Einführung in Python

Arne Hüffmeier

1. Ziele der Vorlesung
2. Einstieg in Python
3. Variablen
4. Blöcke
5. Abfragen
6. Schleifen
7. Listen
8. Ende

Was soll vermittelt werden?

Problemorientiertes Denken

Warum Python ?

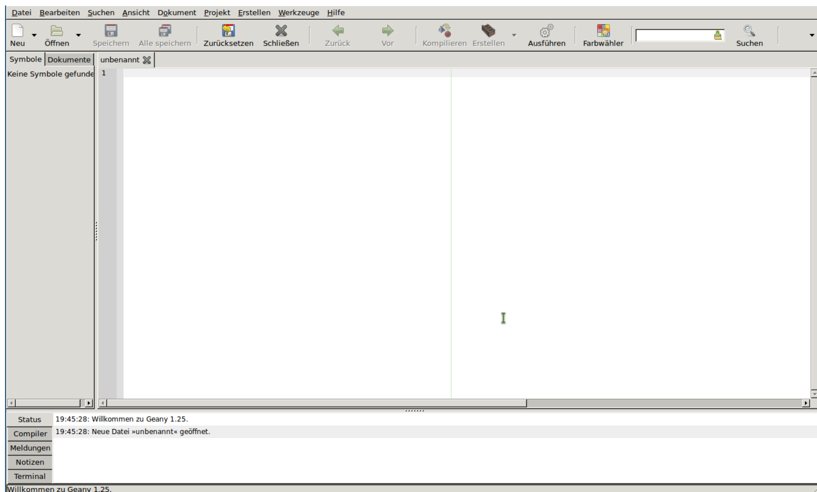
Vorteile

- einfache Syntax
- kein unnötiger Overhead
- relativ einfache Konstrukte
- viele Funktionen gibt es schon

Nachteile

- langsamere Ausführung

Geany einrichten



Geany einrichten

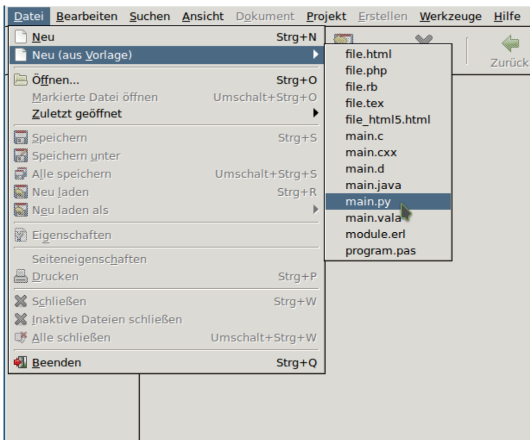


Abbildung: Neue Python Datei erstellen.

Geany einrichten

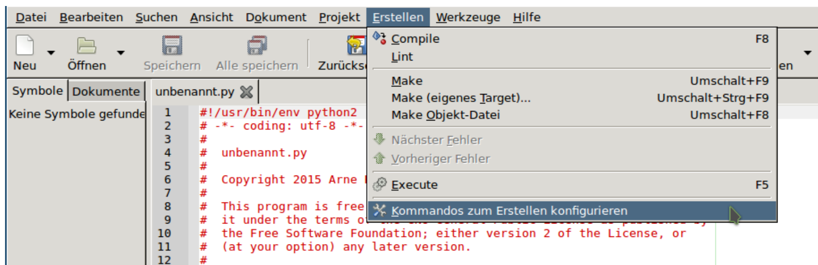


Abbildung: Kommandos zum Erstellen konfigurieren.

Geany einrichten

#	Label	Kommando	Arbeitsverzeichnis	Zurücksetzen
Kommandos für Python				
1.	Compile	python -m py_compile "%f"		
2.				
3.	Lint	pep8 --max-line-length=80 "%f"		
Regulärer Ausdruck für Fehlermeldungen:		(.+):([0-9]+):([0-9]+)		
Dateitypunabhängige Befehle				
1.	Make	make		
2.	Make (eigenes Target)...	make		
3.	Make Objekt-Datei	make %e.o		
4.				
Regulärer Ausdruck für Fehlermeldungen:				
Notiz: Element 2 öffnet ein Dialog und fügt das Ergebnis am Ende des Kommandos an				
Befehle zum Ausführen				
1.	Execute	python "%f"		
2.				

%d, %e, %f, %p, %l werden innerhalb der Kommando- und Verzeichnisfelder ersetzt - Details gibt es in der Dokumentation.

Abbrechen | OK

Abbildung: Aus *python* *python3* machen

Das Erste Programm

```
1 print("Hallo World")
```

Das *print* gibt an, dass etwas ausgegeben werden soll.
In den Anführungszeichen kann ein beliebiger Text stehen.

Was können wir jetzt damit machen?

- Wir können Text ausgeben ;-)
- Wir können rechnen.

Operationen

- ```
  print(10 + 2)
  ```
- ```
  print(10 - 2)
  ```
- ```
  print(10 * 2)
  ```
- ```
  print(10 + 3 + 2)
  ```
- ```
  print(10 % 3)
  ```
- ```
  print(10 ** 3)
  ```

Ein paar Infos am Rande

Infos über print

```
print("Hallo")
print("Du Da")
```

Der *print* Befehl gibt eine Zeile aus. Somit würde das

```
Hallo
Du Da
```

ergeben. Man kann aber den Zeilenumbruch am Ende unterdrücken oder durch etwas anderes ersetzen.

```
print("hallo",end="")
print("Du Da")
```

Wie Python das macht und warum das *end* nicht in " steht, klären wir bei dem Thema Funktionen.

Noch ein paar Infos am Rande

Infos über print

Es ist auch möglich mehrere Dinge in einem print auszugeben

```
print("Hallo", "Du", "Da")
```

Ausgabe

```
Hallo Du Da
```

Wenn man keine Leerzeichen haben will, kann man das so machen

```
print("Hallo"+"Du"+"Da")
```

oder so

```
print("Hallo","Du","Da",sep='')
```

Numerische Operationen

- + Addition
- - Subtraktion
- * Multiplikation
- / Division
- % Modulo (Division mit Rest)
- ** Potenz

Variablen

Aus der Schule kennt ihr

$$f(x) = \dots$$

Das x ist eine Variable, die für eine beliebige Zahl steht.

In der Informatik ist eine Variable ein Zwischenspeicher für einen Wert.

Variablentypen

Beispiele für Typen

boolean: True oder False

int: 42

float: 3.141

Wie benutzt man Variablen?

Da Python auch weiß welcher Datentyp wie aussieht, ist das einfach

```
1 a = 1
2 b = True
3 c = 3.48984
```

Wichtig !!!: Zum Trennen von Vor- und Nachkommastelle wird **kein** Komma sondern ein Punkt verwendet.

Umgang mit Typen

- ```
  a = 5
  b = 10
  print(a + b)
  ```
- ```
  print(a - b)
  ```
- ```
  print(a + b)
  ```
- ```
  print(a % b)
  ```
- ```
  c = True
  print(c)
  ```
  ```

Numerische Operationen

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo (Division mit Rest)
+=	Addition mit Zuweisung a += b -> a = a + b
-=	Subtraktion mit Zuweisung a -= b -> a = a - b

Numerische Operatoren

```
a = 10
b = 20
```

```
a += b
print(a)
```

```
b -= a
print(b)
```

Noch ein paar Infos am Rande

Python kann auch die Rechnenregeln

5+5*3 ist nicht das gleiche wie (5+5)*3

und es versucht den Typ zu raten

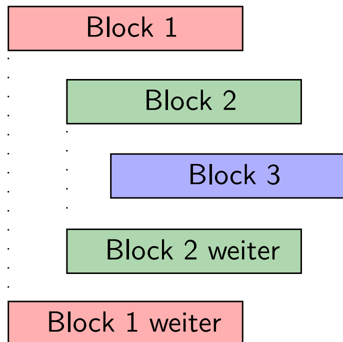
```
23.0/3.0 = 7.666666666666667
```

und

```
23/3 = 7.666666666666667
```

aber

```
21/7 = 3.0
```



Damit der Code übersichtlicher ist, kann und muss man ihn in Blöcke unterteilen.

Die Einrückung besteht aus 4 Leerzeichen **oder** einem Tab.

WICHTIG: Innerhalb einer Datei darf nicht gewechselt werden.

Python Keywords

```
1 pass
```

Mit pass kann man sagen, dass nichts passiert. Dies ist wichtig, wenn eine Einrückung erforderlich ist, aber dort nichts gemacht werden soll.

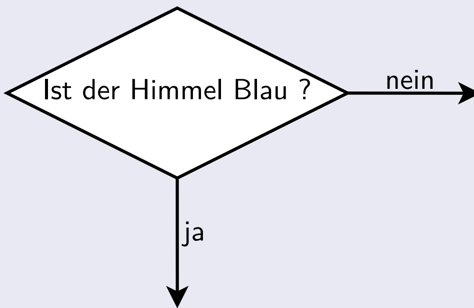
```
1 # Ich bin ein kommentar
```

Alle Zeilen, die mit einer # anfangen, werden von Python ignoriert.

Abfragen

Bis jetzt ist es so, dass jede Zeile, die ihr in die Datei schreibt, auch ausgeführt wird.
Das ist aber nicht immer gewollt und somit gibt es bedingte Befehlsblöcke (if).

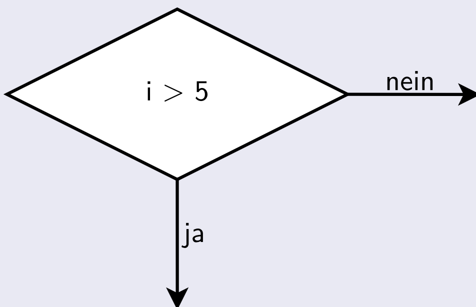
if-Abfrage



Frage?

Wie sähe die Abfrage aus, wenn man überprüfen möchte, ob eine Zahl größer 5 ist?

Antwort



if-Anweisung in Python

```
1 if (5 < 6):
2     print("5 ist kleiner 6")
```

Wenn 5 kleiner als 6 ist, wird der print Befehl ausgeführt sonst nicht.

Blockanweisungen in Python

```
1 if (5 < 6):
2     print("5 ist kleiner 6")
3
4     if(4 < 6):
5         print("4 ist kleiner 6")
6
7         if (3 < 7):
8             print("3 ist kleiner 7")
```

Nur wenn die `if`-Anweisung in Zeile 1 wahr ist, wird die in Zeile 4 überprüft. Und nur wenn diese wahr ist, die in Zeile 7.

if-Anweisung in Python

```
1 if (5 < 6):
2     print("5␣ist␣kleiner␣6")
```

In den runden Klammern in Zeile 1 können unterschiedliche Dinge stehen. Sie müssen aber überprüfbar sein, sodass der Computer bestimmen kann, ob sie wahr oder falsch sind.

Abstrakter Aufbau

```
1 if( <auswertbare Aussage> ):
2 ␣␣␣␣
```

Auswertung

- strikt kleiner

```
if (5 < 6):
```

- strikt größer

```
if (5 > 6):
```

- kleiner gleich

```
if (5 <= 6):
```

- größer gleich

```
if (5 >= 6):
```

- ungleich

```
if (5 != 6):
```

- ist gleich

```
if (5 == 6):
```

Anmerkungen zur if-Abfrage

Aussagen negieren

Mit *not* kann eine Aussage negiert werden

```
if ( not 5 > 6 ):
    print( "5 ist nicht groesser 6")
```

Verbinden von Aussdrücken

Mehrere logische Ausdrücke können in einer `if`-Abfrage überprüft werden

```
if ( <Aussage1> and <Aussage2> or <Aussage3>):
```

bei *and* müssen beide Aussagen wahr sein.
bei *or* reicht es wenn eine der beiden Aussagen wahr ist.

and Verbindung

and	True	False
True	*True*	*False*
False	*False*	*False*

or Verbindung

or	True	False
True	*True*	*True*
False	*True*	*False*

Anmerkungen zur if-Abfrage

if und else

Stellen wir uns vor, wir wollen wissen ob *i* größer oder kleiner gleich 6 ist.
Das könnte so aussehen:

```
1 if (i > 6 ):
2     print( "i␣ist␣Groesser␣6")
3 if (i <= 6 ):
4     print("i␣ist␣kleiner␣als␣6␣oder␣6")
```

Jedoch würden wir so zweimal prüfen. Mit einem *else* geht das leichter und lesbarer.

```
1 if (  i > 6 ):
2     print( "i␣ist␣Groesser␣6")
3 else:
4     print("i␣ist␣kleiner␣als␣6␣oder␣6")
```

Verschachtelung von if

Manchmal möchte man sicherstellen, dass nur, wenn das eine nicht zutrifft, etwas anderes überprüft wird.
Das könnte man so realisieren:

```
1 if (i % 2 == 0):
2     print("i ist gerade")
3 else :
4     if (i < 10):
5         print("i ist ungerade und kleiner 10 ")
```

Oder so:

```
1 if (i % 2 == 0):
2     print("i ist gerade")
3
4 if ((i < 10) and not (i % 2 == 0) ):
5     print("i ist ungerade und kleiner 10 ")
```

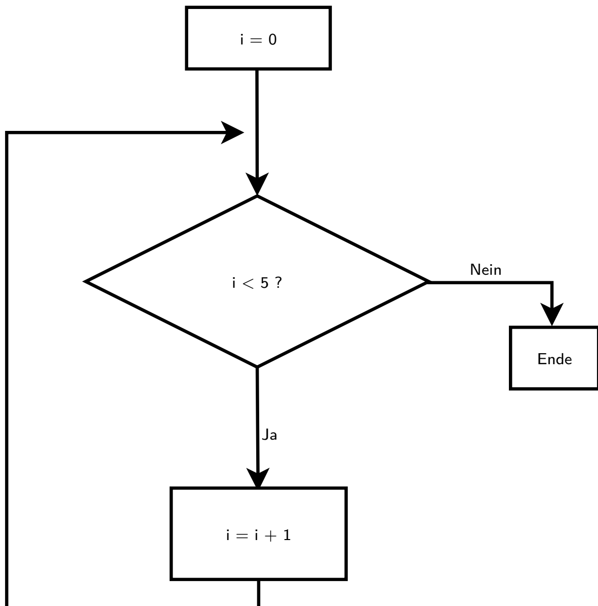
Verschachtelung von if

Da man so etwas aber sehr häufig braucht, gibt es in Python eine Kurzschreibweise.

```
if (i % 2 == 0):
    print("i ist gerade")
elif (i < 10):
    print("i ist ungerade und kleiner 10 ")
else:
    print("i ist nicht gerade und auch nicht kleiner 10")
```

Bis jetzt können wir mit Python nicht viel mehr als mit einem handelsüblichen Taschenrechner. Denn eine wichtige Sache fehlt uns noch.

Schleifen


i = 0
i < 5 ?
Nein
Ende
Ja
i = i + 1

while-Schleife in Python

```python
i = 0
while (i < 5):
    i += i
```

Der eingerückte Teil `i += i` nach dem `while` wird so lange ausgeführt bis die Aussage in den runden Klammern falsch wird.

Bis 10 Zählen

```
1 i = 0
2 while (i < 10):
3     print(i)
```

Abstrakter Aufbau

```
1 while (<auswertbare Aussage>):
```

Natürlich kann man (wie beim `if`) mehrere `while`'s verschachteln

```
1 i = 0
2 while(i < 10):
3     j = 0
4     while(j < 10);
5         print(j)
6         j += j
7     i += 1
```

Verbindung von while und if

Natürlich kann man `if` und `while` beliebig verschachteln

```
i = 0
while (i < 100):
    if (i % 2 == 0):
        print(i, "ist gerade")
    else:
        print(i, "ist ungerade")
    i += 1
```

Listen

Wenn man viele Werte speichern möchte, ist es sehr umständlich für jeden Wert eine Variable anzulegen. Und wenn man nicht weiß, wie viele Werte der Benutzer angibt, ist es sogar unmöglich.

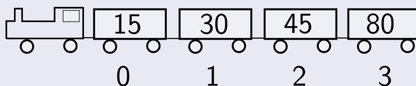
Stellt euch vor, ihr möchtet für eine Formel wie

$$f(x) = x^2 + x * 100$$

die Werte im Bereich zwischen -100 und +100 berechnen.
Das wären 201 Variablen.
Als Lösung haben wir Listen (`Array`).

Listen

Am einfachsten stellt man sich eine Liste als Zug vor.



Die Waggonnummer gibt die Position in der Liste an (mit 0 anfangend durchnummeriert) und in dem Waggon ist die Zahl (oder etwas anderes), die wir speichern wollen.

Listen in Python

Eine Liste in Python zu erzeugen geht so:

```
a = [4,6,12,4,76,8,12]
```

Die Kommas trennen die Einträge.

WICHTIG: benutzt die eckigen Klammern [] und nicht die runden () oder die geschwungenen {}.

Besonderheiten von Listen

Die erste Zahl in der Informatik ist immer die 0, bis auf wenige Ausnahmen. Also hat das erste Element einer Liste den Index 0.

Umgang mit Listen

```
1 liste = [12,16,18,20,22]
2 #Zugriff auf ein Element
3 print(liste[0])
4 print(liste[1])
5 print(liste[2])
6 print(liste[3])
```

```
1 liste = [12,16,18,20,22]
2 print(liste[0])
3 #ueberschreiben von einem Element
4 liste[0] = 11
5 print(liste[0])
```

Umgang mit Listen (Fehlerzustand)

Was passiert, wenn man als Index ein Element angibt, das es nicht gibt?

```
1 liste = [12,16,18]
2 #Zugriff auf ein Element das noch nicht existiert
3 print(liste[3])
```

```
1 Traceback (most recent call last):
2 File "<stdin>", line 1, in <module>
3 IndexError: list index out of range
```

Python sagt: "`list index out of range`"

Umgang mit Listen (Element anhängen)

Jetzt könnte man sich fragen, was es nützt, wenn man alle Elemente einer Liste vorab mit einem Wert belegen muss, damit die Liste lang genug ist. Man kann auch zur Laufzeit eine Liste verlängern.

```
1 liste = [12,16,18]
2 print(liste)
3 liste.append(20)
4 #jetzt ist das element mit dem Index 3 da.
5 print(liste)
```

Ausgabe:

```
1 [12, 16, 18]
2 [12,16,18,20]
```

Umgang mit Listen (Element entfernen)

Und natürlich kann man auch ein Element löschen.

```
liste = [12,16,18]
del(liste[2])
```

Ausgabe:

```
[12, 16]
```

Umgang mit Listen (Länge der Liste)

Um zu wissen, wie lang eine Liste ist, gibt es in Python den `len()` Befehl

```
liste = [12,16,18]
print(len(liste))
liste.append(20)
print(len(liste))
```

Ausgabe:

```
3
4
```

Listen von Listen

Manchmal benötigt man eine Liste von Listen; auch dies ist natürlich möglich.

```
1 temp = [[1,2,3],[4,5,6]]
2 print(temp[0][0])
3 print(temp[1][0])
```

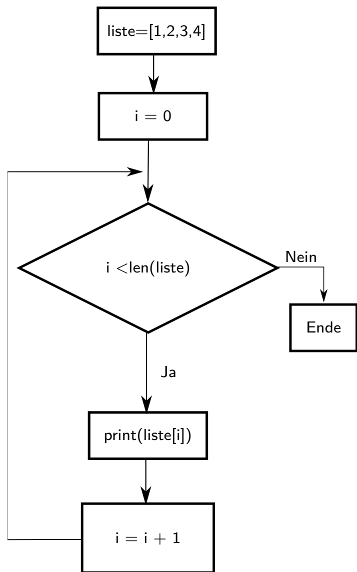
Wie man sich vorstellen kann, ist es möglich dieses weiter zu verschachteln.

```
1 temp = [[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]]
```

Jedoch wird dieses dann sehr schnell unübersichtlich.

Eine kleine Aufgabe

Wie sähe es aus, wenn man mit einer `while`-Schleife alle Elemente einer Liste ausgeben möchte?




```
1 liste = [1,2,3,4]
2
3 i=0
4
5 while(i < len(liste)):
6     print(liste[i])
7     i = i + 1
```

for

Da man sehr häufig über Listen iteriert, gibt es dafür eine Kurzschreibweise.

For

while

```
liste = [1,2,3,4]

i=0

while(i < len(liste)):
    print(liste[i])
    i = i + 1
```

for

```
liste = [1,2,3,4]



for i in liste:
    print(i)
```

Der Befehl range

Wie wir gesehen haben, kann man mit `for` sehr gut über eine Liste iterieren. Aber `for` kann noch mehr.
Allgemein wird die '`for`-Schleife' auch als Zählschleife bezeichnet. Da es aber sehr mühsam ist für eine Schleife, die z. B. 10 Iterationen durchlaufen soll, die Liste

```
lauf = [0,1,2,3,4,5,6,7,8,9]
for i in lauf:
    print(i)
```

zu erstellen, gibt es in Python den Befehl `range`.

```
for i in range(0,10):
    print(i)
```

range

```
range(n,m)
```

erstellt eine Liste, die von n bis m geht. Es geht auch

```
range(10,20)
```

Wichtig ist, das $n \leq m$ ist.
range kann aber noch mehr als nur $+1$ zu rechnen.
Mit einer dritten Zahl kann die Schrittweite angegeben werden.

```
for i in range(0,10,2):
    print(i)
```

Geschafft

Nun habt ihr einen Einstieg in Python

Viel Spaß im Tutorium