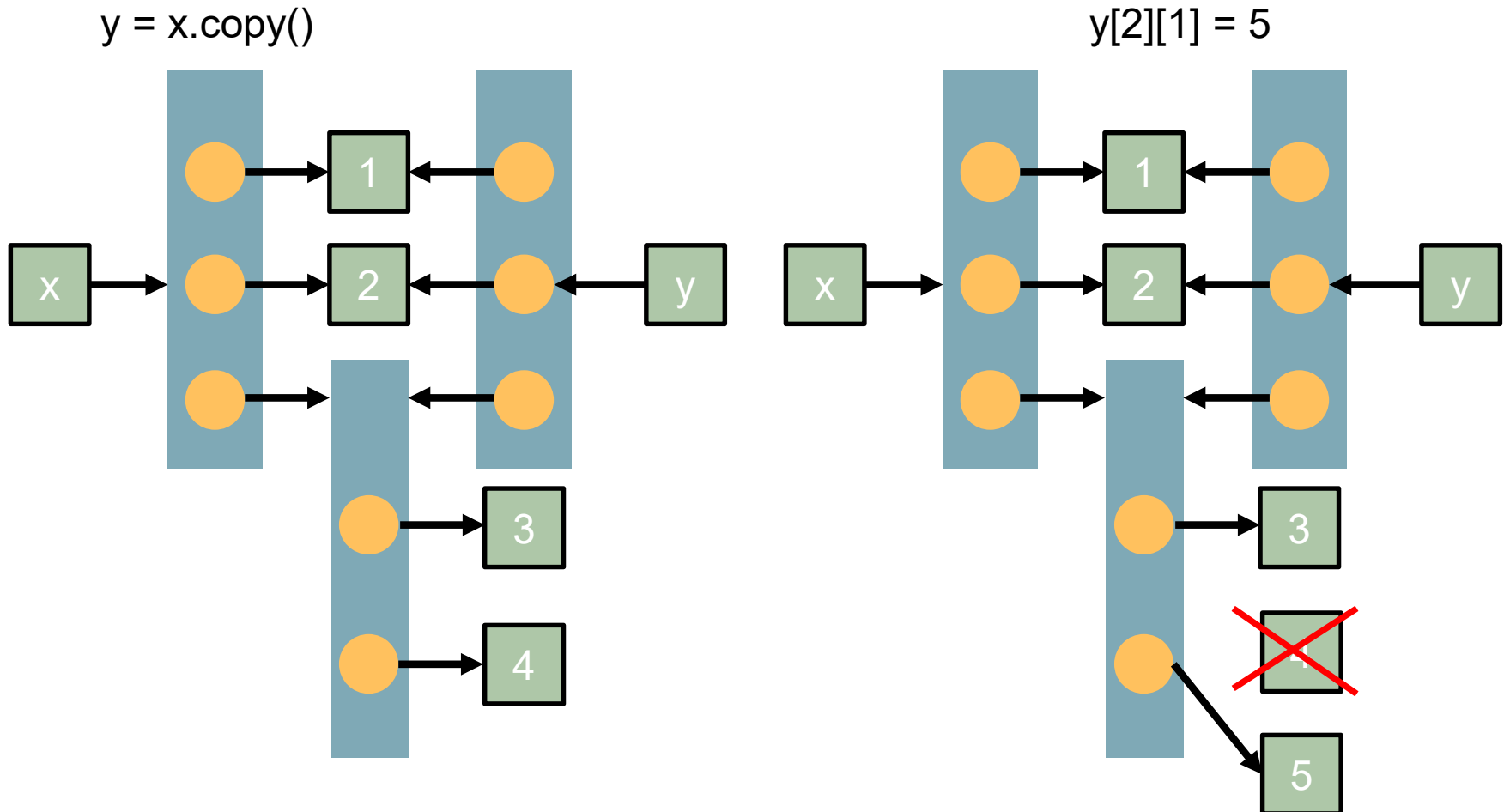
- Mit Hilfe der copy-Methode der list-Klasse können wir flache Kopien von Listen erzeugen.
- Dies bewirkt, dass Zeiger auf die einzelnen Listenelemente kopiert werden.
- Sobald Unterlisten ins Spiel kommen, werden nur Zeiger auf diese Unterlisten kopiert.

Flaches Kopieren: `y = x.copy()`



Flache Kopien

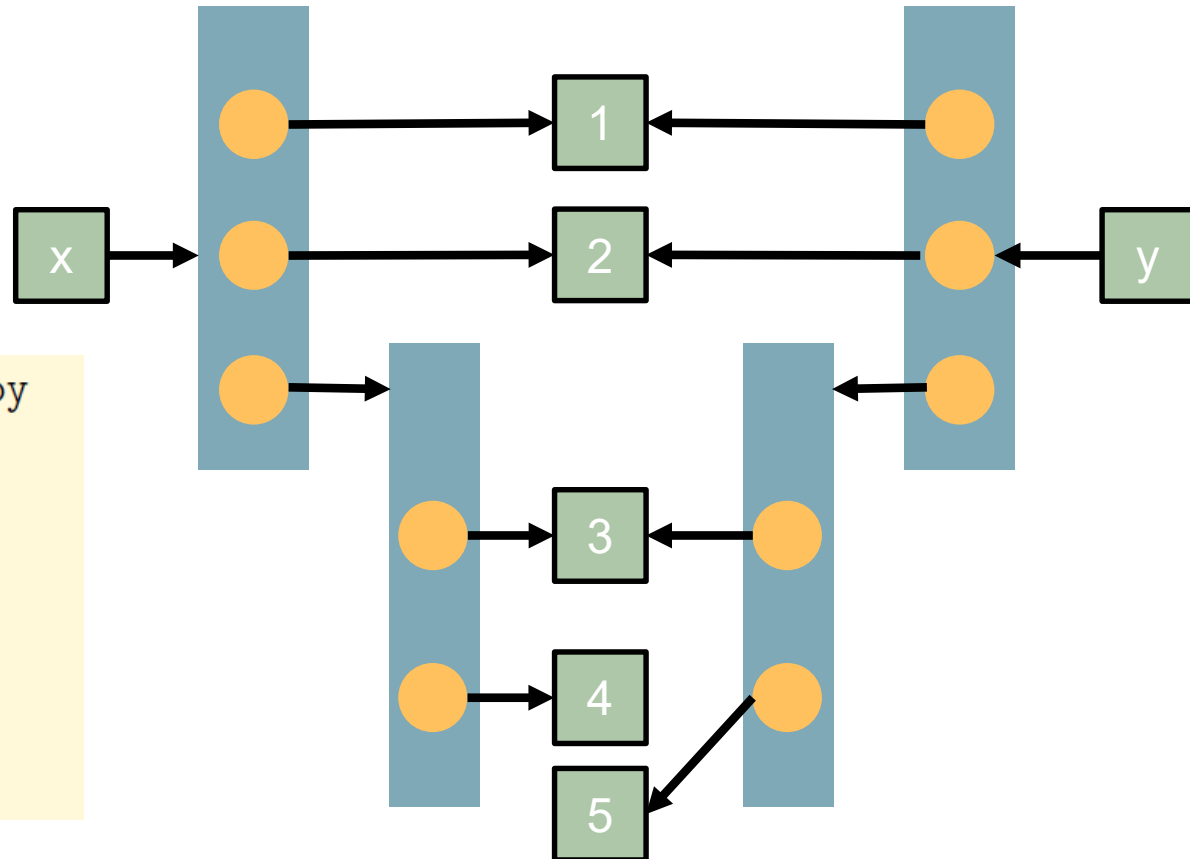
- Ist ein Listenelement wieder eine Liste, so wird durch `copy()` der Zeiger auf diese Liste kopiert.
- Die Änderung eines Eintrags der Unterliste wirkt sich auf beide Listen aus.



Kopieren mit deepcopy

- Abhilfe für das eben beschriebene Problem schafft das Modul copy, das die Funktion **deepcopy** zur Verfügung stellt.

```
>>> from copy import deepcopy
>>> y = [1,2,[3,4]]
>>> x = deepcopy(y)
>>> y[2][1] = 5
>>> y
[1, 2, [3, 5]]
>>> x
[1, 2, [3, 4]]
```



Funktionen



Definition von Funktionen

- Eine Funktionsdefinition in Python wird mit dem Schlüsselwort **def** eingeleitet, gefolgt von einem frei wählbaren Funktionsnamen.

```
def funktionsname(Parameterliste):  
    Anweisung(en)
```

- Der Funktionskörper kann ein oder mehrere **return**-Anweisungen enthalten.
- Eine **return**-Anweisung beendet den Funktionsaufruf, und das Ergebnis des Ausdrucks, der hinter der **return**-Anweisung steht, wird an die aufrufende Stelle zurückgeliefert.
- Falls dem **return** kein Ausdruck folgt, wird der spezielle Wert **None** zurückgeliefert.
- Falls der Funktionsaufruf das Ende des Funktionskörpers erreicht, ohne auf eine **return**-Anweisung zu stoßen, endet der Funktionsaufruf, und es wird ebenfalls der Wert **None** zurückgegeben.

Definition von Funktionen

- Beispiel:

```
sag_hallo.py  
  
def sag_hallo(name):  
    print("Hallo ", name)  
  
sag_hallo("Peter")  
sag_hallo("Melanie")
```

- Ausgabe:

```
User>python sag_hallo.py  
Hallo Peter  
Hallo Melanie
```


- Docstrings stehen direkt nach dem Funktionskopf (nach der „def“-Zeile).
- Der Docstring wird durch den Aufruf `help(Funktionsname)` ausgegeben.

```
>>> def fahrenheit(T_in_celsius):  
...     """ rechnet T_in_celsius in Grad Fahrenheit um """  
...     return T_in_celsius * 9 / 5 + 32  
>>> fahrenheit(10)  
50.0  
>>> help(fahrenheit)  
Help on function fahrenheit in module __main__:  
  
fahrenheit(T_in_celsius)  
    rechnet die Temperatur T_in_celsius in Grad Fahrenheit um
```

Ein Docstring kann auch zu Beginn einer Python-Datei stehen.

```
konvertierungen.py
```

```
"""  
Ein Modul mit Konvertierungshilfen zwischen Grad Celsius in Grad Fahrenheit  
"""  
  
def fahrenheit(T_in_celsius):  
    """ rechnet die Temperatur T_in_Celsius in Grad Fahrenheit um """  
    return T_in_celsius * 9 / 5 + 32  
  
def celsius(T_in_fahrenheit):  
    """ rechnet die Temperatur T_in_fahrenheit in Grad Celsius um """  
    return (T_in_fahrenheit-32) * 5 / 9
```

Docstrings

- Die Datei kann nun wie ein Modul importiert werden.
- Danach kann man `help` auf den Modulnamen aufrufen und erhält folgende Ausgabe:

```
>>> import konvertierungen
>>> help(konvertierungen)
Help on module konvertierungen:

NAME
    konvertierungen - Ein Modul mit Konvertierungshilfen zwischen Grad
    Celsius in Grad Fahrenheit

FUNCTIONS
    celsius(T_in_fahrenheit)
        rechnet die Temperatur T_in_fahrenheit in Grad Celsius

    fahrenheit(T_in_celsius)
        rechnet die Temperatur T_in_Celsius in Graf Fahrenheit

FILE
    c:\users\45a0\konvertierungen.py
```

Default-Werte für Funktionsparameter

- Funktionsparameter können mit Default-Werten belegt werden.
- Funktionsparameter mit Default-Werten sind beim Aufruf optional.
- Wird ein Funktionsparameter beim Aufruf weggelassen, wird der Default-Wert eingesetzt.
- Wird im unten stehenden Beispiel nur ein Argument angegeben, so wird der Wert dem ersten Argument zugewiesen, während für das zweite Argument der Standardwert verwendet wird.
- Mit Hilfe von Schlüsselwortparametern kann man erreichen, dass man nur dem zweiten Argument einen Wert zuweist.

```
>>> def flaeche_rechteck(laenge=1, breite=2):  
...     return laenge*breite  
...  
>>> flaeche_rechteck()  
2  
>>> flaeche_rechteck(3,4)  
12  
>>> flaeche_rechteck(breite=3)           #Schlüsselwortparameter  
3  
>>> flaeche_rechteck(laenge=4)          #Schlüsselwortparameter  
8
```

Funktionen mit mehreren Rückgabeparametern

- Grundsätzlich kann eine Funktion genau ein Objekt zurückliefern.
- Will man beispielsweise mehrere Integers zurückgeben, kann man diese in eine Liste oder ein Tupel packen und dieses zurückgeben:

```
>>> def return_two_numbers():  
...     return (2,3)  
...  
>>> x,y = return_two_numbers()  
>>> x  
2  
>>> y  
3
```

Lokale und globale Variablen

- Variablen sind standardmäßig lokal in einer Funktion.
- Man kann aber auch lesend auf globale Variablen zugreifen.

```
>>> def foo():  
...     print(s)  
...  
>>> s = "Montag"  
>>> foo()  
Montag
```

- Versucht man, s innerhalb von foo zu verändern, kommt ein Fehler:

```
>>> def foo():  
...     print(s)  
...     s = "Dienstag"  
...  
>>> s = "Montag"  
>>> foo()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "<stdin>", line 2, in foo  
UnboundLocalError: local variable 's' referenced before assignment
```

Lokale und globale Variablen

- Will man innerhalb einer Funktion auf eine globale Variable schreibend zugreifen, so muss man diese explizit als global deklarieren:

```
>>> def foo():  
...     global s  
...     print(s)  
...     s = "Dienstag"  
...     print(s)  
...  
>>> s = "Mittwoch"  
>>> foo()  
Mittwoch  
Dienstag  
>>> print(s)  
Dienstag
```

- Definiert man eine Variable innerhalb einer Funktion als global, so muss diese nicht im Namensraum existieren, aus dem die Funktion aufgerufen wird. Die Variable wird zur Laufzeit erzeugt und existiert nach Abarbeitung der Funktion auch im Namensraum, aus dem die Funktion aufgerufen wurde.

Parameterübergabe

- In Python erfolgen Funktionsaufrufe nach dem Mechanismus „Call by Object“
- Werden unveränderliche Argumente wie Integer, Strings oder Tupel übergeben, verhält sich die Übergabe wie eine Wertübergabe.
- Die Referenz auf das unveränderliche Objekt wird an den formalen Parameter der Funktion übertragen.
- Innerhalb der Funktion kann der Inhalt des Objekts nicht verändert werden.

```
>>> def foo(x):  
...     print("Id(x) =", id(x), ", x =", x)  
...     x = 3  
...     print("Id(x) =", id(x), ", x =", x)  
...  
>>> x = 1  
>>> print("Id(x) =", id(x), ", x =", x)  
Id(x) = 140719687119696 , x = 1  
>>> foo(x)  
Id(x) = 140719687119696 , x = 1  
Id(x) = 140719687119760 , x = 3  
>>> print("Id(x) =", id(x), ", x =", x)  
Id(x) = 140719687119696 , x = 1
```


Effekte bei veränderlichen Objekten

- Gerade haben wir gesehen, dass man unveränderliche Objekte, die man als formale Parameter an eine Funktion übergibt, in der Funktion nicht verändern kann.
- Anders sieht es mit veränderlichen Objekten wie Listen oder Dictionaries aus. Diese können innerhalb der Funktion verändert werden.

```
>>> def fun(list):  
...     list.append(10)  
...  
>>> x = [1,2,3]  
>>> fun(x)  
>>> x  
[1, 2, 3, 10]
```

- Frage: was kommt hier raus?

```
>>> def fun(list):  
...     list = list + [10]  
...  
>>> x = [1,2,3]  
>>> fun(x)  
>>> x  
???
```