Python für Einsteiger

Tutorium über drei Stunden

Thomas Lotze

gocept gmbh & co. kg
Halle (Saale)

4.10.2011, PyCon DE, Leipzig

Gliederung

Gliederung

Gliederung

Gliederung

Gliederung

Gliederung

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Einleitung

- 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?

Gliederung

Gliederung

Der Python-Interpreter

- starten, z.B.

```
$ .../Python-3.2/bin/idle3
```

- Fenster öffnet sich

```
Python 3.2.2 (default, Sep 12 2011, 06:35:27)
[GCC 4.5.2] on linux2
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-3.2/bin/idle3
```

- Fenster öffnet sich

```
Python 3.2.2 (default, Sep 12 2011, 06:35:27)
[GCC 4.5.2] on linux2
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-3.2/bin/idle3
```

- Fenster öffnet sich

```
Python 3.2.2 (default, Sep 12 2011, 06:35:27)
[GCC 4.5.2] on linux2
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-3.2/bin/idle3
```

- Fenster öffnet sich

```
Python 3.2.2 (default, Sep 12 2011, 06:35:27)
[GCC 4.5.2] on linux2
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
```

Fehler

- 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

Fehler

- 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

Fehler

- 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

Fehler

- 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

Gliederung

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

- Ausdruck mit print() für den Nutzer des Programms gedacht

```
>>> print(2011)
2011
>>> print('2011')
2011
```

- Objektdarstellung macht Art des Objekts deutlich

```
>>> 2011
2011
>>> '2011'
'2011'
```

- Objektdarstellung mit repr() erzeugen

```
>>> print(repr(2011))
2011
>>> print(repr('2011'))
'2011'
```

Objektdarstellung

- Ausdruck mit print() für den Nutzer des Programms gedacht

```
>>> print(2011)
2011
>>> print('2011')
2011
```

- Objektdarstellung macht Art des Objekts deutlich

```
>>> 2011
2011
>>> '2011'
'2011'
```

- Objektdarstellung mit repr() erzeugen

```
>>> print(repr(2011))
2011
>>> print(repr('2011'))
'2011'
```

Objektdarstellung

- Ausdruck mit print() für den Nutzer des Programms gedacht

```
>>> print(2011)
2011
>>> print('2011')
2011
```

- Objektdarstellung macht Art des Objekts deutlich

```
>>> 2011
2011
>>> '2011'
'2011'
```

- Objektdarstellung mit repr() erzeugen

```
>>> print(repr(2011))
2011
>>> print(repr('2011'))
'2011'
```

Eingebaute Funktionen

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

Eingebaute Funktionen

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

Eingebaute Funktionen

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

Eingebaute Funktionen

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

Gliederung

Gliederung

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

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

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

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

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

}
```

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

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

}
```

Zum Vergleich: Ruby und C

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

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

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

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

}
```

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

Python im Vergleich

- Blöcke durch Einrückung gekennzeichnet, durch : eingeleitet
- implizite Zeilenenden, kein ;
- minimalistische Syntax: keine Klammern {}, keine unnötigen Schlüsselwörter (do)
- Klammern um Funktionsargumente
- Variablen implizit deklariert
- nur eine Art von while-Schleifen
- print() kann beliebige Objekte ausdrucken
- wesentliche Funktionen schon eingebaut
- kein Grundgerüst (main)
- sinnvolle Vorgaben, z.B. impliziter Zeilenumbruch in print()

Gliederung

Schleifen

- 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, $a \leq x < b$

Schleifen

- 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, $a \leq x < b$

Schleifen

- 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, $a \leq x < b$

Schleifen

- 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, $a \leq x < b$

Schleifen

- 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

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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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

Zusammen: for-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)
```

while-Schleifen steuern

- syntaktisch analog zu for-Schleifen
- fußgesteuerte Schleife:

```
i = 10
while i < 10:
    print(i)
```

```
i = 10
while True:
    print(i)
    if i >= 10:
        break
```

while-Schleifen steuern

- syntaktisch analog zu for-Schleifen
- fußgesteuerte Schleife:

```
i = 10
while i < 10:
    print(i)
```

```
i = 10
while True:
    print(i)
    if i >= 10:
        break
```

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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- zurück zum Beispielprogramm: wiederverwendbar machen
- Aufgabe:
  - Algorithmus zum Zeilenzählen auf beliebige Dateien anwenden
  - Grenze für Zeilenlänge konfigurierbar, sinnvolle Vorgabe
  - Zeilenzahl soll weiterverarbeitet werden können
  - Fehlerbehandlung
- 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

Funktionen

- 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:
        print('nichtleere Zeilen:', i)
        return i
```

Funktionen

- ohne Fehler:

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

- Fehler provozieren:

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

Funktionen

- ohne Fehler:

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

    print('nichtleere Zeilen:', i)
    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

Fehler behandeln

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

- 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

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

Gliederung

Gliederung

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)
```

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

- 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
```

Zahlen

- 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 2011')
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 2011')
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 2011')
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 2011')
True
>>> bool(print)
True
>>> bool(None)
False
```

- 0 oder „leer“ bedeutet False, sonst True

Gliederung

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'
'PyCon DE 2001'
>>> 'PyCon DE' + ' ' + str(2011)
'PyCon DE 2011'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2011)
'PyCon DE 2011'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2011'])
'PyCon DE 2011'
```

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'
'PyCon DE 2001'
>>> 'PyCon DE' + ' ' + str(2011)
'PyCon DE 2011'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2011)
'PyCon DE 2011'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2011'])
'PyCon DE 2011'
```

Zeichenketten-Operationen

- Verkettung mit +

```
>>> 'PyCon DE' + ' ' + '2001'
'PyCon DE 2001'
>>> 'PyCon DE' + ' ' + str(2011)
'PyCon DE 2011'
```

- meist lesbarer: Formatierung

```
>>> '{} {}'.format('PyCon DE', 2011)
'PyCon DE 2011'
```

- Verkettung mit Verbindungstext

```
>>> ' '.join(['PyCon', 'DE', '2011'])
'PyCon DE 2011'
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2011')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2011'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2011
==============
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2011'
True
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2011')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2011'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2011
=============
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2011'
True
```

Zeichenketten-Operationen

- Länge bestimmen mit len()

```
>>> len('PyCon DE 2011')
13
```

- Vervielfältigung mit *

```
>>> x = 'PyCon DE 2011'
>>> y = len(x) * '='
>>> print('\n'.join([x, y]))
PyCon DE 2011
=============
```

- testen, ob eine Zeichenkette enthalten ist

```
>>> 'DE' in 'PyCon DE 2011'
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2011'.split()
['PyCon', 'DE', '2011']
>>> 'PyCon-DE-2011'.split('-')
['PyCon', 'DE', '2011']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2011'.index('DE')
6
>>> 'PyCon DE 2011'.count('1')
2
```

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

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2011'.startswith('Py')
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2011'.split()
['PyCon', 'DE', '2011']
>>> 'PyCon-DE-2011'.split('-')
['PyCon', 'DE', '2011']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2011'.index('DE')
6
>>> 'PyCon DE 2011'.count('1')
2
```

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

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2011'.startswith('Py')
True
```

Zeichenketten-Methoden

- zerlegen:

```
>>> 'PyCon DE 2011'.split()
['PyCon', 'DE', '2011']
>>> 'PyCon-DE-2011'.split('-')
['PyCon', 'DE', '2011']
```

- Teile suchen und zählen

```
>>> 'PyCon DE 2011'.index('DE')
6
>>> 'PyCon DE 2011'.count('1')
2
```

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

```
>>> '17'.isnumeric()
True
>>> 'PyCon DE 2011'.startswith('Py')
True
```

Zeichenketten-Methoden

- Manipulationen von Leerraum und Großschreibung, Ersetzungen

```
>>> '   PyCon DE 2011  '.strip()
'PyCon DE 2011'
>>> '   pycon de 2011  '.title()
'   Pycon De 2011  '
>>> '   PyCon DE 2011  '.replace('11', '12')
'   PyCon DE 2012  '
```

- Zeichenketten unveränderbar, Manipulationen arbeiten mit Kopien

```
>>> x = '   PyCon DE 2011  '
>>> x.strip()
'PyCon DE 2011'
>>> x
'   PyCon DE 2011  '
```

Zeichenketten-Methoden

- Manipulationen von Leerraum und Großschreibung, Ersetzungen

```
>>> '   PyCon DE 2011  '.strip()
'PyCon DE 2011'
>>> '   pycon de 2011  '.title()
'   Pycon De 2011  '
>>> '   PyCon DE 2011  '.replace('11', '12')
'   PyCon DE 2012  '
```

- Zeichenketten unveränderbar, Manipulationen arbeiten mit Kopien

```
>>> x = '   PyCon DE 2011  '
>>> x.strip()
'PyCon DE 2011'
>>> x
'   PyCon DE 2011  '
```

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2011'[2]
'C'
>>> 'PyCon DE 2011'[20]
Traceback (most recent call last):
  ...
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2011'[-4]
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2011'[0:5]
'PyCon'
>>> 'PyCon DE 2011'[-4:]
'2011'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2011'[2]
'C'
>>> 'PyCon DE 2011'[20]
Traceback (most recent call last):
  ...
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2011'[-4]
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2011'[0:5]
'PyCon'
>>> 'PyCon DE 2011'[-4:]
'2011'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2011'[2]
'C'
>>> 'PyCon DE 2011'[20]
Traceback (most recent call last):
  ...
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2011'[-4]
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2011'[0:5]
'PyCon'
>>> 'PyCon DE 2011'[-4:]
'2011'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Auf Teile von Zeichenketten zugreifen

- einzelne Zeichen per Index ansprechen

```
>>> 'PyCon DE 2011'[2]
'C'
>>> 'PyCon DE 2011'[20]
Traceback (most recent call last):
  ...
IndexError: string index out of range
```

- Indizieren von hinten

```
>>> 'PyCon DE 2011'[-4]
'2'
```

- Ausschnitte ansprechen (Slicing)

```
>>> 'PyCon DE 2011'[0:5]
'PyCon'
>>> 'PyCon DE 2011'[-4:]
'2011'
```

- Intervalle wieder halboffen, End-Index liegt dahinter

Listen

- 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

- 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

- 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

- 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

- 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

- 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

- 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']
```

Tupel

- 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
```

Tupel

- 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
```

Tupel

- 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
```

Tupel

- 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
```

Gliederung

Zuordnungen: Dictionarys

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2011}
{'name': 'PyCon DE', 'year': 2011}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2011)
{'name': 'PyCon DE', 'year': 2011}
```

- 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': 2011}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2011}
{'name': 'PyCon DE', 'year': 2011}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2011)
{'name': 'PyCon DE', 'year': 2011}
```

- 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': 2011}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2011}
{'name': 'PyCon DE', 'year': 2011}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2011)
{'name': 'PyCon DE', 'year': 2011}
```

- 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': 2011}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- Literal benutzt {}

```
>>> {'name': 'PyCon DE', 'year': 2011}
{'name': 'PyCon DE', 'year': 2011}
```

- Konstruktor: dict(), nimmt Schlüsselwort-Argumente

```
>>> dict(name='PyCon DE', year=2011)
{'name': 'PyCon DE', 'year': 2011}
```

- 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': 2011}
>>> y = x.copy()
>>> x.clear()
>>> x
{}
>>> y
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- Elementzugriff

```
>>> x = {'name': 'PyCon DE', 'year': 2011}
>>> x['name']
'PyCon DE'
>>> x['year'] = 2012
>>> x['location'] = 'Leipzig'
>>> x
{'location': 'Leipzig', 'name': 'PyCon DE',
 'year': 2012}
```

- Dictionarys vereinigen mit update()

```
>>> x.update({'year': 2011,
              'start': '2011-10-04'})
>>> x
{'start': '2011-10-04', 'location': 'Leipzig',
 'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- Elementzugriff

```
>>> x = {'name': 'PyCon DE', 'year': 2011}
>>> x['name']
'PyCon DE'
>>> x['year'] = 2012
>>> x['location'] = 'Leipzig'
>>> x
{'location': 'Leipzig', 'name': 'PyCon DE',
 'year': 2012}
```

- Dictionarys vereinigen mit update()

```
>>> x.update({'year': 2011,
              'start': '2011-10-04'})
>>> x
{'start': '2011-10-04', 'location': 'Leipzig',
 'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- testen auf Vorhandensein eines Schlüssels: in

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

- löschen: del, pop(), popitem()

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

>>> x.popitem()
('start', '2011-10-04')
>>> x
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- testen auf Vorhandensein eines Schlüssels: in

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

- löschen: del, pop(), popitem()

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

>>> x.popitem()
('start', '2011-10-04')
>>> x
{'name': 'PyCon DE', 'year': 2011}
```

Zuordnungen: Dictionarys

- 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 2011-10-04
location Leipzig
name PyCon DE
year 2011
```

Mengen

- 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
```

Mengen

- 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
```

Mengen

- 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
```

Mengen

- 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
```

Mengen

- 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
```

Gliederung

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

Ebenfalls Datentypen

- 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

Gliederung

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__
- Duck-Typing

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__
- Duck-Typing

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__
- Duck-Typing

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__
- Duck-Typing

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__
- Duck-Typing

Klassen

- 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]
```

Klassen

- 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]
```

Klassen

- 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
```