

Python für Einsteiger

Tutorium über drei Stunden

Thomas Lotze

gocept gmbh & co. kg
Halle (Saale)

29.10.2012, PyCon DE, Leipzig

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- Über das Thema: Was ist Python?
 - interpretiert: schneller Entwicklungszyklus, einbetten
 - imperativ mit Anleihen bei funktionalen Sprachen
 - intuitiv, insbesondere gut lesbar
- Über den Tutor: Python bei gocept
 - gocept arbeitet von Anfang an (2000) mit Python und Zope
 - persönlich seit 2004
- Über das Publikum: Vorwissen, Erwartungen, Wünsche?

- 1 Einleitung
- 2 **Hallo Welt**
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 **Hallo Welt**
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Der Python-Interpreter

- starten, z.B.

```
$ ../Python-3.3/bin/idle3
```

- Fenster öffnet sich

```
Python 3.3.0 (default, Oct 28 2012, 09:13:40)
[GCC 4.6.3] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

- Funktionstest: Taschenrechner

```
>>> 2 + 2
4
```

- Eingaben wiederholen, im Shell-Fenster suchen

Der Python-Interpreter

- starten, z.B.

```
$ ../Python-3.3/bin/idle3
```

- Fenster öffnet sich

```
Python 3.3.0 (default, Oct 28 2012, 09:13:40)
[GCC 4.6.3] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

- Funktionstest: Taschenrechner

```
>>> 2 + 2
4
```

- Eingaben wiederholen, im Shell-Fenster suchen

Der Python-Interpreter

- starten, z.B.

```
$ ../Python-3.3/bin/idle3
```

- Fenster öffnet sich

```
Python 3.3.0 (default, Oct 28 2012, 09:13:40)
[GCC 4.6.3] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

- Funktionstest: Taschenrechner

```
>>> 2 + 2
```

```
4
```

- Eingaben wiederholen, im Shell-Fenster suchen

Der Python-Interpreter

- starten, z.B.

```
$ ../Python-3.3/bin/idle3
```

- Fenster öffnet sich

```
Python 3.3.0 (default, Oct 28 2012, 09:13:40)  
[GCC 4.6.3] on linux  
Type "copyright", "credits" or "license()" for more information.  
>>>
```

- Funktionstest: Taschenrechner

```
>>> 2 + 2
```

```
4
```

- Eingaben wiederholen, im Shell-Fenster suchen

Python als Taschenrechner

- Grundrechenarten, Klammern

```
>>> (10 - 2) * 4  
32
```

- Gleitkommazahlen

```
>>> (10 - 2) / 4  
2.0
```

```
>>> (10.5 - 2.5) * 4  
32.0
```

- Potenzen und Ganzzahldivision

```
>>> 2 ** 3  
8
```

```
>>> 10 // 4  
2
```

```
>>> -10 // 4  
-3
```

Python als Taschenrechner

- Grundrechenarten, Klammern

```
>>> (10 - 2) * 4  
32
```

- Gleitkommazahlen

```
>>> (10 - 2) / 4  
2.0
```

```
>>> (10.5 - 2.5) * 4  
32.0
```

- Potenzen und Ganzzahldivision

```
>>> 2 ** 3  
8
```

```
>>> 10 // 4  
2
```

```
>>> -10 // 4  
-3
```

Python als Taschenrechner

- Grundrechenarten, Klammern

```
>>> (10 - 2) * 4  
32
```

- Gleitkommazahlen

```
>>> (10 - 2) / 4  
2.0
```

```
>>> (10.5 - 2.5) * 4  
32.0
```

- Potenzen und Ganzzahldivision

```
>>> 2 ** 3  
8
```

```
>>> 10 // 4  
2
```

```
>>> -10 // 4  
-3
```

Ausdrücke und Anweisungen

- gesehen: Python-Shell liest Ausdruck ein, berechnet ihn, gibt seinen Wert aus

```
>>> 2 + 2  
4
```

- auch möglich: Python-Shell liest Anweisung ein, führt sie aus, kein Wert zum Ausgeben; Beispiel: Zuweisung

```
>>> x = 2 + 2
```

- Anweisungen haben Nebenwirkungen: Variable mit Wert belegt, kann in Ausdrücken verwendet werden

```
>>> x  
4  
>>> x + 3  
7
```

Ausdrücke und Anweisungen

- gesehen: Python-Shell liest Ausdruck ein, berechnet ihn, gibt seinen Wert aus

```
>>> 2 + 2
4
```

- auch möglich: Python-Shell liest Anweisung ein, führt sie aus, kein Wert zum Ausgeben; Beispiel: Zuweisung

```
>>> x = 2 + 2
```

- Anweisungen haben Nebenwirkungen: Variable mit Wert belegt, kann in Ausdrücken verwendet werden

```
>>> x
4
>>> x + 3
7
```

Ausdrücke und Anweisungen

- gesehen: Python-Shell liest Ausdruck ein, berechnet ihn, gibt seinen Wert aus

```
>>> 2 + 2  
4
```

- auch möglich: Python-Shell liest Anweisung ein, führt sie aus, kein Wert zum Ausgeben; Beispiel: Zuweisung

```
>>> x = 2 + 2
```

- Anweisungen haben Nebenwirkungen: Variable mit Wert belegt, kann in Ausdrücken verwendet werden

```
>>> x  
4  
>>> x + 3  
7
```

- Beispiel: unverständliche Eingabe

```
>>> 2 2
```

```
SyntaxError: invalid syntax
```

- Beispiel: unbekannte Variable wird benutzt

```
>>> y
```

```
Traceback (most recent call last):
```

```
...
```

```
NameError: name 'y' is not defined
```

- Fehler werden abgefangen, Interpreter läuft weiter
- Fehlermeldung informativ: Art des Fehlers, ggf. Ort im Programm

- Beispiel: unverständliche Eingabe

```
>>> 2 2
```

```
SyntaxError: invalid syntax
```

- Beispiel: unbekannte Variable wird benutzt

```
>>> y
```

```
Traceback (most recent call last):
```

```
...
```

```
NameError: name 'y' is not defined
```

- Fehler werden abgefangen, Interpreter läuft weiter
- Fehlermeldung informativ: Art des Fehlers, ggf. Ort im Programm

- Beispiel: unverständliche Eingabe

```
>>> 2 2
```

```
SyntaxError: invalid syntax
```

- Beispiel: unbekannte Variable wird benutzt

```
>>> y
```

```
Traceback (most recent call last):
```

```
...
```

```
NameError: name 'y' is not defined
```

- Fehler werden abgefangen, Interpreter läuft weiter
- Fehlermeldung informativ: Art des Fehlers, ggf. Ort im Programm

- Beispiel: unverständliche Eingabe

```
>>> 2 2
```

```
SyntaxError: invalid syntax
```

- Beispiel: unbekannte Variable wird benutzt

```
>>> y
```

```
Traceback (most recent call last):
```

```
...
```

```
NameError: name 'y' is not defined
```

- Fehler werden abgefangen, Interpreter läuft weiter
- Fehlermeldung informativ: Art des Fehlers, ggf. Ort im Programm

- 1 Einleitung
- 2 **Hallo Welt**
 - Die Entwicklungsumgebung `idle`
 - **Python-Programme**
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Programm schreiben und ausführen

- in idle neues Fenster öffnen, Programm schreiben

```
1 x = 1
2 x
```

- Programm speichern (C-S), ausführen (F5)

```
>>> ===== RESTART =====
>>>
```

- keine Ausgabe: Ausdruck wird zwar ausgewertet, aber nichts sorgt dafür, daß der Wert ausgegeben wird

Programm schreiben und ausführen

- in idle neues Fenster öffnen, Programm schreiben

```
1 x = 1
2 x
```

- Programm speichern (C-S), ausführen (F5)

```
>>> ===== RESTART =====
>>>
```

- keine Ausgabe: Ausdruck wird zwar ausgewertet, aber nichts sorgt dafür, daß der Wert ausgegeben wird

Programm schreiben und ausführen

- in idle neues Fenster öffnen, Programm schreiben

```
1 x = 1
2 x
```

- Programm speichern (C-S), ausführen (F5)

```
>>> ===== RESTART =====
>>>
```

- keine Ausgabe: Ausdruck wird zwar ausgewertet, aber nichts sorgt dafür, daß der Wert ausgegeben wird

Die Funktion print()

- Wert eines Ausdrucks auf die Standardausgabe drucken

```
x = 1  
print(x)
```

- Zeichenketten ausgeben: print() druckt den Inhalt

```
>>> print('Hallo Welt\n=====')  
Hallo Welt  
=====
```

- Gegensatz: Interpreter gibt eine Darstellung aus

```
>>> 'Hallo Welt\n=====  
'Hallo Welt\n====='
```

Die Funktion print()

- Wert eines Ausdrucks auf die Standardausgabe drucken

```
x = 1  
print(x)
```

- Zeichenketten ausgeben: print() druckt den Inhalt

```
>>> print('Hallo Welt\n=====')  
Hallo Welt  
=====
```

- Gegensatz: Interpreter gibt eine Darstellung aus

```
>>> 'Hallo Welt\n=====  
'Hallo Welt\n====='
```

Die Funktion print()

- Wert eines Ausdrucks auf die Standardausgabe drucken

```
x = 1  
print(x)
```

- Zeichenketten ausgeben: print() druckt den Inhalt

```
>>> print('Hallo Welt\n=====')  
Hallo Welt  
=====
```

- Gegensatz: Interpreter gibt eine Darstellung aus

```
>>> 'Hallo Welt\n=====  
'Hallo Welt\n====='
```

- Objektdarstellung macht Art des Objekts deutlich, nützlich für Programmierer

```
>>> 2012
```

```
2012
```

```
>>> '2012'
```

```
'2012'
```

- Ausgabe von `print()` ist für Benutzer gedacht

```
>>> print(2012)
```

```
2012
```

```
>>> print('2012')
```

```
2012
```

- Objektdarstellung mit `repr()` erzeugen

```
>>> print(repr(2012))
```

```
2012
```

```
>>> print(repr('2012'))
```

```
'2012'
```

- Objektdarstellung macht Art des Objekts deutlich, nützlich für Programmierer

```
>>> 2012
2012
>>> '2012'
'2012'
```

- Ausgabe von `print()` ist für Benutzer gedacht

```
>>> print(2012)
2012
>>> print('2012')
2012
```

- Objektdarstellung mit `repr()` erzeugen

```
>>> print(repr(2012))
2012
>>> print(repr('2012'))
'2012'
```

- Objektdarstellung macht Art des Objekts deutlich, nützlich für Programmierer

```
>>> 2012
2012
>>> '2012'
'2012'
```

- Ausgabe von `print()` ist für Benutzer gedacht

```
>>> print(2012)
2012
>>> print('2012')
2012
```

- Objektdarstellung mit `repr()` erzeugen

```
>>> print(repr(2012))
2012
>>> print(repr('2012'))
'2012'
```

Eingebaute Funktionen

- `print()`, `repr()` bereits gesehen
- `help()`: interaktive Hilfe
- Hilfe zu Funktionen: `help(print)`, `help(help)`
- Variablen im Interpreter: `dir()`, `vars()`

Eingebaute Funktionen

- `print()`, `repr()` bereits gesehen
- `help()`: interaktive Hilfe
- Hilfe zu Funktionen: `help(print)`, `help(help)`
- Variablen im Interpreter: `dir()`, `vars()`

Eingebaute Funktionen

- `print()`, `repr()` bereits gesehen
- `help()`: interaktive Hilfe
- Hilfe zu Funktionen: `help(print)`, `help(help)`
- Variablen im Interpreter: `dir()`, `vars()`

Eingebaute Funktionen

- `print()`, `repr()` bereits gesehen
- `help()`: interaktive Hilfe
- Hilfe zu Funktionen: `help(print)`, `help(help)`
- Variablen im Interpreter: `dir()`, `vars()`

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Beispiel: while-Schleife

- bisher Einzeiler benutzt: syntaktisch trivial
- Blöcke: aufeinander bezogene Zeilen

```
x = 1
while x < 100:
    print(x)
    x *= 2
```

- Ergebnis:

```
1
2
4
8
16
32
64
```

Beispiel: while-Schleife

- bisher Einzeiler benutzt: syntaktisch trivial
- Blöcke: aufeinander bezogene Zeilen

```
x = 1
while x < 100:
    print(x)
    x *= 2
```

- Ergebnis:

```
1
2
4
8
16
32
64
```

Beispiel: while-Schleife

- bisher Einzeiler benutzt: syntaktisch trivial
- Blöcke: aufeinander bezogene Zeilen

```
x = 1
while x < 100:
    print(x)
    x *= 2
```

- Ergebnis:

```
1
2
4
8
16
32
64
```

Zum Vergleich: Ruby und C

```
x = 1
while x < 100
  puts x
  x *= 2
end
```

```
x = 1
while x < 100
  puts(x)
  x *= 2
end
```

```
#include <stdio.h>
void main(int argc, char *argv[]) {
```

```
    int x = 1;
    while (x < 100) {
        printf("%d\n", x);
        x *= 2;
    }
```

```
    int x = 1;
    do {
        printf("%d\n", x);
        x *= 2;
    } while (x < 100);
```

```
}
```

Zum Vergleich: Ruby und C

```
x = 1
while x < 100
  puts x
  x *= 2
end
```

```
x = 1
while x < 100
  puts(x)
  x *= 2
end
```

```
#include <stdio.h>
```

```
void main(int argc, char *argv[]) {
```

```
  int x = 1;
  while (x < 100) {
    printf("%d\n", x);
    x *= 2;
  }
```

```
  int x = 1;
  do {
    printf("%d\n", x);
    x *= 2;
  } while (x < 100);
```

```
}
```

- Blöcke eingerückt, durch :
eingeleitet, implizites Ende
- implizite Zeilenenden, kein ;
- minimalistische Syntax:
keine Klammern {}, keine unnötigen Schlüsselwörter (do)
- Einheitlichkeit: immer Klammern um Funktionsargumente,
nur eine Art von while-Schleifen
- Variablen implizit deklariert
- wesentliche Funktionen schon eingebaut
- kein Grundgerüst (main)
- sinnvolle Vorgaben: Umwandlung in String und
Zeilenumbruch implizit in print()

```
x = 1
while x < 100:
    print(x)
    x *= 2
```

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - **Grundlegende Programmkonstrukte**
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- while-Schleifen bereits gesehen
- for-Schleifen iterieren über Sammlungen von Elementen

```
>>> for x in [1, 2, 4, 7]:  
...     print(x, end=' ')  
1 2 4 7
```

- Iteration über Zahlenfolge mit range()

```
>>> for x in range(3, 7):  
...     print(x, end=' ')  
3 4 5 6
```

- ausdrucksstärker als Mechanik mit Schritt und Abbruch
- Konvention in Python: halboffene Intervalle, $a \leq x < b$

- while-Schleifen bereits gesehen
- for-Schleifen iterieren über Sammlungen von Elementen

```
>>> for x in [1, 2, 4, 7]:  
...     print(x, end=' ')  
1 2 4 7
```

- Iteration über Zahlenfolge mit range()

```
>>> for x in range(3, 7):  
...     print(x, end=' ')  
3 4 5 6
```

- ausdrucksstärker als Mechanik mit Schritt und Abbruch
- Konvention in Python: halboffene Intervalle, $a \leq x < b$

- while-Schleifen bereits gesehen
- for-Schleifen iterieren über Sammlungen von Elementen

```
>>> for x in [1, 2, 4, 7]:  
...     print(x, end=' ')  
1 2 4 7
```

- Iteration über Zahlenfolge mit range()

```
>>> for x in range(3, 7):  
...     print(x, end=' ')  
3 4 5 6
```

- ausdrucksstärker als Mechanik mit Schritt und Abbruch
- Konvention in Python: halboffene Intervalle, $a \leq x < b$

- while-Schleifen bereits gesehen
- for-Schleifen iterieren über Sammlungen von Elementen

```
>>> for x in [1, 2, 4, 7]:  
...     print(x, end=' ')  
1 2 4 7
```

- Iteration über Zahlenfolge mit range()

```
>>> for x in range(3, 7):  
...     print(x, end=' ')  
3 4 5 6
```

- ausdrucksstärker als Mechanik mit Schritt und Abbruch
- Konvention in Python: halboffene Intervalle, $a \leq x < b$

- while-Schleifen bereits gesehen
- for-Schleifen iterieren über Sammlungen von Elementen

```
>>> for x in [1, 2, 4, 7]:  
...     print(x, end=' ')  
1 2 4 7
```

- Iteration über Zahlenfolge mit range()

```
>>> for x in range(3, 7):  
...     print(x, end=' ')  
3 4 5 6
```

- ausdrucksstärker als Mechanik mit Schritt und Abbruch
- Konvention in Python: halboffene Intervalle, $a \leq x < b$

Bedingungen

- einfache Alternative

```
if x > 0:  
    print('positiv')  
else:  
    print('nicht positiv')
```

- mehrfache Fallunterscheidung

```
if x > 0:  
    print('positiv')  
elif x < 0:  
    print('negativ')  
else:  
    print('null')
```

- Python kennt kein switch, nur Mehrfachbedingungen

Bedingungen

- einfache Alternative

```
if x > 0:  
    print('positiv')  
else:  
    print('nicht positiv')
```

- mehrfache Fallunterscheidung

```
if x > 0:  
    print('positiv')  
elif x < 0:  
    print('negativ')  
else:  
    print('null')
```

- Python kennt kein switch, nur Mehrfachbedingungen

- einfache Alternative

```
if x > 0:  
    print('positiv')  
else:  
    print('nicht positiv')
```

- mehrfache Fallunterscheidung

```
if x > 0:  
    print('positiv')  
elif x < 0:  
    print('negativ')  
else:  
    print('null')
```

- Python kennt kein switch, nur Mehrfachbedingungen

Schleifen steuern: continue, break

- Iterationsschritt überspringen:

```
>>> i = 0
>>> while i < 5:
...     i += 1
...     if i == 3:
...         continue
...     print(i, end=' ')
1 2 4 5
```

- fußgesteuerte Schleife:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
i = 0
while True:
    print(i)
    i += 1
    if i >= 5:
        break
```

Schleifen steuern: continue, break

- Iterationsschritt überspringen:

```
>>> i = 0
>>> while i < 5:
...     i += 1
...     if i == 3:
...         continue
...     print(i, end=' ')
1 2 4 5
```

- fußgesteuerte Schleife:

```
i = 0
while i < 5:
    print(i)
    i += 1
```

```
i = 0
while True:
    print(i)
    i += 1
    if i >= 5:
        break
```

Schleifen steuern: break und else

```
>>> n = 15
>>> for i in range(2, n):
...     if n/i == n//i:
...         print('Teiler:', i)
...         break
...     else:
...         print('Primzahl')
```

Teiler: 3

```
>>> n = 17
>>> for i in range(2, n):
...     
```

```
...
Primzahl
```

Übung: Schleifen steuern

- Aufgabe:

- Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben

- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```

- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:

- Zeilen einer Datei lesen
- nichtleere Zeilen zählen und ausgeben
- am Ende Anzahl der nichtleeren Zeilen ausgeben
- bei zu langer Zeile abbrechen, Fehler ausgeben

- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```

- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:

- Zeilen einer Datei lesen
- nichtleere Zeilen zählen und ausgeben
- am Ende Anzahl der nichtleeren Zeilen ausgeben
- bei zu langer Zeile abbrechen, Fehler ausgeben

- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```

- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben

- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```

- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben
- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```
- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben
- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```
- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben
- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```
- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben
- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```
- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Aufgabe:
 - Zeilen einer Datei lesen
 - nichtleere Zeilen zählen und ausgeben
 - am Ende Anzahl der nichtleeren Zeilen ausgeben
 - bei zu langer Zeile abbrechen, Fehler ausgeben
- Datei lesen: `open()`, iterierbar wie `range()`

```
for line in open('foo.txt'):  
    print(line)
```

- leere Zeile testen:

```
if line == '\n':
```
- Länge einer Zeichenkette bestimmen: `len()`
- `print(x, y, z)` druckt mehrere Dinge nacheinander
- `continue`, `break`, `else`

Übung: Schleifen steuern

- Lösung:

```
i = 0
for line in open('foo.txt'):
    if line == '\n':
        continue

    i += 1
    if len(line) > 80:
        print('Zeile zu lang')
        break

    print(line, end='')
else:
    print('nichtleere Zeilen:', i)
```

Schleifen und Bedingungen in Ausdrücken

- Listenausdrücke

```
>>> [2*x for x in range(5)]  
[0, 2, 4, 6, 8]
```

- Listenausdrücke mit Filter

```
>>> [x/2 for x in range(5) if x % 2 == 0]  
[0.0, 1.0, 2.0]
```

- bedingte Ausdrücke

```
>>> x = -1  
>>> ('positiv' if x > 0 else  
      'negativ' if x < 0 else 'null')  
'negativ'
```

- ausdrucksstärker, weniger mechanisch

- kann verschachtelt werden, wird aber unübersichtlich

Schleifen und Bedingungen in Ausdrücken

- Listenausdrücke

```
>>> [2*x for x in range(5)]  
[0, 2, 4, 6, 8]
```

- Listenausdrücke mit Filter

```
>>> [x/2 for x in range(5) if x % 2 == 0]  
[0.0, 1.0, 2.0]
```

- bedingte Ausdrücke

```
>>> x = -1  
>>> ('positiv' if x > 0 else  
      'negativ' if x < 0 else 'null')  
'negativ'
```

- ausdrucksstärker, weniger mechanisch

- kann verschachtelt werden, wird aber unübersichtlich

Schleifen und Bedingungen in Ausdrücken

- Listenausdrücke

```
>>> [2*x for x in range(5)]  
[0, 2, 4, 6, 8]
```

- Listenausdrücke mit Filter

```
>>> [x/2 for x in range(5) if x % 2 == 0]  
[0.0, 1.0, 2.0]
```

- bedingte Ausdrücke

```
>>> x = -1  
>>> ('positiv' if x > 0 else  
      'negativ' if x < 0 else 'null')  
'negativ'
```

- ausdrucksstärker, weniger mechanisch

- kann verschachtelt werden, wird aber unübersichtlich

Schleifen und Bedingungen in Ausdrücken

- Listenausdrücke

```
>>> [2*x for x in range(5)]  
[0, 2, 4, 6, 8]
```

- Listenausdrücke mit Filter

```
>>> [x/2 for x in range(5) if x % 2 == 0]  
[0.0, 1.0, 2.0]
```

- bedingte Ausdrücke

```
>>> x = -1  
>>> ('positiv' if x > 0 else  
     'negativ' if x < 0 else 'null')  
'negativ'
```

- ausdrucksstärker, weniger mechanisch

- kann verschachtelt werden, wird aber unübersichtlich

Schleifen und Bedingungen in Ausdrücken

- Listenausdrücke

```
>>> [2*x for x in range(5)]  
[0, 2, 4, 6, 8]
```

- Listenausdrücke mit Filter

```
>>> [x/2 for x in range(5) if x % 2 == 0]  
[0.0, 1.0, 2.0]
```

- bedingte Ausdrücke

```
>>> x = -1  
>>> ('positiv' if x > 0 else  
     'negativ' if x < 0 else 'null')  
'negativ'
```

- ausdrucksstärker, weniger mechanisch

- kann verschachtelt werden, wird aber unübersichtlich

- Algorithmen wiederverwendbar machen, Abstraktion
- Funktionen auf verschiedene Parameter anwenden, Ergebnis bekommen
- Funktionen definieren mit def:

```
def algorithm(input1, input2,  
              option1='foo', option2=False):  
    """Apply the algorithm."""  
    pass
```

- Wert des Aufrufs zurückgeben mit return
- impliziter Rückgabewert None

- Algorithmen wiederverwendbar machen, Abstraktion
- Funktionen auf verschiedene Parameter anwenden, Ergebnis bekommen
- Funktionen definieren mit def:

```
def algorithm(input1, input2,  
              option1='foo', option2=False):  
    """Apply the algorithm."""  
    pass
```

- Wert des Aufrufs zurückgeben mit return
- impliziter Rückgabewert None

- Algorithmen wiederverwendbar machen, Abstraktion
- Funktionen auf verschiedene Parameter anwenden, Ergebnis bekommen
- Funktionen definieren mit def:

```
def algorithm(input1, input2,  
              option1='foo', option2=False):  
    """Apply the algorithm."""  
    pass
```

- Wert des Aufrufs zurückgeben mit return
- impliziter Rückgabewert None

- Algorithmen wiederverwendbar machen, Abstraktion
- Funktionen auf verschiedene Parameter anwenden, Ergebnis bekommen
- Funktionen definieren mit def:

```
def algorithm(input1, input2,  
              option1='foo', option2=False):  
    """Apply the algorithm."""  
    pass
```

- Wert des Aufrufs zurückgeben mit return
- impliziter Rückgabewert None

- Algorithmen wiederverwendbar machen, Abstraktion
- Funktionen auf verschiedene Parameter anwenden, Ergebnis bekommen
- Funktionen definieren mit def:

```
def algorithm(input1, input2,  
              option1='foo', option2=False):  
    """Apply the algorithm."""  
    pass
```

- Wert des Aufrufs zurückgeben mit return
- impliziter Rückgabewert None

- Primzahlbestimmung: Suche nach Teiler ausgliedern

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            factor = i  
            break  
    else:  
        factor = None  
    return factor
```

- Abstraktion: Teiler als Zahl zurückgeben
- Alternative:

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            return i
```

- Primzahlbestimmung: Suche nach Teiler ausgliedern

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            factor = i  
            break  
    else:  
        factor = None  
    return factor
```

- Abstraktion: Teiler als Zahl zurückgeben

- Alternative:

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            return i
```

- Primzahlbestimmung: Suche nach Teiler ausgliedern

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            factor = i  
            break  
    else:  
        factor = None  
    return factor
```

- Abstraktion: Teiler als Zahl zurückgeben
- Alternative:

```
def factor(n):  
    for i in range(2, n):  
        if n/i == n//i:  
            return i
```

- Textausgabe ausgliedern:

```
def prime(n):  
    f = factor(n)  
    if f is None:  
        print('Primzahl')  
    else:  
        print('Teiler:', f)
```

- Aufrufen:

```
>>> prime(15)  
Teiler: 3  
>>> prime(17)  
Primzahl
```

- Funktionen können Nebenwirkung haben

- Textausgabe ausgliedern:

```
def prime(n):  
    f = factor(n)  
    if f is None:  
        print('Primzahl')  
    else:  
        print('Teiler:', f)
```

- Aufrufen:

```
>>> prime(15)  
Teiler: 3  
>>> prime(17)  
Primzahl
```

- Funktionen können Nebenwirkung haben

- Textausgabe ausgliedern:

```
def prime(n):  
    f = factor(n)  
    if f is None:  
        print('Primzahl')  
    else:  
        print('Teiler:', f)
```

- Aufrufen:

```
>>> prime(15)  
Teiler: 3  
>>> prime(17)  
Primzahl
```

- Funktionen können Nebenwirkung haben

zurück zum Beispielprogramm, Aufgabe:

- Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
- Grenze für Zeilenlänge konfigurierbar machen
- sinnvoller Vorgabewert
- Zeilenzahl soll weiterverarbeitet werden können
- Fehlerbehandlung

zurück zum Beispielprogramm, Aufgabe:

- Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
- Grenze für Zeilenlänge konfigurierbar machen
- sinnvoller Vorgabewert
- Zeilenzahl soll weiterverarbeitet werden können
- Fehlerbehandlung

zurück zum Beispielprogramm, Aufgabe:

- Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
- Grenze für Zeilenlänge konfigurierbar machen
- sinnvoller Vorgabewert
- Zeilenzahl soll weiterverarbeitet werden können
- Fehlerbehandlung

zurück zum Beispielprogramm, Aufgabe:

- Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
- Grenze für Zeilenlänge konfigurierbar machen
- sinnvoller Vorgabewert
- Zeilenzahl soll weiterverarbeitet werden können
- Fehlerbehandlung

zurück zum Beispielprogramm, Aufgabe:

- Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
- Grenze für Zeilenlänge konfigurierbar machen
- sinnvoller Vorgabewert
- Zeilenzahl soll weiterverarbeitet werden können
- Fehlerbehandlung

- Lösung:

```
def count_lines(filename, max_len=80):  
    i = 0  
    for line in open(filename):  
        if line == '\n':  
            continue  
  
        i += 1  
        if len(line) > max_len:  
            print('Zeile zu lang')  
            break  
  
        print(line, end='')  
    else:  
        return i
```

- Wächter-Muster: kein else-Zweig für nichtleere Zeilen nötig

- Lösung:

```
def count_lines(filename, max_len=80):  
    i = 0  
    for line in open(filename):  
        if line == '\n':  
            continue  
  
        i += 1  
        if len(line) > max_len:  
            print('Zeile zu lang')  
            break  
  
        print(line, end='')  
    else:  
        return i
```

- Wächter-Muster: kein else-Zweig für nichtleere Zeilen nötig

Übung: Funktionen

- Alternative Lösung:

```
def count_lines(filename, max_len=80):  
    i = 0  
    for line in open(filename):  
        line = line.strip('\r\n')  
        if line == '':  
            continue  
  
        i += 1  
        if len(line) > max_len:  
            print('Zeile zu lang')  
            break  
  
        print(line)  
    else:  
        return i
```

- String-Methode strip(), entfernt alle angegebenen Zeichen vom Rand

Übung: Funktionen

- Alternative Lösung:

```
def count_lines(filename, max_len=80):  
    i = 0  
    for line in open(filename):  
        line = line.strip('\r\n')  
        if line == '':  
            continue  
  
        i += 1  
        if len(line) > max_len:  
            print('Zeile zu lang')  
            break  
  
        print(line)  
    else:  
        return i
```

- String-Methode strip(), entfernt alle angegebenen Zeichen vom Rand

- ohne Fehler:

```
>>> x = count_lines('zeilen.txt')
eins
eins zwei
eins zwei drei
>>> x
3
```

- Fehler provozieren:

```
>>> x = count_lines('zeilen.txt', max_len=10)
eins
eins zwei
Zeile zu lang
>>> x
>>> print(x)
None
```

- ohne Fehler:

```
>>> x = count_lines('zeilen.txt')
eins
eins zwei
eins zwei drei
>>> x
3
```

- Fehler provozieren:

```
>>> x = count_lines('zeilen.txt', max_len=10)
eins
eins zwei
Zeile zu lang
>>> x
>>> print(x)
None
```

Alternative: Fehlermeldung mit Exception

- Fehler im Programm erzeugen

```
def count_lines(filename, max_len=80):  
    i = 0  
    for line in open(filename):  
        if line == '\n':  
            continue  
  
        i += 1  
        if len(line) > max_len:  
            raise ValueError('Zeile zu lang')  
  
        print(line, end='')  
  
    return i
```

Alternative: Fehlermeldung mit Exception

- Interpreter stellt Fehler fest und gibt ihn aus

```
>>> x = count_lines('zeilen.txt', max_len=10)
eins
eins zwei
Traceback (most recent call last):
  File "<pyshell#160>", line 1, in <module>
    x = count_lines('zeilen.txt', max_len=10)
  File ".../foo.py", line 9, in count_lines
    raise ValueError('Zeile zu lang')
ValueError: Zeile zu lang
```

- Fehlermeldung bezieht sich jetzt auf gespeicherte Datei
- Ausführung wird nach dem Fehler abgebrochen

Alternative: Fehlermeldung mit Exception

- Interpreter stellt Fehler fest und gibt ihn aus

```
>>> x = count_lines('zeilen.txt', max_len=10)
eins
eins zwei
Traceback (most recent call last):
  File "<pyshell#160>", line 1, in <module>
    x = count_lines('zeilen.txt', max_len=10)
  File ".../foo.py", line 9, in count_lines
    raise ValueError('Zeile zu lang')
ValueError: Zeile zu lang
```

- Fehlermeldung bezieht sich jetzt auf gespeicherte Datei
- Ausführung wird nach dem Fehler abgebrochen

Alternative: Fehlermeldung mit Exception

- Interpreter stellt Fehler fest und gibt ihn aus

```
>>> x = count_lines('zeilen.txt', max_len=10)
eins
eins zwei
Traceback (most recent call last):
  File "<pyshell#160>", line 1, in <module>
    x = count_lines('zeilen.txt', max_len=10)
  File ".../foo.py", line 9, in count_lines
    raise ValueError('Zeile zu lang')
ValueError: Zeile zu lang
```

- Fehlermeldung bezieht sich jetzt auf gespeicherte Datei
- Ausführung wird nach dem Fehler abgebrochen

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiojekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try, except, else, finally`

- Aufgabe:
 - ganze Zahl aus erster Zeile einer Datei lesen
 - den Kehrwert bilden, Division durch 0 abfangen
 - bei Erfolg den Kehrwert ausgeben
 - in jedem Fall Datei schließen
- `open()` liefert ein Dateiobjekt
- Methoden `readline()` und `close()`
- Zeichenkette mit `int()` in ganze Zahl umwandeln
- Division durch 0 wirft `ZeroDivisionError`
- `try`, `except`, `else`, `finally`

Fehler behandeln

- Lösung:

```
f = open('foo.txt')
try:
    line = f.readline()
    number = int(line)
    reciprocal = 1/number
except ZeroDivisionError:
    print('Division durch 0')
else:
    print(reciprocal)
finally:
    f.close()
    print('Datei geschlossen')
print('erfolgreich beendet')
```

Fehler behandeln

- Erfolg: else-Zweig, Programm läuft weiter
- erwarteter Fehler: except-Zweig, Programm läuft weiter
- unerwarteter Fehler: try-Block bricht am Fehler ab, Programm kann nicht weitergeführt werden
- finally-Block wird immer ausgeführt, auch bei Abbruch

Fehler behandeln

- Erfolg: else-Zweig, Programm läuft weiter
- erwarteter Fehler: except-Zweig, Programm läuft weiter
- unerwarteter Fehler: try-Block bricht am Fehler ab, Programm kann nicht weitergeführt werden
- finally-Block wird immer ausgeführt, auch bei Abbruch

Fehler behandeln

- Erfolg: else-Zweig, Programm läuft weiter
- erwarteter Fehler: except-Zweig, Programm läuft weiter
- unerwarteter Fehler: try-Block bricht am Fehler ab, Programm kann nicht weitergeführt werden
- finally-Block wird immer ausgeführt, auch bei Abbruch

- Erfolg: else-Zweig, Programm läuft weiter
- erwarteter Fehler: except-Zweig, Programm läuft weiter
- unerwarteter Fehler: try-Block bricht am Fehler ab, Programm kann nicht weitergeführt werden
- finally-Block wird immer ausgeführt, auch bei Abbruch

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 **Datentypen**
 - **Einfache Typen**
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Zahlen

- ganze Zahlen gesehen: int, str

- rationale Zahlen

```
>>> float('2.7e10')  
27000000000.0
```

- viele mathematische Funktionen in der Standardbibliothek

```
>>> import math  
>>> math.sqrt(4)  
2.0
```

- Vergleiche, Verkettung

```
>>> 0 == 0 < 4 != 3 > 2  
True
```

- numerische Fehler vermeiden: Festkommazahlen, Brüche

```
>>> from fractions import Fraction  
>>> Fraction(2, 6)  
Fraction(1, 3)
```

Zahlen

- ganze Zahlen gesehen: int, str
- rationale Zahlen

```
>>> float('2.7e10')  
27000000000.0
```

- viele mathematische Funktionen in der Standardbibliothek

```
>>> import math  
>>> math.sqrt(4)  
2.0
```

- Vergleiche, Verkettung

```
>>> 0 == 0 < 4 != 3 > 2  
True
```

- numerische Fehler vermeiden: Festkommazahlen, Brüche

```
>>> from fractions import Fraction  
>>> Fraction(2, 6)  
Fraction(1, 3)
```

Zahlen

- ganze Zahlen gesehen: int, str
- rationale Zahlen

```
>>> float('2.7e10')
27000000000.0
```

- viele mathematische Funktionen in der Standardbibliothek

```
>>> import math
>>> math.sqrt(4)
2.0
```

- Vergleiche, Verkettung

```
>>> 0 == 0 < 4 != 3 > 2
True
```

- numerische Fehler vermeiden: Festkommazahlen, Brüche

```
>>> from fractions import Fraction
>>> Fraction(2, 6)
Fraction(1, 3)
```

- ganze Zahlen gesehen: int, str
- rationale Zahlen

```
>>> float('2.7e10')  
27000000000.0
```

- viele mathematische Funktionen in der Standardbibliothek

```
>>> import math  
>>> math.sqrt(4)  
2.0
```

- Vergleiche, Verkettung

```
>>> 0 == 0 < 4 != 3 > 2  
True
```

- numerische Fehler vermeiden: Festkommazahlen, Brüche

```
>>> from fractions import Fraction  
>>> Fraction(2, 6)  
Fraction(1, 3)
```

- ganze Zahlen gesehen: int, str
- rationale Zahlen

```
>>> float('2.7e10')  
27000000000.0
```

- viele mathematische Funktionen in der Standardbibliothek

```
>>> import math  
>>> math.sqrt(4)  
2.0
```

- Vergleiche, Verkettung

```
>>> 0 == 0 < 4 != 3 > 2  
True
```

- numerische Fehler vermeiden: Festkommazahlen, Brüche

```
>>> from fractions import Fraction  
>>> Fraction(2, 6)  
Fraction(1, 3)
```

- komplexe Zahlen eingebaut

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

- komplexe mathematische Funktionen getrennt verfügbar

```
>>> math.sqrt(-1)  
Traceback (most recent call last):  
  ...  
ValueError: math domain error  
  
>>> import cmath  
>>> cmath.sqrt(-1)  
1j
```

- komplexe Zahlen eingebaut

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

- komplexe mathematische Funktionen getrennt verfügbar

```
>>> math.sqrt(-1)
Traceback (most recent call last):
...
ValueError: math domain error
```

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Wahrheitswerte

- Literale gesehen: True, False
- logische Verknüpfungen: and, or, not

```
>>> True and not False
True
```
- alle Objekte haben einen Wahrheitswert

```
>>> bool(1)
True
>>> bool('')
False
>>> bool('PyCon DE 2012')
True
>>> bool(print)
True
>>> bool(None)
False
```
- 0 oder „leer“ bedeutet False, sonst True

Wahrheitswerte

- Literale gesehen: True, False
- logische Verknüpfungen: and, or, not

```
>>> True and not False
```

```
True
```

- alle Objekte haben einen Wahrheitswert

```
>>> bool(1)
```

```
True
```

```
>>> bool('')
```

```
False
```

```
>>> bool('PyCon DE 2012')
```

```
True
```

```
>>> bool(print)
```

```
True
```

```
>>> bool(None)
```

```
False
```

- 0 oder „leer“ bedeutet False, sonst True

Wahrheitswerte

- Literale gesehen: True, False
- logische Verknüpfungen: and, or, not

```
>>> True and not False
```

```
True
```

- alle Objekte haben einen Wahrheitswert

```
>>> bool(1)
```

```
True
```

```
>>> bool('')
```

```
False
```

```
>>> bool('PyCon DE 2012')
```

```
True
```

```
>>> bool(print)
```

```
True
```

```
>>> bool(None)
```

```
False
```

- 0 oder „leer“ bedeutet False, sonst True

Wahrheitswerte

- Literale gesehen: True, False
- logische Verknüpfungen: and, or, not

```
>>> True and not False
```

```
True
```

- alle Objekte haben einen Wahrheitswert

```
>>> bool(1)
```

```
True
```

```
>>> bool('')
```

```
False
```

```
>>> bool('PyCon DE 2012')
```

```
True
```

```
>>> bool(print)
```

```
True
```

```
>>> bool(None)
```

```
False
```

- 0 oder „leer“ bedeutet False, sonst True

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - **Sequenzen: Zeichenketten, Listen, Tupel**
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'  
'PyCon DE 2001'
```

```
>>> 'PyCon DE' + ' ' + str(2012)  
'PyCon DE 2012'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2012)  
'PyCon DE 2012'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2012'])  
'PyCon DE 2012'
```

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'  
'PyCon DE 2001'
```

```
>>> 'PyCon DE' + ' ' + str(2012)  
'PyCon DE 2012'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2012)  
'PyCon DE 2012'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2012'])  
'PyCon DE 2012'
```

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'  
'PyCon DE 2001'
```

```
>>> 'PyCon DE' + ' ' + str(2012)  
'PyCon DE 2012'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2012)  
'PyCon DE 2012'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2012'])  
'PyCon DE 2012'
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2012')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2012'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2012
=====
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2012'
True
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2012')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2012'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2012
=====
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2012'
True
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2012')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2012'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2012
=====
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2012'
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2012'.split()
['PyCon', 'DE', '2012']
>>> 'PyCon-DE-2012'.split('-')
['PyCon', 'DE', '2012']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2012'.index('DE')
6
>>> 'PyCon DE 2012'.count('1')
2
```

- Abfragen: isnumeric(), istitle(), startswith()

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2012'.startswith('Py')
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2012'.split()
['PyCon', 'DE', '2012']
>>> 'PyCon-DE-2012'.split('-')
['PyCon', 'DE', '2012']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2012'.index('DE')
6
>>> 'PyCon DE 2012'.count('1')
2
```

- Abfragen: isnumeric(), istitle(), startswith()

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2012'.startswith('Py')
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2012'.split()
['PyCon', 'DE', '2012']
>>> 'PyCon-DE-2012'.split('-')
['PyCon', 'DE', '2012']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2012'.index('DE')
6
>>> 'PyCon DE 2012'.count('1')
2
```

- Abfragen: isnumeric(), istitle(), startswith()

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2012'.startswith('Py')
True
```

Zeichenketten-Methoden

- Manipulationen von Leerraum und Großschreibung, Ersetzungen

```
>>> '  PyCon DE 2012  '.strip()
'PyCon DE 2012'
>>> '  pycon de 2012  '.title()
'  Pycon De 2012  '
>>> '  PyCon DE 2012  '.replace('11', '12')
'  PyCon DE 2012  '
```

- Zeichenketten unveränderbar, Manipulationen arbeiten mit Kopien

```
>>> x = '  PyCon DE 2012  '
>>> x.strip()
'PyCon DE 2012'
>>> x
'  PyCon DE 2012  '
```

Zeichenketten-Methoden

- Manipulationen von Leerraum und Großschreibung, Ersetzungen

```
>>> '  PyCon DE 2012  '.strip()
'PyCon DE 2012'
>>> '  pycon de 2012  '.title()
'  Pycon De 2012  '
>>> '  PyCon DE 2012  '.replace('11', '12')
'  PyCon DE 2012  '
```

- Zeichenketten unveränderbar, Manipulationen arbeiten mit Kopien

```
>>> x = '  PyCon DE 2012  '
>>> x.strip()
'PyCon DE 2012'
>>> x
'  PyCon DE 2012  '
```

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2012' [2]
```

```
'C'
```

```
>>> 'PyCon DE 2012' [20]
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2012' [-4]
```

```
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2012' [0:5]
```

```
'PyCon'
```

```
>>> 'PyCon DE 2012' [-4:]
```

```
'2012'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2012'[2]
```

```
'C'
```

```
>>> 'PyCon DE 2012'[20]
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2012'[-4]
```

```
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2012'[0:5]
```

```
'PyCon'
```

```
>>> 'PyCon DE 2012'[-4:]
```

```
'2012'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2012' [2]
```

```
'C'
```

```
>>> 'PyCon DE 2012' [20]
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2012' [-4]
```

```
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2012' [0:5]
```

```
'PyCon'
```

```
>>> 'PyCon DE 2012' [-4:]
```

```
'2012'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2012' [2]
```

```
'C'
```

```
>>> 'PyCon DE 2012' [20]
```

```
Traceback (most recent call last):
```

```
...
```

```
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2012' [-4]
```

```
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2012' [0:5]
```

```
'PyCon'
```

```
>>> 'PyCon DE 2012' [-4:]
```

```
'2012'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

- Literal benutzt [], unterschiedliche Elementtypen erlaubt

```
>>> [1, 'abc', [1, 2, 3]]  
[1, 'abc', [1, 2, 3]]
```

- mit list() aus iterierbarer Sammlung erzeugen

```
>>> list(range(2, 6))  
[2, 3, 4, 5]
```

- Folgenoperationen analog zu Zeichenketten: len(), Elementzugriff und Slicing

```
>>> ['a', 'b', 'c'][1:]  
['b', 'c']
```

- ähnlich wie bei Zeichenketten, aber auf einzelne Elemente bezogen: in, Methoden wie index() und count()

```
>>> ['a', 'b'] in ['a', 'b', 'c']  
False  
>>> ['a', 'b', 'c'].index('b')  
1
```

- Literal benutzt [], unterschiedliche Elementtypen erlaubt

```
>>> [1, 'abc', [1, 2, 3]]  
[1, 'abc', [1, 2, 3]]
```

- mit list() aus iterierbarer Sammlung erzeugen

```
>>> list(range(2, 6))  
[2, 3, 4, 5]
```

- Folgenoperationen analog zu Zeichenketten: len(), Elementzugriff und Slicing

```
>>> ['a', 'b', 'c'][1:]  
['b', 'c']
```

- ähnlich wie bei Zeichenketten, aber auf einzelne Elemente bezogen: in, Methoden wie index() und count()

```
>>> ['a', 'b'] in ['a', 'b', 'c']  
False  
>>> ['a', 'b', 'c'].index('b')  
1
```

- Literal benutzt [], unterschiedliche Elementtypen erlaubt

```
>>> [1, 'abc', [1, 2, 3]]  
[1, 'abc', [1, 2, 3]]
```

- mit list() aus iterierbarer Sammlung erzeugen

```
>>> list(range(2, 6))  
[2, 3, 4, 5]
```

- Folgenoperationen analog zu Zeichenketten: len(), Elementzugriff und Slicing

```
>>> ['a', 'b', 'c'][1:]  
['b', 'c']
```

- ähnlich wie bei Zeichenketten, aber auf einzelne Elemente bezogen: in, Methoden wie index() und count()

```
>>> ['a', 'b'] in ['a', 'b', 'c']  
False  
>>> ['a', 'b', 'c'].index('b')  
1
```

- Literal benutzt [], unterschiedliche Elementtypen erlaubt

```
>>> [1, 'abc', [1, 2, 3]]  
[1, 'abc', [1, 2, 3]]
```

- mit list() aus iterierbarer Sammlung erzeugen

```
>>> list(range(2, 6))  
[2, 3, 4, 5]
```

- Folgenoperationen analog zu Zeichenketten: len(), Elementzugriff und Slicing

```
>>> ['a', 'b', 'c'][1:]  
['b', 'c']
```

- ähnlich wie bei Zeichenketten, aber auf einzelne Elemente bezogen: in, Methoden wie index() und count()

```
>>> ['a', 'b'] in ['a', 'b', 'c']  
False  
>>> ['a', 'b', 'c'].index('b')  
1
```

- Listen sind veränderbar: `append()`, `extend()`, `insert()`

```
>>> x = ['a', 'b', 'c']
```

```
>>> x.append('d')
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

- `del`, `remove()`, `pop()`

```
>>> del x[1:3]
```

```
>>> x
```

```
['a', 'd']
```

- `sort()`, `reverse()`, `sorted()`, `reversed()`

```
>>> x = ['b', 'd', 'c', 'a']
```

```
>>> x.sort()
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

```
>>> sorted(['b', 'd', 'c', 'a'])
```

```
['a', 'b', 'c', 'd']
```

- Listen sind veränderbar: `append()`, `extend()`, `insert()`

```
>>> x = ['a', 'b', 'c']
```

```
>>> x.append('d')
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

- `del`, `remove()`, `pop()`

```
>>> del x[1:3]
```

```
>>> x
```

```
['a', 'd']
```

- `sort()`, `reverse()`, `sorted()`, `reversed()`

```
>>> x = ['b', 'd', 'c', 'a']
```

```
>>> x.sort()
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

```
>>> sorted(['b', 'd', 'c', 'a'])
```

```
['a', 'b', 'c', 'd']
```

- Listen sind veränderbar: `append()`, `extend()`, `insert()`

```
>>> x = ['a', 'b', 'c']
```

```
>>> x.append('d')
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

- `del`, `remove()`, `pop()`

```
>>> del x[1:3]
```

```
>>> x
```

```
['a', 'd']
```

- `sort()`, `reverse()`, `sorted()`, `reversed()`

```
>>> x = ['b', 'd', 'c', 'a']
```

```
>>> x.sort()
```

```
>>> x
```

```
['a', 'b', 'c', 'd']
```

```
>>> sorted(['b', 'd', 'c', 'a'])
```

```
['a', 'b', 'c', 'd']
```

- Literal benutzt ()

```
>>> (1, 2, ('a', 'b'))  
(1, 2, ('a', 'b'))
```

- unveränderlich, insbesondere feste Länge, daher oft für Gruppen einer bekannten Menge von Werten verwendet
- auspacken: Mehrfachzuweisung, Klammern unnötig

```
>>> x, y = 1, 2  
>>> x, y = y, x  
>>> x, y  
(2, 1)  
>>> for x, y in [(1, 2), (3, 4)]:  
        print(x*y, end=' ')  
  
2 12
```

- mehrere Werte aus Funktion zurückgeben

```
>>> return x, y, z
```

- Literal benutzt ()

```
>>> (1, 2, ('a', 'b'))  
(1, 2, ('a', 'b'))
```

- unveränderlich, insbesondere feste Länge, daher oft für Gruppen einer bekannten Menge von Werten verwendet
- auspacken: Mehrfachzuweisung, Klammern unnötig

```
>>> x, y = 1, 2  
>>> x, y = y, x  
>>> x, y  
(2, 1)  
>>> for x, y in [(1, 2), (3, 4)]:  
        print(x*y, end=' ')  
  
2 12
```

- mehrere Werte aus Funktion zurückgeben

```
>>> return x, y, z
```

- Literal benutzt ()

```
>>> (1, 2, ('a', 'b'))  
(1, 2, ('a', 'b'))
```

- unveränderlich, insbesondere feste Länge, daher oft für Gruppen einer bekannten Menge von Werten verwendet
- auspacken: Mehrfachzuweisung, Klammern unnötig

```
>>> x, y = 1, 2  
>>> x, y = y, x  
>>> x, y  
(2, 1)  
>>> for x, y in [(1, 2), (3, 4)]:  
        print(x*y, end=' ')  
  
2 12
```

- mehrere Werte aus Funktion zurückgeben

```
>>> return x, y, z
```

- Literal benutzt ()

```
>>> (1, 2, ('a', 'b'))  
(1, 2, ('a', 'b'))
```

- unveränderlich, insbesondere feste Länge, daher oft für Gruppen einer bekannten Menge von Werten verwendet
- auspacken: Mehrfachzuweisung, Klammern unnötig

```
>>> x, y = 1, 2  
>>> x, y = y, x  
>>> x, y  
(2, 1)  
>>> for x, y in [(1, 2), (3, 4)]:  
        print(x*y, end=' ')  
  
2 12
```

- mehrere Werte aus Funktion zurückgeben

```
>>> return x, y, z
```

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - **Ungeordnete Sammlungen: Dictionarys, Mengen**
 - Weitere
- 5 Klassen

Zuordnungen: Dictionaries

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2012}
{'name': 'PyCon DE', 'year': 2012}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2012)
{'name': 'PyCon DE', 'year': 2012}
```

- Dictionary-Ausdrücke

```
>>> {x: x+1 for x in range(3)}
{0: 1, 1: 2, 2: 3}
```

- kopieren mit copy(), leeren mit clear()

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2012}
```

Zuordnungen: Dictionarys

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2012}
{'name': 'PyCon DE', 'year': 2012}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2012)
{'name': 'PyCon DE', 'year': 2012}
```

- Dictionary-Ausdrücke

```
>>> {x: x+1 for x in range(3)}
{0: 1, 1: 2, 2: 3}
```

- kopieren mit copy(), leeren mit clear()

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2012}
```

Zuordnungen: Dictionaries

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2012}
{'name': 'PyCon DE', 'year': 2012}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2012)
{'name': 'PyCon DE', 'year': 2012}
```

- Dictionary-Ausdrücke

```
>>> {x: x+1 for x in range(3)}
{0: 1, 1: 2, 2: 3}
```

- kopieren mit copy(), leeren mit clear()

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2012}
```

Zuordnungen: Dictionaries

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2012}
{'name': 'PyCon DE', 'year': 2012}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2012)
{'name': 'PyCon DE', 'year': 2012}
```

- Dictionary-Ausdrücke

```
>>> {x: x+1 for x in range(3)}
{0: 1, 1: 2, 2: 3}
```

- kopieren mit copy(), leeren mit clear()

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2012}
```

- Elementzugriff

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> x['name']
'PyCon DE'
>>> x['year'] = 2012
>>> x['location'] = 'Leipzig'
>>> x
{'location': 'Leipzig', 'name': 'PyCon DE',
 'year': 2012}
```

- Dictionaries vereinigen mit update()

```
>>> x.update({'year': 2012,
              'start': '2012-10-29'})
>>> x
{'start': '2012-10-29', 'location': 'Leipzig',
 'name': 'PyCon DE', 'year': 2012}
```

- Elementzugriff

```
>>> x = {'name': 'PyCon DE', 'year': 2012}
>>> x['name']
'PyCon DE'
>>> x['year'] = 2012
>>> x['location'] = 'Leipzig'
>>> x
{'location': 'Leipzig', 'name': 'PyCon DE',
 'year': 2012}
```

- Dictionaries vereinigen mit update()

```
>>> x.update({'year': 2012,
              'start': '2012-10-29'})
>>> x
{'start': '2012-10-29', 'location': 'Leipzig',
 'name': 'PyCon DE', 'year': 2012}
```

- testen auf Vorhandensein eines Schlüssels: in

```
>>> 'location' in x
True
```

- löschen: del, pop(), popitem()

```
>>> del x['location']
>>> 'location' in x
False
```

```
>>> x.popitem()
('start', '2012-10-29')
>>> x
{'name': 'PyCon DE', 'year': 2012}
```

- testen auf Vorhandensein eines Schlüssels: in

```
>>> 'location' in x
True
```

- löschen: del, pop(), popitem()

```
>>> del x['location']
>>> 'location' in x
False
```

```
>>> x.popitem()
('start', '2012-10-29')
>>> x
{'name': 'PyCon DE', 'year': 2012}
```

- iterieren über Schlüssel, Werte und Schlüssel-Wert-Paare

```
>>> for key in x:  
    print(key, end=' ')  
start location name year
```

```
>>> for key, value in x.items():  
    print(key, value)  
start 2012-10-29  
location Leipzig  
name PyCon DE  
year 2012
```

- Literal auch mit {}

```
>>> {1, 'a', (True, False)}  
{'a', 1, (True, False)}
```

- Sonderfall: leere Menge mit set()
- manipulieren: add(), pop(), remove(), discard(), update()
- Mengenoperationen mit Operatoren

```
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}
```

```
>>> {1, 2, 3} & {3, 4}  
{3}
```

```
>>> {1, 2, 3} - {3, 4}  
{1, 2}
```

- Ordnung anhand von Teilmengenbeziehungen

```
>>> {1, 2} < {1, 2, 3, 4}  
True
```

```
>>> {1, 2} < {3, 4}  
False
```

- Literal auch mit {}

```
>>> {1, 'a', (True, False)}  
{'a', 1, (True, False)}
```

- Sonderfall: leere Menge mit set()

- manipulieren: add(), pop(), remove(), discard(), update()

- Mengenoperationen mit Operatoren

```
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}
```

```
>>> {1, 2, 3} & {3, 4}  
{3}
```

```
>>> {1, 2, 3} - {3, 4}  
{1, 2}
```

- Ordnung anhand von Teilmengenbeziehungen

```
>>> {1, 2} < {1, 2, 3, 4}  
True
```

```
>>> {1, 2} < {3, 4}  
False
```

- Literal auch mit {}

```
>>> {1, 'a', (True, False)}  
{'a', 1, (True, False)}
```

- Sonderfall: leere Menge mit set()
- manipulieren: add(), pop(), remove(), discard(), update()
- Mengenoperationen mit Operatoren

```
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}
```

```
>>> {1, 2, 3} & {3, 4}  
{3}
```

```
>>> {1, 2, 3} - {3, 4}  
{1, 2}
```

- Ordnung anhand von Teilmengenbeziehungen

```
>>> {1, 2} < {1, 2, 3, 4}  
True
```

```
>>> {1, 2} < {3, 4}  
False
```

- Literal auch mit {}

```
>>> {1, 'a', (True, False)}  
{'a', 1, (True, False)}
```

- Sonderfall: leere Menge mit set()
- manipulieren: add(), pop(), remove(), discard(), update()
- Mengenoperationen mit Operatoren

```
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}
```

```
>>> {1, 2, 3} & {3, 4}  
{3}
```

```
>>> {1, 2, 3} - {3, 4}  
{1, 2}
```

- Ordnung anhand von Teilmengenbeziehungen

```
>>> {1, 2} < {1, 2, 3, 4}  
True
```

```
>>> {1, 2} < {3, 4}  
False
```

- Literal auch mit {}

```
>>> {1, 'a', (True, False)}  
{'a', 1, (True, False)}
```

- Sonderfall: leere Menge mit set()
- manipulieren: add(), pop(), remove(), discard(), update()
- Mengenoperationen mit Operatoren

```
>>> {1, 2, 3} | {3, 4}  
{1, 2, 3, 4}
```

```
>>> {1, 2, 3} & {3, 4}  
{3}
```

```
>>> {1, 2, 3} - {3, 4}  
{1, 2}
```

- Ordnung anhand von Teilmengenbeziehungen

```
>>> {1, 2} < {1, 2, 3, 4}  
True
```

```
>>> {1, 2} < {3, 4}  
False
```

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

Dateiartige Objekte

- bereits gesehen: öffnen, iterieren
- read, readline, readlines
- öffnen zum Schreiben und Anhängen
- seek, tell, write
- close

Dateiartige Objekte

- bereits gesehen: öffnen, iterieren
- read, readline, readlines
- öffnen zum Schreiben und Anhängen
- seek, tell, write
- close

Dateiartige Objekte

- bereits gesehen: öffnen, iterieren
- read, readline, readlines
- öffnen zum Schreiben und Anhängen
- seek, tell, write
- close

Dateiartige Objekte

- bereits gesehen: öffnen, iterieren
- read, readline, readlines
- öffnen zum Schreiben und Anhängen
- seek, tell, write
- close

Dateiartige Objekte

- bereits gesehen: öffnen, iterieren
- read, readline, readlines
- öffnen zum Schreiben und Anhängen
- seek, tell, write
- close

- Fehler (Exceptions): Hierarchie
- Funktionen, Methoden: herumreichen
- Klassen: dir

Ebenfalls Datentypen

- Fehler (Exceptions): Hierarchie
- Funktionen, Methoden: herumreichen
- Klassen: dir

Ebenfalls Datentypen

- Fehler (Exceptions): Hierarchie
- Funktionen, Methoden: herumreichen
- Klassen: dir

- 1 Einleitung
- 2 Hallo Welt
 - Die Entwicklungsumgebung `idle`
 - Python-Programme
- 3 Grundzüge der Sprache Python
 - Python im Vergleich
 - Grundlegende Programmkonstrukte
- 4 Datentypen
 - Einfache Typen
 - Sequenzen: Zeichenketten, Listen, Tupel
 - Ungeordnete Sammlungen: Dictionaries, Mengen
 - Weitere
- 5 Klassen

- einfaches Beispiel:

```
class Example(object):  
    """A simple example."""  
  
    def __init__(self, value):  
        self.value = value  
  
    def add(self, other_number):  
        return self.value + other_number  
  
    def __str__(self):  
        return 'Example instance {}'.format(self)
```

- Methoden, self
- __init__()
- spezielle Methoden: __str__, __repr__, __len__,
 __contains__

- einfaches Beispiel:

```
class Example(object):  
    """A simple example."""  
  
    def __init__(self, value):  
        self.value = value  
  
    def add(self, other_number):  
        return self.value + other_number  
  
    def __str__(self):  
        return 'Example instance {}'.format(self)
```

- Methoden, self

- `__init__()`

- spezielle Methoden: `__str__`, `__repr__`, `__len__`,
`__contains__`

- einfaches Beispiel:

```
class Example(object):  
    """A simple example."""  
  
    def __init__(self, value):  
        self.value = value  
  
    def add(self, other_number):  
        return self.value + other_number  
  
    def __str__(self):  
        return 'Example instance {}'.format(self)
```

- Methoden, self
- __init__()
- spezielle Methoden: __str__, __repr__, __len__,
 __contains__

- einfaches Beispiel:

```
class Example(object):  
    """A simple example."""  
  
    def __init__(self, value):  
        self.value = value  
  
    def add(self, other_number):  
        return self.value + other_number  
  
    def __str__(self):  
        return 'Example instance {}'.format(self)
```

- Methoden, self
- __init__()
- spezielle Methoden: __str__, __repr__, __len__,
 __contains__

- mit Zahl als Wert benutzen

```
>>> e = Example(3)
>>> str(e)
'Example instance 3'
>>> e.add(1)
4
```

- mit Liste als Wert benutzen

```
>>> e = Example([1,2,3])
>>> str(e)
'Example instance [1, 2, 3]'
>>> e.add([4, 5])
[1, 2, 3, 4, 5]
```

- mit Zahl als Wert benutzen

```
>>> e = Example(3)
>>> str(e)
'Example instance 3'
>>> e.add(1)
4
```

- mit Liste als Wert benutzen

```
>>> e = Example([1,2,3])
>>> str(e)
'Example instance [1, 2, 3]'
>>> e.add([4, 5])
[1, 2, 3, 4, 5]
```

- inkompatible Werte

```
>>> e.add(1)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#10>", line 1, in <module>  
    e.add(1)
```

```
File ".../foo.py", line 8, in add
```

```
    return self.value + other_number
```

```
TypeError: can only concatenate list  
            (not "int") to list
```