Teil II

Datentypen

Numerische Typen - ganze Zahlen (int)

- Rechnen, wie wir es gewohnt sind (`12*34+567`)

Numerische Typen - ganze Zahlen (int)

- Rechnen, wie wir es gewohnt sind (`12*34+567`)
- In Python ist der Zahlenbereich beliebig groß
  ⇒kein Overflow, sondern automatische Typ-Konvertierung
  ⇒Beispiel: int ⇒long

```
>>> 1234**56
129911902554871451941032084396235137754657820101 27
392384379012704624259433055094648925678485362472 90
201061395156473849109449211865238658490562753590 66
26235291168250476992921 6L
>>> num = 92233720368547758 07
>>> type(num)
<type 'int'>
>>> num = num+1
>>> type(num)
<type 'long'>
```

- Zahlengröße: eingeschränkt durch Arbeitsspeicher

Numerische Typen - float und bool

- `bool` – wahr und falsch

```
>>> print True, False
>>> True == True
>>> True != True
```

Numerische Typen - float und bool

- `bool` – wahr und falsch

```
>>> print True, False
>>> True == True
>>> True != True
```

- `float` – Gleitkommazahl (double precision)
  $f = s \cdot m \cdot 2^e$, 1+52+11 Bit

```
>>> 1.0 - 49.0*(1.0/49.0)
1.1102230246251565e-16
```

Numerische Typen - float und bool

- `bool` – wahr und falsch

```
>>> print True, False
>>> True == True
>>> True != True
```

- `float` – Gleitkommazahl (double precision)
  $f = s \cdot m \cdot 2^e$, 1+52+11 Bit

```
>>> 1.0 - 49.0*(1.0/49.0)
1.1102230246251565e-16
```

- `None` – spezieller Typ (“undefined”) (ähnlich zu `NULL` in C/C++)

Numerische Typen - Komplexe Zahlen

```
>>> 1+4j
(1+4j)
>>> c = complex(2,3)
>>> c
(2+3j)
>>> d = 2*(c**3 - 3j)
>>> d
(-92+12j)
>>> d.real
-92.0
>>> d.imag
12.0
```

- Imaginärteil hat Zusatz `j`
- Basisoperationen normal anwendbar
- Für komplexere Operationen (z.B. sin) spezielle Bibliotheken

Allgemeine Hinweise

- Alles in Python ist ein Objekt! (Attribute und Methoden)

Allgemeine Hinweise

- Alles in Python ist ein Objekt! (Attribute und Methoden)
- Zugriff auf Objektmethoden: `<Objektname>.<Methodenname>()`
- Zugriff auf Objektattribute: `<Objektname>.<Attributname>`

```
c = complex(2,3)
print c.real, c.imag
```

Allgemeine Hinweise

- Alles in Python ist ein Objekt! (Attribute und Methoden)
- Zugriff auf Objektmethoden: `<Objektname>.<Methodenname>()`
- Zugriff auf Objektattribute: `<Objektname>.<Attributname>`

```
c = complex(2,3)
print c.real, c.imag
```

- Dynamische Typisierung: Typ von Variablen durch Zuweisung fix
- Automatische Konvertierung wo nötig und möglich

Typ-Konvertierung

Hands-On: Welche Typen erhält man in folgender Anweisung?

```
a = 1234
type(a)
b = 100
type(b)
a = 1234**100
type(a)
b = b*1.0
type(b)
```

Numerische Typen - Arithmetische Operationen

- Klassische Operationen

```
x + y      # Addition
x - y      # Subtraktion
x * y      # Multiplikation
x / y      # Division
x // y     # Truncated division (integer division)
x ** y     # Exponentiation
x % y      # Modulo
x -= 2; x *= 4; ...
```

Numerische Typen - Arithmetische Operationen

- Klassische Operationen

```
x + y      # Addition
x - y      # Subtraktion
x * y      # Multiplikation
x / y      # Division
x // y     # Truncated division (integer division)
x ** y     # Exponentiation
x % y      # Modulo
x -= 2; x *= 4; ...
```

- Achtung: Division verschieden für `int` und `float` (Python $<$ 3.0)

```
7/4
7.0/4
```

Numerische Typen - Vergleichsoperatoren

- `a == b` gibt `True` bei Gleichheit (= dient Zuweisung)

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

Numerische Typen - Vergleichsoperatoren

- `a == b` gibt `True` bei Gleichheit (= dient Zuweisung)

```
>>> 1 == 2
False
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

- weitere Operatoren: `!=`, `<`, `>`, `<=`, `>=`
- Verknüpfung mit `and` und `or`, Negation mit `not`

```
>>> not(1==2 or 2**2 == 1+3)
False
```

Frage

- **Hands-On:** Wie weit können Sie mit Ihren Fingern zählen?

Bitweise Operatoren

- **Hands-On:** Wie weit können Sie mit Ihren Fingern zählen?
- Bitweise Darstellung
  $1 \Rightarrow 0*4+0*2+1*1 \Rightarrow 001$
  $3 \Rightarrow 0*4+1*2+1*1 \Rightarrow 011$
  $4 \Rightarrow 1*4+0*2+0*1 \Rightarrow 100$
  ...

- **Hands-On:** Wie weit können Sie mit Ihren Fingern zählen?
- Bitweise Darstellung
  $1 \Rightarrow 0*4+0*2+1*1 \Rightarrow 001$
  $3 \Rightarrow 0*4+1*2+1*1 \Rightarrow 011$
  $4 \Rightarrow 1*4+0*2+0*1 \Rightarrow 100$
  ...
- Shifting: `<<`, `>>`

```
>>> 3<<2
12
```

- **Hands-On:** Wie weit können Sie mit Ihren Fingern zählen?
- Bitweise Darstellung
  1 $\Rightarrow$ 0*4+0*2+1*1 $\Rightarrow$ 001
  3 $\Rightarrow$ 0*4+1*2+1*1 $\Rightarrow$ 011
  4 $\Rightarrow$ 1*4+0*2+0*1 $\Rightarrow$ 100
  ...
- Shifting: `<<`, `>>`

```
>>> 3<<2
12
```

- Bitweise Operatoren: `&` (bitwise and), `|` (bitwise or), `^`(bitwise exclusive or), `~`(complement)

```
>>> 3&1
1
>>> 4|3
7
```

Mathematische Funktionen: Modul math

- Ein Modul ist eine Sammlung von Funktionalitäten
- Einbinden mit `import`
- `math` enthält mathematische Standardfunktionen
- `sin, cos, sqrt, exp, log, pow, ...`
- Zusätzlich Konstanten `pi` und `e`
- Funktioniert nur mit integer und float
- Für komplexe Zahlen: `cmath`

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

Mathematische Funktionen: Modul math

- Ein Modul ist eine Sammlung von Funktionalitäten
- Einbinden mit `import`
- `math` enthält mathematische Standardfunktionen
- `sin, cos, sqrt, exp, log, pow, ...`
- Zusätzlich Konstanten `pi` und `e`
- Funktioniert nur mit integer und float
- Für komplexe Zahlen: `cmath`

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
```

```
>>> from math import sin
>>> sin(1)
0.8414709848078965
```

Sequenzen - Strings

- Sequenz von Zeichen (character)
- einzelne oder doppelte Anführungszeichen
- dreifache Anführungszeichen für mehrzeiligen String

```
"...␣there␣lived␣a␣hobbit."
"a␣'hobbit'"
'a␣"hobbit"'
'a␣\'hobbit\''
"""In a hole in the ground
there "lived" a 'hobbit'"""
```

Sequenzen - Strings

- Sequenz von Zeichen (character)
- einzelne oder doppelte Anführungszeichen
- dreifache Anführungszeichen für mehrzeiligen String

```
"...␣there␣lived␣a␣hobbit."
"a␣'hobbit'"
'a␣"hobbit"'
'a␣\'hobbit\''
"""In a hole in the ground
there "lived" a 'hobbit'"""
```

- Verknüpfung von Strings

```
s1 = "In␣a␣hole␣in␣the␣ground"
s2 = "there␣lived␣a␣hobbit"
s = s1 + '␣' + s2
print s
```

Sequenzen - Strings (2)

- Replikation

```
>>> "Hi!␣"*3
'Hi!␣Hi!␣Hi!␣'
```

Sequenzen - Strings (2)

- Replikation

```
>>> "Hi!␣"*3
'Hi!␣Hi!␣Hi!␣'
```

- Indizierung und Abschnitte

```
>>> s = "12345678"
>>> s[3], s[2:4], s[-2]
('4', '34', '7')
>>> s[6:], s[:3]
('78', '123')
>>> s[0:-1:2]  # von 0 bis (ausschl.) letztem
               # Element, Schrittweite 2
'1357'
>>> s[2] = 4   # Fehler: Strings sind unveränderbar
>>> len(s)
8
```

Ausgewählte String-Methoden

```
s = "␣␣Frodo␣and␣␣Sam␣and␣␣␣Bilbo"
s.islower()  # true, falls s komplett klein geschr.
s.isupper()
s.startswith("Frodo") # s.startswith("Frodo", 2)
s.endswith("Bilbo")
s = s.strip()#entfernt Leerzeichen vorne/hinten
s.upper() # konvertiert in Gross-Buchst.
s = s.lower()
s = s.center(len(s)+4)# zentrieren
s.lstrip()
s.rstrip("␣")
s = s.strip()
s.find("sam")
s.rfind("and")          # use help(str.rfind)
```

Ausgabe verschiedener Datentypen

print

```
>>> a=3; b=4.5; c='text'
>>> print a, b, c
3 4.5 text
```

Ausgabe verschiedener Datentypen

print

```
>>> a=3; b=4.5; c='text'
>>> print a, b, c
3 4.5 text
```

Stringinterpolation

- Stringinterpolation gibt hohe Flexibilität
- Platzhalter mit % in einem String
- Nach dem String folgen ein % und ein Tupel mit Werten.

```
>>> print "int: %d, float: %f und string: %s" \
          % (4, 3.5, "bla")
int: 4, float: 3.500000 und string: bla
```

Konvertierungszeichen zur Stringinterpolation

- `%d` Dezimal Integer
- `%f` Gleitpunkt (float)
- `%e`,`%E` Gleitpunkt wissenschaftlich (m.ddde+xx)
- `%g`,`%G` kompakteste Gleitpunktdarstellung
- `%s` String
- `%c` einzelnes Zeichen
- `%o` Oktaldarstellung eines Integers
- `%x`, `%X` Hexadezimaldarstellung eines Integers
- `%%` Das Zeichen `%`

```
>>> a = 7.357
>>> print "%f, %.2f, %8.1f" % (a, a, a)
7.357000, 7.36,      7.4
```

Sequenzen - Tupel

Sequenzen - Tupel

- Sequenzen beliebiger Objekte
- Unveränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> t = (1, "zwei", (3,4,5))
>>> t[0]
1
>>> t[-1]
(3,4,5)
>>> t[1] = 2   # error
>>> len(t)
3
>>> t + (6,7)
(1, 'zwei', (3, 4, 5), 6, 7)
```

Sequenzen - Tupel

- Sequenzen beliebiger Objekte
- Unveränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> t = (1, "zwei", (3,4,5))
>>> t[0]
1
>>> t[-1]
(3,4,5)
>>> t[1] = 2  # error
>>> len(t)
3
>>> t + (6,7)
(1, 'zwei', (3, 4, 5), 6, 7)
```

- Minimum und Maximum mit `min(t)` und `max(t)`
  $\Rightarrow$eigene Erweiterung der Min./Max.-Definition möglich

Sequenzen - Listen

Sequenzen - Listen

- Sequenzen beliebiger Objekte
- Veränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> l = [1, "zwei", (3,4,5)]
>>> l[0]
1
>>> l[1] = 2; l   # funktioniert!
[1, 2, (3,4,5)]
>>> l + [6,7]
[1 , 2, (3, 4, 5), 6, 7]
```

Sequenzen - Listen

- Sequenzen beliebiger Objekte
- Veränderlich
- Indizierung und Ausschnitte wie bei Strings

```
>>> l = [1, "zwei", (3,4,5)]
>>> l[0]
1
>>> l[1] = 2; l   # funktioniert!
[1, 2, (3,4,5)]
>>> l + [6,7]
[1 , 2, (3, 4, 5), 6, 7]
```

- Operationen auf Listen

```
l = [0,1,2,3,4,5,6,7,8,9]
l[2:4] = [10, 11]
l[1:7:2] = [-1, -2, -3]
del l[::2]
```

Sequenzen - Listen (2)

- Listen sind Objekte mit Methoden:

```
>>> l = [0,1,2]
>>> l.append("x")      # [0, 1, 2, 'x']
>>> l.extend([5,6])    # [0, 1, 2, 'x', 5, 6]
>>> l.insert(2, "x")   # [0, 1, 'x', 2, 'x', 5, 6]
>>> l.count("x")       # 2
>>> l.sort()           # [0, 1, 2, 5, 6, 'x', 'x']
>>> l.reverse()        # ['x', 'x', 6, 5, 2, 1, 0]
>>> l.remove("x")      # ['x', 6, 5, 2, 1, 0]
>>> l.pop()            # ['x', 6, 5, 2, 1]
0
```

Sequenzen - Listen (2)

- Listen sind Objekte mit Methoden:

```
>>> l = [0,1,2]
>>> l.append("x")      # [0, 1, 2, 'x']
>>> l.extend([5,6])    # [0, 1, 2, 'x', 5, 6]
>>> l.insert(2, "x")   # [0, 1, 'x', 2, 'x', 5, 6]
>>> l.count("x")       # 2
>>> l.sort()           # [0, 1, 2, 5, 6, 'x', 'x']
>>> l.reverse()        # ['x', 'x', 6, 5, 2, 1, 0]
>>> l.remove("x")      # ['x', 6, 5, 2, 1, 0]
>>> l.pop()            # ['x', 6, 5, 2, 1]
0
```

- Integerlisten erzeugen: `range` Funktion

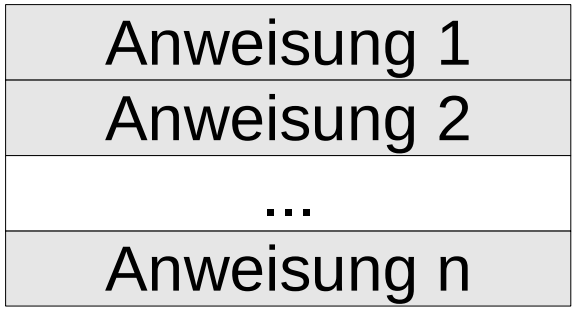
```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(-10, 5, 3)
[-10, -7, -4, -1, 2]
```

Teil III

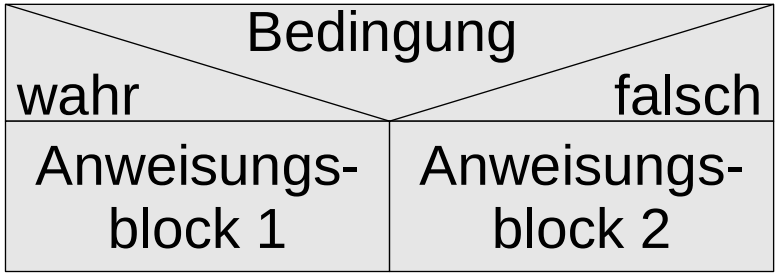
Kontrollstrukturen

Struktogramme

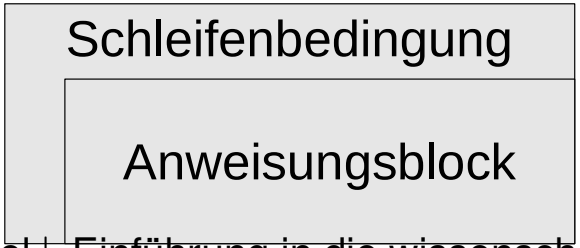
- Graphische Darstellung von Programmen
- Anweisungen



- Bedingungen



- Schleifen



Struktogramme - Beispiel

- Euklidscher Algorithmus zur Berechnung des ggT:
  Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.

  [Euklid, Die Elemente, Buch VII, §2]

Struktogramme - Beispiel

- Euklidscher Algorithmus zur Berechnung des ggT:
  Wenn CD aber AB nicht misst, und man nimmt bei AB, CD abwechselnd immer das kleinere vom größeren weg, dann muss (schließlich) eine Zahl übrig bleiben, die die vorangehende misst.

[Euklid, Die Elemente, Buch VII, §2]