Informatik II
Einführung in Python, Basics
Vergleich mit C++

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

Intro

- Skript-Sprache
  - Nich compiliert, sondern "interpretiert"
  - "Glue"-Sprache (Filter-Skripte, Prototyping, ...)
- Erfinder: Guido von Rossum
- Web-Seite: www.python.org
- Newsgroup: comp.lang.python
- Sehr aktive Weiterentwicklung (PEPs):
  - "Python Enhancement Proposal"
  - http://www.python.org/peps/
  - Wohldefinierte Prozedur unter Einbeziehung der Community
- Achtung: wir verwenden Version 2.6 !

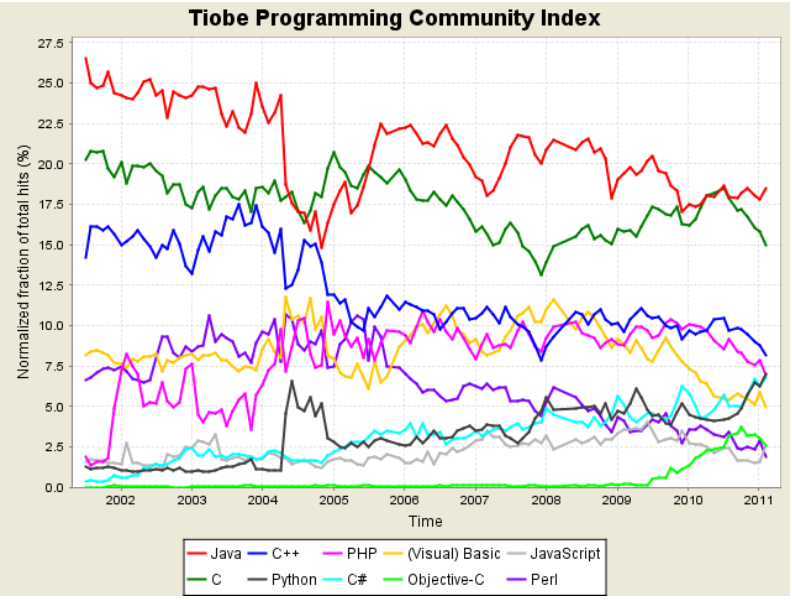
Warum Python?

- Einige allgemeine Konzepte von Programmiersprachen lassen sich besonders leicht durch Vergleich (z.B. mit C++) erklären
- Als Informatiker muß man viele Sprachen lernen
- Als Informatiker muß man lernen, das dem Problem angepaßte "Mind-Set" (= Programmiersprache) zu wählen (→ Intuition!)

> *Programming languages are not merely technologies, but habits of mind as well.*
> [aus Paul Graham: "Beating the Averages", http://www.paulgraham.com/avg.html]

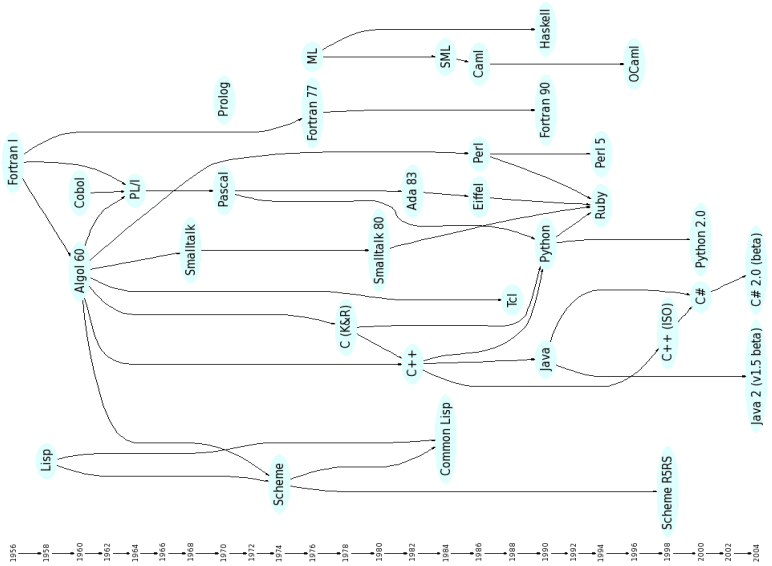
- Python ist eine VHL (very high level) Sprache → eignet sich gut zum "prototypischen" Programmieren
- Leicht erlernbar: wenig Keywords, klare Konzepte
  - "Life's better without braces" (Bruce Eckel)
- Python verbreitet sich sehr stark
  - Ist z.B. eine der Hauptsprachen bei Google

Der Tiobe-Index



http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Programming Languages History

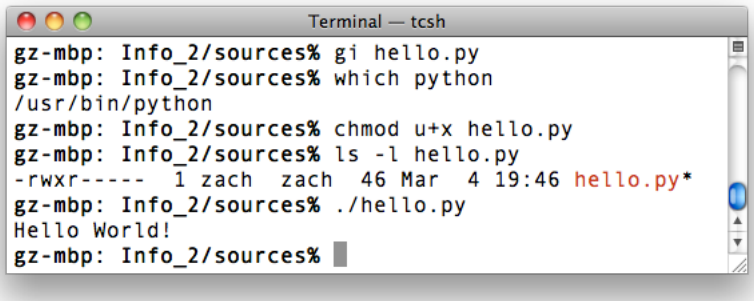


Hello World in Python

- Muß jeder Programmierer einmal geschrieben haben! ☺

Python-Skripte

- Skript = ASCII-File mit gesetztem Exec-Permission-Bit (Linux / Mac)
- Beispiel:

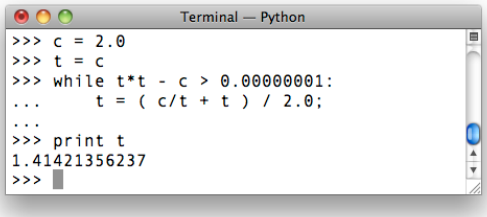


Ein wenig Computer-Folklore

- The ACM "Hello World" project:
  http://www2.latech.edu/~acm/HelloWorld.html
- Für erfahrene Programmierer:
  http://www.gnu.org/fun/jokes/helloworld.html
- A propos "Real Programmers":
  http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?Real+Programmers+Don't+Use+Pascal

Die interaktive Python-Shell

- **`python`** im Terminal aufrufen
  - Kommandozeilen-History
  - Kommandozeilen-Editor
- Mehrzeilige Konstrukte mit Leerzeile abschließen
- Beispiel:



Python IDEs

- Nicht wirklich nötig in diesem Kurs!
- Python Tools for Visual Studio: http://pytools.codeplex.com/
- Eric: http://eric-ide.python-projects.org/
- Boa Constructor: http://boa-constructor.sourceforge.net
- Eclipse mit PyDev:
  - Nur geeignet für jemand, der Eclipse schon beherrscht
- Xcode: Mac only
  - Leider geht damit kein Debugging
- Wing IDE 101: http://wingware.com/wingide-101
  - Cross-platform; inkl. kleinem Debugger

Kommentare

- Startet mit **#**, bis zum Ende der Zeile

```
x = 10  # Bedeutung der Variable
# Kommentarzeile
```

- Zum Vergleich in C++

```
const int Ntries;  // the rest of the line …
// … is treated like a comment
```


- Langer Kommentarblock

```
"""
Blubber
bla bla bla bla.
"""
```

  - Kann überall dort stehen, wo ein Statement stehen kann
  - Funktioniert nicht so allgemein in der interaktiven Pyhton-Shell
- In C/C++

```
const int Ntries;
/*  this is a multiline comment:
    Everything is treated like a comment.
    Comments can't be nested. The comment is
    closed with a */
```

Identifier

- Wie in C++ und praktisch jeder anderen Sprache
- Anderes Wort für Name (Variablename, Funktionsname)
- Zeichenkette
  - Zugelassen: alphanumerische Zeichen und Underscore (_)
  - Erstes Zeichen darf nicht Ziffer sein
    - blub und _bla sind ok
    - 2pi nicht ok
- Kein Limit auf Länge
- Case-sensitiv

Keywords

- Wörter mit spezieller Bedeutung
- Sind reserviert, kann man nicht als Namen verwenden

Keywords in Python				
and	del	for	is	raise
assert	elif	from	lambda	return
break	else	global	not	try
class	except	if	or	while
continue	exec	import	pass	yield
def	finally	in	print	

In C++ :

keyword			
asm	else	operator	throw
auto	enum	private	true
bool	explicit	protected	try
break	extern	public	typedef
case	false	register	typeid
catch	float	reinterpret_	typename
char	for	return	union
class	friend	short	unsigned
const	goto	signed	using
const_cast	if	sizeof	virtual
continue	inline	static	void
default	int	static_cast	volatile
delete	long	struct	wchar_t
do	mutable	switch	while
double	namespace	template	
dynamic_cast	new	this	

Eingebaute (*built-in*) einfache Typen

- **`int`** : Integers (ganze Zahlen)
- **`bool`** : Wahrheitswerte
- **`float`** : Floating-Point-Zahlen (floats) ("reelle Zahlen")
- **`str`** : Strings (im Gegensatz zu C++ echter Datentyp)
- Höhere Datenstrukturen
  - Liste, Komplexe Zahlen, Dictionary: später
  - Set, Module, File, ...
- Unterschiede zu C++:
  - **`int`** kann beliebig groß werden (Python schaltet intern autom. um)
  - **`float`** = **`double`** (ex. kein single precision float)
  - Höhere Datenstrukturen (**`str`**, Sequenzen, **`complex`**, **`dict`**, etc.) fest eingebaut, nicht über Header-Files / Libraries

Literale

- Im großen ganzen wie in C++:

Literal	Bemerkung
`123 0123 0x123`	**int's (decimal, octal, hex)**
`3.14 3E1`	**floats**
`"" "123"`	**string (first = null string)**
`"name"`	
`"a \"string\""`	**prints: a "string"**
`"""a string ...`	**string over**
`spanning two lines"""`	**several lines is built-in**
`True False`	**bool's**
`None`	**"Zeiger-Wert" / 0-Obj. (NULL in C++)**
`() (1,) (1,2,3)`	**Tupel**
`[] [1,2,3]`	**Liste**
`type(x)`	**Typen kann man vergleichen und zuweisen**

Variablen

- Ähnliche Funktion wie in Mathematik, speichert Wert
- Variable = Paar von Name und "Zeiger" auf Speicher (*Binding*)
- Unterschiede zu C++ und Java:
  - Variablenname hat keinen Typ!! Wert der Variable sehr wohl!
  - Ist nur "Zeiger" auf Wert im Speicher, dieser kann beliebig oft geändert werden
  - Braucht nicht deklariert zu werden!
  - Muß natürlich vor der Verwendung erzeugt worden sein
- Erzeugung / Initialisierung:

```
seconds_per_day = 60*60*24   # erzeugt Variable
seconds_per_day = 86400      # ändert deren Wert
```


- Objekte im Speicher haben immer einen Typ

```
>>> a = 1
>>> type(a)
<type 'int'>
>>> a = "Hello"
>>> type(a)
<type 'string'>
>>> type(1.0)
<type 'float'>
```

Style-Guidelines für Variablen

- Wie in C++ und jeder anderen Sprache
- "Kleine" Variablen:
  - Laufvariablen, Variablen mit sehr kurzer Lebensdauer
  - 1-3 Buchstaben
  - `i,j,k,`... für int's
  - `a,b,c,x,y,z` ... für float's
- "Große" Variablen:
  - Längere Lebensdauer, größere Bedeutung
  - Verwenden Sie *labeling names*! ("Sprechende Namen")
  - `mein_alter, meinAlter, determinante,` ...

Operatoren und Ausdrücke

- Ausdruck (*expression*) = mathematischer oder logischer Term
- Beispiele:

```
import math                      # lade Mathe-Funktionen
math.sin(x)*math.sin(2*x)        # arithm. expression
"to " + "be"                     # string expr.
2 < 1                            # logic expression
(x > "aaa") and (x < "zzz")      # dito
```

- Ausdruck setzt sich zusammen aus:
  - Literalen (Konstanten), Variablen, Funktionen
  - Operatoren
  - Klammern
  - Übliche Regeln

Operatoren und Ausdrücke

Arithm. Operator			Meaning
-i	+w		unary + and -
a*b	a/b	i%2	mult., div., modulo
a+b	a-b		binary + and -
x**y			power

Definiert für alle numerischen Typen (im Gegensatz zu C++)

Bit-wise Operators	
~ i	bitwise Complement
i & j	bitwise AND
i | j	bitwise OR
i ^ j	bitwise XOR

Relational Operators	
<	less than
>	greater than
<=	less or equal
>=	greater or equal
==	equals
!=	not equal

Definiert für alle Typen! (im Gegensatz zu C++)

Assignment op.	equivalent
x □= y	x = x □ (y)
Bsp.:	
x -= y	x = x - y
x /= y	x = x / y
x &= y	x = x & y

Boole'sche Operatoren

- in Python

Boolean Operators	
not	unary not
and	logical and
or	logical or

- Beispiele

```
not x
(not x) and y
((not x) and y) or (z and w))
```

- in C++

Boolean Operators	
!	unary not
&&	logical and
||	logical or

Beispiel: Berechnung von Schaltjahren

```
y = 2005
# Durch 4 teilbar, aber nicht durch 100
isLeapYear = (y % 4 == 0) and (y % 100 != 0)
# Oder durch 400 teilbar
isLeapYear = isLeapYear or (y % 400 == 0)
```


Increment- / Decrement-Operatoren

- In C++:

Increment and decrement	Equivalent to	
`k = ++ j;`	`j = j + 1;`	`k = j;`
`k = j ++;`	`k = j;`	`j = j + 1;`
`k = -- j;`	`j = j - 1;`	`k = j;`
`k = j --;`	`k = j;`	`j = j - 1;`

- Gibt es in Python nicht!
- Problem mit Increment- / Decrement-Operatoren:
  - Nebeneffekt → schlechte Wartbarkeit; und
    - Beispiel: Tabelle oben
  - Reihenfolge der Auswertung der Operanden nicht festgelegt → versch Ergebnisse
    - Beispiel:

```
while ( source[j] )
        dest[i++] = source[j++] ;
```

```
i = 2;
j = (i++) * (i--);
j = ?
```

Ganzzahlarithmetik

- Wertebereich
  - Integers können beliebig lang werden
- Overflow:
  - Kein Überlauf in Python!
- Division:
  - wie in C++: 3/2 → 1, 3.0/2 → 1.5
- Division und Modulo: `(x/y)*y + x%y == x`

```
import sys
print sys.maxint
2147483647
print sys.maxint * 2
4294967294
```

Float-Arithmetik (praktisch identisch zu C++)

- Implementiert IEEE 754-1985 Standard
- Überlauf ("overflow"):
  - Zahl wird zu groß / zu klein
  - Beispiel: max.float * 2
  - Resultat = +∞ bzw. -∞
- Underflow:
  - Zahlen liegen zu dicht an der 0
  - Resultat = +0.0 bzw. -0.0
- NaN (Not a Number) (in C++):
  - Rechnungen, wo kein sinnvoller Wert rauskommt
  - Bsp.: 1/0 , ∞*0 , sqrt(-1.0)
  - In Python: Fehlermeldung

Beispiel: Quadratische Gleichungen

- Lösen der quadratischen Gleichung $x^2 + bx + c = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

```
import math
sqroot = math.sqrt(b*b - 4.0*c)
root1 = (-b + sqroot) / 2.0
root2 = (-b - sqroot) / 2.0
```


Typ-Konvertierung (Cast)

- 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 später!

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 OR
	`< … >=`	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`**

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
```

- Formatiert:

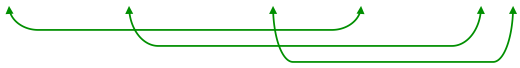
```
print "format-string" % (arg1, arg2, …)
```

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

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**

Beispiel: Tabellarische Ausgabe

- Einfache Ausgabe:

```
"%d %f"
```

```
 x      y
-10 1.321
-1 3.452678
0 7.701345
117 -0.001
```

- Padding für positive Zahlen:

```
"% d % f"
```

Space

- Mit Spaltenbreite:

```
"% 10d % 10f"
```

- Mit Präzision:

```
"% 10d % 10.3f"
```

```
 x       y
-10     1.321
-1      3.452
 0      7.701
 117   -0.001
```


Lesen von stdin

- Ganze Zeile als String einlesen:

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

  - Das Prompt ist optional

- Einzelne Zahl einlesen:

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

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

Komplexere Eingaben lesen

- Mehrere Zahlen in einer Zeile einlesen:

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

  - Eingabe muß `1, 2, 3` sein!

- 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 einzelne Wörter auf und liefert ein Array

System-Aufrufe

- Aufgabe: 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 = open("/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 durch ein eigenes Keyword (Algol-68, Ada, COBOL, Shell-Sprachen)
  2. Einzelne Anweisung mit speziellen Keywords/Zeichen, um eine Folge von Anweisungen zu einer Anweisung zu klammern (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

Die Kontrollstrukturen im Überblick

- If:

```
if condition :
    # Block: do something
else:
    # Block: do somehting else
```

- While:

```
while condition :
    # Block: do something
```

- For:

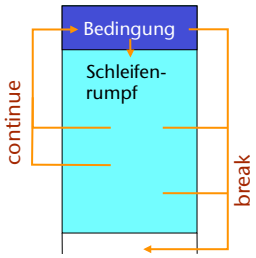
```
for x in list :
    # Block: do something
```

  - Häufigster Fall:

```
for x in range(0,n):
    # x = 0, 1, ..., n-1
```


- Break & continue:
  - Brechen eine Schleife ab bzw. starten den nächsten Schleifendurchlauf

```
while oder for:
    if ...:
        break
    if ...:
        continue
    ...
    # continue jumps here
# break jumps here
```



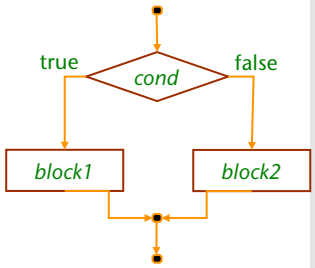
If

- 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

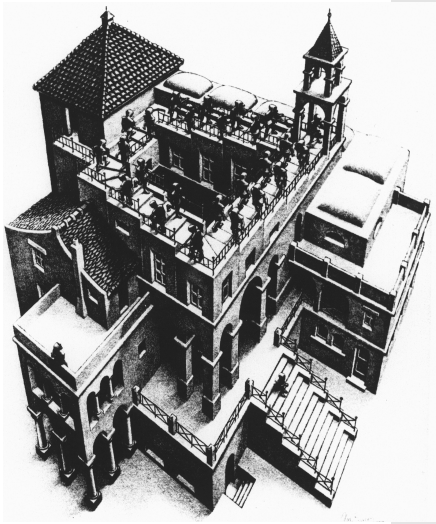
Schleifen

"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."

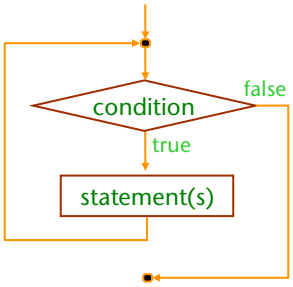
-- Edna St. Vincent Millay



While-Schleife

- Definition (Syntax & Semantik):

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



- Beispiel:

Python

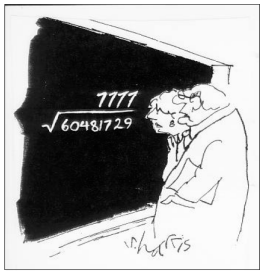
```
b = input()
a = 1
while a < b :
    a *= 2
print a
```

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

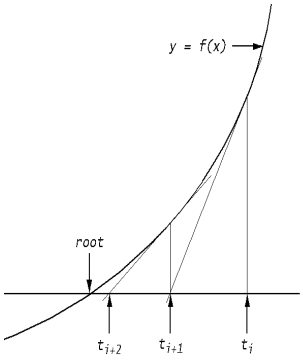


"A wonderful square root. Let's hope it can be used for the good of mankind."

```
c = 2.0                          # input
t = c
while t*t - c > 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

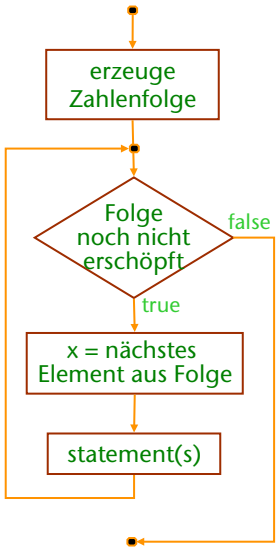


For-Schleife

- Anders als in C++
- Definition:

```
for x in list :
    statements
```

- Oft wird *list* durch die **`range`**-Funktion generiert (s. später)
- Statt **`range`** kann jede andere Art von Listen stehen
- Erstes Beispiel eines Iterators!



Beispiel C++ vs. Python

- Beispiel: geometrische Reihe

```
# float q, int n
s = 0.0
# s = geom. Reihe 1 + q + q^2 + q^3 + … + q^n
qq = 1.0
for i in range(0,n):
    s += qq
    qq *= q
print s
```

Loop body (Schleifenrumpf)

"Schleifenvariable" (loop variable)

- in C++:

```
// float q; unsigned int n;
float s = 0;
// s = geom. Reihe 1 + q + q^2 + q^3 + … + q^n
float qq = 1;
for ( unsigned int i = 0; i < n; i ++ )
{
    s += qq;
    qq *= q;
}
```


Beispiel: Mittelwert

Diese Funktion liest alle Zeilen auf einmal in eine Liste

```
lines = sys.stdin.readlines()
n_values = 0
sum = 0.0
for line in lines:
    sum += float(line)
    n_values += 1
if n_values > 0 :
    average = sum / n_values
    print "Average = %f\n" % (average)
else:
    print "No values on stdin!"
```

Problem: was, wenn der input stream gar nicht mehr aufhört?!

Oder Fehlercode zurückgeben, oder Exception werfen, oder ...

NB: Eingabe auf stdin mit CTRL-D abschließen.
(CTRL-D = EOF [end-of-file] unter unix)

Längeres Beispiel: Longest Run

```
list_size = 3
n_values = 0
longest_run = 0
length = 0
last_value = 0
while n_values < list_size:
     value = input()
    if value == last_value:
        length += 1
    else
        length = 1
    if longest_run < length:
        longest_run = length
    last_value = value
    n_values += 1
print "Longest Run = %d\n" % (longest_run)
```

Angenommen, wir wüssten das


```
list_size = 3
n_values = 0
longest_run = 0
length = 0
last_value = 0
while n_values < list_size:
    value = input()
    if value == last_value:
        length += 1
    else
        length = 1
    if longest_run < length:
        longest_run = length
    last_value = value
    n_values += 1
print "Longest Run = %d\n" %(longest_run)
```

Eingabe: 1.0 5.0 5.0

list_size	3
n_values	3
longest_run	2
length	2
last_value	5.0
value	5.0

Korrekte Programme durch vollständige Fallunterscheidung

```
list_size = 3
n_values = 0
longest_run = 0
length = 0
last_value = 0
while n_values < list_size:
    value = input()
    if value == last_value:
        length += 1
    else
        length = 1
    if longest_run < length:
        longest_run = length
    last_value = value
    n_values += 1
print "Longest Plateau = %d\n" % (longest_run)
```

Kennt man i.A. nicht!


```
import sys
lines = sys.stdin.readlines()
longest_run = 0
length = 0
last_value = 0
for line in lines:
    value = float(line)
    if value == last_value:
        length += 1
    else
        length = 1
    if longest_run < length:
        longest_run = length
    last_value = value
print "Longest Plateau = %d\n" % (longest_run)
```

Diese Funktion liest alle Zeilen auf einmal in eine Liste

Problem: was, wenn float(lines[0]) == 0?!

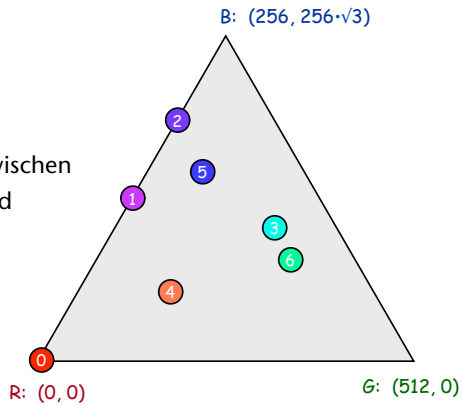
NB: Eingabe auf stdin mit CTRL-D abschließen.
(CTRL-D = EOF [end-of-file] unter unix)

```
import sys
lines = sys.stdin.readlines()
length = 0
longest_run = 0
first = True
for line in lines:
    value = float(line)
    if first:
        last_value = value - 1
        first = False
    if value == last_value:
        length += 1
        if longest_run < length:
            longest_run = length
        else
            length = 1
    last_value = value
print "Longest Plateau = %d\n" % (longest_run)
```

Noch ein Problem:
was, wenn der input stream gar nicht mehr aufhört?!

Beispiel: "Chaos Game"

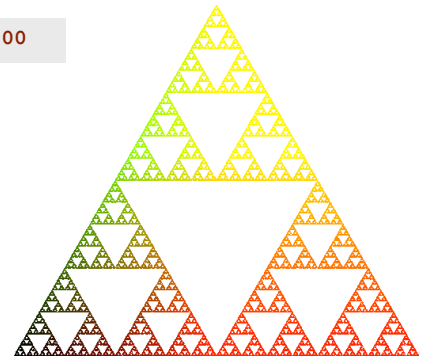
- Spiel in einem gleichseitigem Dreieck, dessen Ecken rot (R), grün (G) und blau (B) eingefärbt sind
- Starte bei Punkt R
- Wiederhole:
  - Wähle zufällig einen Eckpunkt
  - Gehe die Hälfte der Strecke zwischen momentanem Standpunkt und dem ausgewählten Eckpunkt
  - Male dort einen Punkt



```
import Image       # Windows: from PIL import Image
import random
import sys

im = Image.new("RGB", (512,512), (256 256,256) )
N = int( sys.argv[1] )            # number of iterations

x = 0.0
y = 0.0

for i in range( 0, N ):
    r = random.random()
    if r < 0.333:
        x0 = 0.0
        y0 = 0.0
    elif r < 0.6667:
        x0 = 512.0
        y0 = 0.0
    else:
        x0 = 256.0
        y0 = 443.4
    x = ( x0 + x ) / 2.0
    y = ( y0 + y ) / 2.0
    im.putpixel ( (int(x), int(y)), (int(x), int(y), 0) )

im.show()
```

Hier fehlt eigtl ein Test, ob überhaupt ein Command-Line-Argument angegeben wurde!


```
% ./Sierpinski.py 100000
```



- Aufgabe: ausprobieren, was passiert, wenn man einen zufälligen Punkt im Inneren des ursprünglichen RGB-Dreiecks als "Seed" nimmt

Geschachtelte Schleifen (nested loops)

- Analog wie in anderen Sprachen
- Schleifenrumpf kann wieder Schleife enthalten:

```
for i in range(…):
    for j in range(…):
        ...
```

Andere Schleifenvariable nehmen!

- Beispiel:

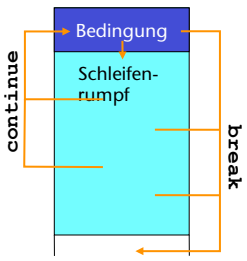
```
for i in range(0,5):
    for j in range(0,i):
        print "*",
    print ""
```

Ausgabe

```
*
**
***
****
```


Break und continue

- Wie in C++: zusätzliche Sprünge innerhalb der Schleife
  - **break**: springt aus Schleife heraus (nur 1 Level!)
  - **continue**: startet sofort den nächsten Schleifendurchlauf




```
while ...:
    if ...:
        break
    if ...:
        continue
    ...
    # continue jumps here
# break jumps here
```

```
while (...)
{
    if ( ... )
        break;
    if ( ... )
        continue;
    ...
    // continue jumps here
}
// break jumps here
```

Interaktives Programm

- Lissajous-Figuren
  - Idee: zwei orthogonale Schlitten, die hin- und herfahren und gemeinsam einen Stift führen
  - Schlitten werden durch periodische Funktionen gesteuert, z.B. sin/cos