Typ-Konvertierung

- Wie in C++, mit anderer Syntax:

Typ-Konvertierung	
`int(x)`	**Konvertiert x in eine Ganzzahl**
`float(x)`	**Konvertiert x in eine Fließkommazahl**
`str(x)`	**Konvertiert Objekt x in eine String-Darstellung**
`repr(x)`	**Konvertiert Objekt x in einen String-Ausdruck**
`eval(x)`	**Wertet String x aus und gibt ein Objekt zurück**
`tuple(x)`	**Konvertiert Sequenz in ein Tupel**
`list(x)`	**Konvertiert Sequenz in eine Liste**
`chr(x)`	**Konvertiert eine Ganzzahl in ein Zeichen**
`ord(x)`	**Konvertiert einzelnes Zeichen in dessen Zahlw.**

- In C++:

Schreibweise
`y = static_cast<float>(x);`

Beispiel: Zufallszahlen

```
import random
N = 1000
# random() liefert eine Zufallszahl zwischen 0 und 1
r = random.random()
s = int( r*N )
# s = ganzzahlige Zufallszahl zwischen 0 und 1000
```

Vergleichsoperatoren (wie in C++)

- Operanden sollten gleichen Typ haben
  - Sonst automatische Konvertierung
- Resultat: Wahrheitswert (in Python: False oder True)
- Achtung: verwechsle nicht = und == !
- Richtiger Vergleich von Floating-Point-Werten siehe C++-Vorlesung

Bit-Operatoren (wie in C++)

- Für integrale Typen definiert (int, ...)
- Wert wird als Bitmuster betrachtet
- Beispiele:

x	y	x & y	x | y	x ^ y	~x	~y
1010	1100	1000	1110	0110	0101	0011

- Beispiele:

```
if x & 1 :
    # x is odd
else:
    # x is even
```

```
x = x >> 2    # = x / 4
x &= 0x11     # = x % 4
x = x << 3    # = x * 8
```

"Premature optimization is the root of all evil." D. E. Knuth

Präzedenz und Assoziativität (fast wie in C++)

Präzedenz	Operator	
	`[…,…]`	`(…,…)` Liste-, Tupel-Bildung
	`[]`	`()` Indizierung, Fkt.-aufruf
	`~ + -`	unäre Operatoren
	`* / %`	binäre multiplikative Op.
	`+ -`	binäre additive Op.
	`&`	bit-weise logisches UND
	`|`	bit-weises XOR
	`< … >=`	Vergleichsoperatoren
	`== !=`	
	`in`	Bsp.: `x in [1,2,4,8,16]`
	`not`	logische Verknüpfung
	`and`	logische Verknüpfung
	`or`	
	`= += … |=`	Zuweisungsoperatoren

Short circuit logic bei Boole'schen Ausdrücken

- Wie bei C++
- **`and`** und **`or`** werden von links nach rechts ausgewertet
- Falls Wahrheitswert feststeht, keine weitere Auswertung!
  - **`True or x → True`**
  - **`False and x → False`**

Statements (Anweisungen)

- Unterschied zu C++:
  - Eine Anweisung pro Zeile (normalerweise)
  - kein Semikolon nötig am Zeilenende
  - Leere Anweisung = **`pass`**

I/O, System-Aufrufe, Argumente

- Input


Keyboard


Mouse


Storage


Network


Digital camera


3D Scanner

- Output


Display


Speakers


Storage

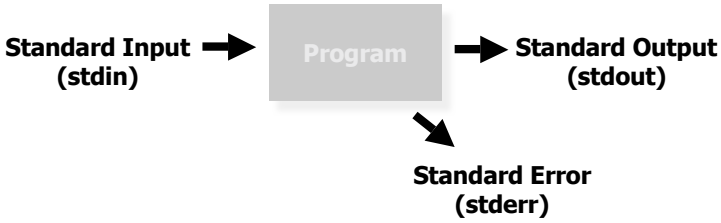

Network


Printer


MP3 Player

stdin / stdout / stderr

- Jeder Prozeß hat stdin, stdout, stderr



- Normalerweise verbunden mit Tastatur bzw. Terminalfenster
- Oder durch Redirection verbunden mit Files!

```
% program parameters < infile > outfile
```

  - `stderr` erscheint weiterhin im Terminal

Standard Output

- Flexible Schnittstelle für den Output von Programmen
- In Python wird der Output von `print` zu `stdout` geleitet
- Normalerweise wird stdout in einem Terminalfenster ausgegeben
- Man kann ihn aber auch in eine Datei umleiten, ohne Änderungen am Programm vorzunehmen

Ausgabe auf stdout

- Unformatiert:

```
print "hallo", x, 2+3
print "x:\t", x , "\ny:\t", y
```

  - in C++:

```
puts("text");
puts( zeiger-auf-string );
```

- Formatiert:

```
print "format-string" % (arg1, arg2, …)
```

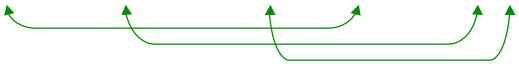
  - % ist ein Operator, der einen String und eine Liste nimmt, und wieder einen String liefert.
  - in C++:

```
printf( "format-string", arg1, arg2, … );
```


Der Format-String

- Normale Zeichen und mit `%` eingeleitete Formatierungsanweisung
- Korrespondenz zwischen `%`-Anweisung und Argumenten:

```
print "Blub %· Bla %· Blubber …" % (arg1, arg2, …)
```



  - in C++:

```
printf( "Blub %· Bla %· Blubber …", arg1, arg2, … );
```

- Häufige %-Formatangaben (wie in C++):

%-Angabe	Formatierung
`%d`	**int**
`%u`	**unsigned int**
`%f`	**float**
`%s`	**string**
`%x`	**Hexadezimal-Zahl**

Ausgabe auf stderr

- Ein beliebiges Objekt als String auf stderr ausgeben:

```
import sys
sys.stderr.write( str(x) + "\n" )
```

  - in C:

```
fputs( "text", stderr );
fputs( zeiger-auf-string, stderr );
```

- Formatiert:

```
sys.stderr.write( "format-string" % (arg1, arg2, …) )
```

  - in C++:

```
fprintf( stderr, "format-string", arg1, arg2, … );
```


Lesen von stdin

- Ganze Zeile als String

```
x = raw_input("Prompt:")
```

  - Das Prompt ist optional
  - in C++:

```
gets( x );  // x = Zeiger auf String
```

- Einzelne Zahl

```
x = input("prompt")
```

  - Klappt nur, wenn der eingegebene String eine einzelne Zahl ist
  - Genauer: Eingabe muß ein einzelner gültiger Python-Ausdruck sein
  - in C++:

```
scanf("%d", &x );
```

Komplexere Eingaben lesen

- Mehrere Zahlen in einer Zeile:

```
v1, v2, v3 = input("Geben Sie den Vektor ein: ")
```

  - Eingabe muß `1, 2, 3` sein!

- in C++:

```
printf("Geben Sie den Vektor ein: ");
float v1, v2, v3;
scanf( "%f %f %f", &v1, &v2, &v3 );
```

- Zeilen lesen bis Input leer:

```
import sys
for line in sys.stdin :
    # do something with line
```


- Von Tastatur einlesen:

```
% ./program
Geben Sie den Vektor ein: □
```

- Mit I/O-Redirection aus File lesen:

```
% ./program < vector.txt
Geben Sie den Vektor ein:
%
```

  - Achtung: falls mit Schleife gelesen wird, muß man manchmal 2x Ctrl-D drücken zum Beenden der Eingabe (Bug in Python?)

Kommando-Zeilen-Argumente

- Bei Aufruf der Form

```
% ./program arg1 arg2 arg3
```

  werden Kommandozeilenargumente im sog. *Environment* des neuen Prozesses gespeichert

- Zugriff über die `argv`-Variable:

```
import sys
for arg in sys.argv:
    print arg
```

  - Oder:

```
print argv[0], argv[1]
```

  - argv[0] enthält Name des Skriptes

Beispiel: Berechnung von Schaltjahren

```
import sys

y = int( sys.argv[1] )
isLeapYear = (y % 4 == 0) and (y % 100 != 0)
isLeapYear = isLeapYear or (y % 400 == 0)

if ( isLeapYear ):
      print y, " ist ein Schaltjahr"
else:
      print y, " ist kein Schaltjahr"
```

File-Eingabe und -ausgabe

- Bisher gesehen:
  - Ausgaben in Datei mit Redirection: program > outfile
  - Eingaben aus einer Datei mit Redirection: program < infile
- Erstellen, Schreiben und Lesen von Dateien aber auch direkt im Programm selbst möglich:

```
f = open("filename")    # Ergibt ein Datei-Objekt
line = f.readline()     # Liest die erste Zeile der Datei
while line:             # Zeilenweise Lesen der Datei
    print line
    line = f.readline()
f.close()               #Schließen der Datei
```


- Man kann auch den gesamten Inhalt direkt lesen:

```
f = open( "filename" ) # Ergibt ein Datei-Objekt
inhalt = f.read()      # Liest die gesamte Datei
f.close()              # Schließen der Datei
```

- Erzeugen und Schreiben in eine Datei:

```
f = open( "out", "w" ) # Öffne Datei zum Schreiben
i = 0;
while i < 10:
    f.write( "%d\n" % i ) # Datei-Ausgabe
    i += 1
f.close()              #Schließen der Datei
```


Beispiel

- Liste mit Daten von Studenten (Name, Matrikelnummer, Note der Informatik-Klausur) sei in File **`Studenten.dat`** gespeichert.
- Das Prüfungsbüro will eine Liste mit den Namen aller Studenten, die die Klausur bestanden haben.

Studenten.dat		
Name	**MatrNr**	**Note**
Müller	123456	1.0
Meier	348565	3.7
Mustermann	540868	5.0
…	…	…


```
import string

in = open( "Studenten.dat" )     # Ergibt ein Datei-Objekt
line = in.readline()
while line:                      # Zeilenweise Lesen der Datei
      a = string.split( line )
      name.append( a[0] )
      note.append( float(a[2]) )
      line = in.readline()
in.close()

out = open( "Bestanden.dat", "w" )
for i in range( 0, len( name ) ):
      if note[i] < 5.0:
            out.write( "%s\n" % name[i] ) # Datei-Ausgabe

out.close()
```

Die Funktion "split" teilt einen String in seine einzelnen Wörter auf

System-Aufrufe

- Problem: andere Programme von Skript aus aufrufen
- Lösung in Python: Das **os**-Modul

```
import os
os.system("command args …")
```

- Beispiel: Mails aus einem Programm heraus versenden

```
import os

message = file("/tmp/msg", "w")
message.write("test 1 2 3 \n")
message.close()

os.system("Mail -s Betreff zach < /tmp/msg")
```


Kontrollstrukturen (*flow control*)

- Ändern den Ablauf der Ausführung der Befehle
- Fassen Befehle zu einem größeren Ganzen zusammen

> We sometimes like to point out the close analogies between computer programs, on the one hand, and written poetry or written musical scores, on the other. All three present themselves as […] symbols on a two-dimensional page […]. Yet, in all three cases, the visual, two-dimensional, *frozen-in-time* representation communicates (or is supposed to communicate) something rather different, namely a process that *unfolds in time*. A poem is meant to be read; music, played; a program, executed as a sequential series of computer instructions.
>
> (Numerical Recipes)

Block

- Keine Kontrollstruktur im eigentlichen Sinn
- Dient zur Zusammenfassung mehrerer Anweisungen
- Unteschied zu C++: Blockzugehörigkeit wird in Python über Einrückung angezeigt!
- Beispiel:

```
a = 1
b = a*a
```

selbe Einrückungstiefe!

  - in C++:

```
{
    a = 1;
    b = a*a;
}
```

- Wird fast ausschließlich für Kontrollstrukturen gebraucht
- Kann man schachteln ("innerer" und "äußerer" Block)

- Etwas längerer Vergleich zeigt den Vorteil der Python-Syntax:

Python

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

C++

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0)
    {
        printf("%d\n", i);
        if (i%5 == 0)
            printf("Bingo!\n");
    }
    printf("---\n");
}
```

Exkurs über Strukturierung von Blöcken

- Es gibt drei Möglichkeiten Kontroll-Strukturen syntaktisch zu behandeln:
  1. Folgen von Anweisungen mit explizitem Ende (Algol-68, Ada, COBOL, Shell-Sprachen)
  2. Einzelne Anweisung (Algol-60, Pascal, C)
  3. Einrücken (ABC, Python)

```
IF condition THEN
     stmt;
     stmt;
     ..
ELSIF condition THEN
     stmt;
     ..
ELSE
     stmt;
     ..
END IF;
next statement;
```

Folge von Anweisungen mit Ende-Keyword

```
IF condition THEN
     BEGIN
        stmt;
        stmt;
     END    ..
ELSIF condition THEN
     BEGIN
        stmt;
        ..
     END;
ELSE
     BEGIN
        stmt;
     ..
     END;
next-statement;
```

Einzelne Anweisung mit Block-Keywords

`begin … end` ist, grammatikalisch gesehen, 1 Statement! Dito für `{…}`

```
IF condition:
     stm;
     stm;
     ..
ELSIF condition:
     stm;
     ..
ELSE:
     stm;
     ..
next-statement
```

Einrückung

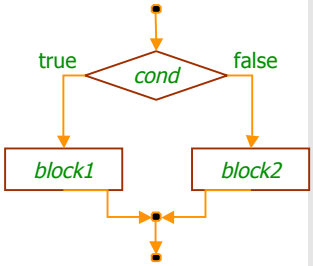
If

- Beispiel:

```
if a < b :
    # Block: do something
else:
    # Block: do somehting else
```

  - In C++:

```
if (a < b)
{
        ...
}
else
{
        ...
}
```



- Beliebte Falle:

```
if i = 1 :
    block
```

  - In Python: zum Glück Fehlermeldung
  - In C++: keine Fehlermeldung, aber schwierig zu findender Bug

Geschachtelte If's

- Wie in C++: If's kann man schachteln, d.h., Anweisung(en) innerhalb **`if`** oder **`else`** können wieder **`if`**'s enthalten ("inneres" und "äußeres" If)
- Bei langen "Listen" von geschachtelten If's empfiehlt sich **`elif`**:

```
if condition1 :
    # Block1
else:
    if condition2 :
        # Block2
    else:
        if condition3 :
            # Block3
        else:
            # Block4
```

```
if condition1 :
    # Block1
elif condition2 :
    # Block2
elif condition3 :
    # Block3
else:
    # Block4
```

- Beispiel (vollständige Fallunterscheidung):

```
# sort 3 numbers
if x < y :
    if y < z :
        pass # schon sortiert
    else :
        if x < z :
            x, y, z = x, z, y
        else :
            x, y, z = z, x, y
else :
    if x < z :
        x, y, z = y, x, z
    else :
        if y < z :
            x, y, z = y, z, x
        else :
            x, y, z = z, y, x
```

- Hier kann man natürlich nicht umschreiben

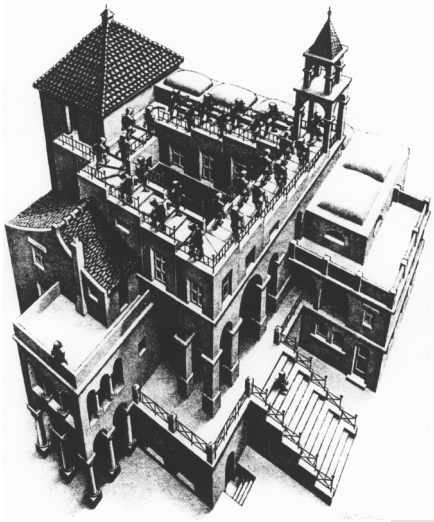
Schleifen

"Life is just one damn thing after another."

-- Mark Twain

"Life isn't just one damn thing after another … it's the same damn thing over and over and over again."

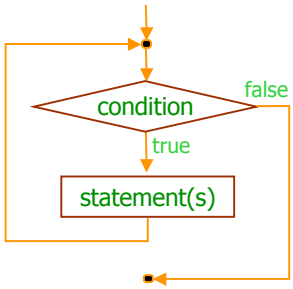
-- Edna St. Vincent Millay



While-Schleife

- Definition (Syntax & Semantik):

`while` *condition* **:**
statements



- Beispiel:

Python

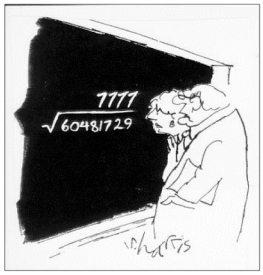
```
b = input()
a = 1
while a < b :
    a *= 2
print a
```

C++

```
// int b
int a = 1;
while ( a < b )
{
    a *= 2;
}
```


Beispiel: Quadratwurzeln (Newton-Raphson)

- Ziel: Berechnung der Quadratwurzel einer Floatingpoint-Zahl c
- Initialisiere t = c
- Ersetze t durch den Mittelwert von t und c/t
- Wiederhole, bis t=c/t



"A wonderful square root. Let's hope it can be used for the good of mankind."

```
c = 2.0
t = c
while t - c/t > 0.00000000001 :
        t = ( c/t + t ) / 2.0
print t
```

Funktionsweise der Newton-Raphson-Methode

- Ziel: Finde die Nullstelle einer Funktion f(x)
  - z.B. $f(x) = x^2 - c$
- Starte mit einem beliebigen t_0
- Betrachte die Tangente an dem Punkt (t_i,$f(t_i)$)
- t_{i+1} ist der Punkt, an dem diese Tangente die x-Achse schneidet
  - d.h. $t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)}$
- Wiederhole dies, bis Nullstelle gefunden
- Anwendungen
  - Nullstellen differenzierbarer Funktionen finden
  - Extrempunkte zweifach differenzierbarer Funktionen finden