

Modul: Programmierung B-PRG Grundlagen der Programmierung 1

V03 Kontrollstrukturen (Verzweigungen + Schleifen)

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Stichwort CodeRunner

1. Auch für uns ist es ein Versuch. Auch die Tutoren waren verunsichert.
2. Warum überhaupt so etwas: Es unterstützt Sie in der Testphase, also kurz vor dem Abgeben ihres Programms:
 - Es ist professionell, ein Testsystem einzusetzen.
 - Falls erfolgreich → mehr Sicherheit für Sie, dass alles richtig ist – **nur Formfehler** (Pogrammierhandbuch) können noch zur Abwertung führen.
 - Falls **nicht** erfolgreich → ja, ggf. mehrmalig etwas penalty: 10% weniger Maximalpunkte, aber eine Möglichkeit weiteres zu lernen.
 - Für die Tutoren: Weniger Korrekturaufwand.
3. Aber Sie können auch nur die .py Dateien abgeben → höheres Risiko für Fehler. Ab nächster Woche, Aufgabe EPR_02 **einmailig** -25% der Maximalpunkte.

Unser heutiges Lernziele

- **Unsere Programme sind noch lanweilig!**
- **Fallunterscheidung und Iteration** (sowie *Rekursion*) sind als *grundlegende mathematische und informatische Lösungsmethoden zu erfassen und deren Realisierungen in Python sind kennen zu lernen.*
- *Hierzu gehören insbesondere die verschiedenen Ausprägungen von **Fallunterscheidungen und Schleifen für die Iteration***
- *Rekursion und deren Realisierungen in Programmiersprachen: **Prozedur und Funktion** (am übernächsten Montag)*
- *Die verschiedenen Mechanismen zur Parameterübergabe kennenlernen*

Übersicht

- 1. Fallunterscheidungen: Verzweigungen**
 - Prinzipien
 - Graphische Repräsentationen
 - Realisierung in Python: `if – else – elif`
- 2. Iterative Grundstrukturen**
 - Prinzipien
 - Schleifen: Realisierungsformen der Iteration
 - Realisierungen in Python:
`for und while`
`break – continue_ – pass – else`

Kontrollstrukturen

Verzweigungen bilden zusammen mit den **Schleifen** und den **Prozeduren** die **Kontrollstrukturen** moderner Programmiersprachen.

In allen imperativen und objektorientierten Sprachen sind sie in unterschiedlichen Ausprägungen vorhanden.

Leistungsfähige Schleifenkonstrukte (zusammen mit Unterprogrammmethoden) sind essentielle Konstituenten der **strukturierten Programmierung** (dritte Programmiersprachen-Generation) und auch der **Objektorientierten Programmierung**.

Verzweigungen (Prinzip)

- ▶ Eine **wertabhängige** Fallunterscheidung ist in der Informatik für viele Art von Algorithmen elementar. (Was ist der minimale Befehlssatz um Turing-vollständig zu sein?)
- ▶ Aufgrund einer Bedingung wird der Programmfluss (die Abfolge der Ausführung der Befehle) verzweigt. Wir unterscheiden die **einfache (bedingte) Verzweigung** die den Programmfluss in **zwei** Pfade auftrennt und eine **mehrfache Verzweigung**.

Die mehrfache Verzweigung (Prinzip)

Realisiert ist dies in den meisten Programmiersprachen sehr ähnlich, ungefähr folgendermaßen:

```

if <Bedingung>           # Einfachverzweigung
  then <Aktionsfolge>
  else <AlternativeAktionsfolge>
endif

oder
case <aVariable>          # Mehrfachverzweigung
  aValue1: <AktionsfolgeA>;
  aValue2: <AktionsfolgeB>;
  ...
  otherwise: <Aktionsfolge-sonst>
endcase ;
  
```

Graphische Repräsentationen

Zur Darstellung von Programmstrukturen wurden schon sehr früh graphische Repräsentationen (visuelle Sprachen) eingesetzt, die später auch genormt wurden:

- Programmablaufpläne (und Datenflusspläne) nach DIN 66001
- Nassi-Schneidermann Diagramme nach DIN 66261
- Jackson-Diagramme
- UML Diagramme (hier allein 14 verschiedene Diagrammartentypen, kommen teilweise später)

Programmablaufplan

ist ein **Ablaufdiagramm** für ein Computerprogramm, das auch als *Flussdiagramm* (engl. *flowchart*) oder *Programmstrukturplan* bezeichnet wird.

Das Konzept der Programmablaufpläne stammt aus den 1960er-Jahren.

... aber es zeigt das Vorgehen doch sehr anschaulich!

Verzweigung: einfach-einseitig

Pseudocode	Ablaufplan	Python
<pre>if <Bedingung> then <Block > endif;</pre>	<pre>graph TD A{?} -- wahr --> B[Block] A -- falsch --> C(()) B --> D(()) C --> D D --> E[]</pre>	<pre>if <expression> then <suite></pre>

in Python

Bedingung = ? : ➔ *expression* (ein Ausdruck, der sich zu *bool* auswerten lässt (und das ist fast Alles))

Block : ➔ *suite*

Zweifach- und Mehrfach-Verzweigungen

Pseudocode	Ablaufplan	Python
<pre>if <Bedingung> then <Block 1>; else <Block 2>; endif;</pre>	<pre>graph TD Start(()) --> Decision{?} Decision -- wahr --> Block1[Block 1] Decision -- falsch --> Block2[Block 2] Block1 --> Merge(()) Block2 --> Merge Merge --> End(())</pre>	<pre>If <expression>:: <suite1> else: <suite1></pre>
<pre>case <Variable> Wert 1: <Block0> Wert 2: <Block1>... otherwise: <Bn> endcase;</pre>	<pre>graph TD Start(()) --> Decision{Variable =} Decision -- Wert 1 --> B1[B1] Decision -- Wert 2 --> B2[B2] Decision -- sonst --> Bn[Bn] B1 --> Merge(()) B2 --> Merge Bn --> Merge Merge --> End(())</pre>	<pre>If <expression>:: <suite1> elif <expression1> <suite2> ... else: <suite n></pre>

11

Vorlesung PRG 1 – V4
Kontrollstrukturen

Prof. Dr. Detlef Krömer

Zwischenruf: Kode-Strukturierung durch „Blöcke“

- Zum ersten Mal wurden Blockstrukturen (oder suites) schon in ALGOL (1958) verwendet.
- Die Notationen sind in den Programmiersprachen verschieden:
 - begin ... end
 - if ... fi
 - do ... done
 - { ... }
- In **Python** heißen Blöcke „suite“ und werden **durch Einrückung** kenntlich gemacht.

12

Vorlesung PRG 1 – V4
Kontrollstrukturen

Prof. Dr. Detlef Krömer

Wir benutzen laut Style Guide Einrückung durch 4 Spaces (Blanks)



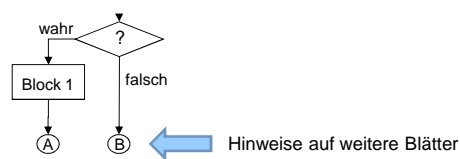
From [emacswiki](#) licensed under [Creative Commons ShareAlike](#).

Programmierhandbuch (ein Hinweis auch schon für die EPR 3)

- ▶ **Ab der Aufgabe EPR 3** müssen Sie entsprechend der Regeln im Programmierhandbuch arbeiten, sonst gibt es Punktabzug.
- ▶ Diese sind im sogenannten **Programmierhandbuch (Style Guide)** bekannt gemacht, siehe Moodle.
Das müssen Sie **unbedingt lesen** und beachten.
- ▶ In Moodle finden Sie eine Vorlage für den Header `header.py`, den Sie am besten für sich anpassen und dann abspeichern und immer wieder benutzen.

Kritik an Programmablaufplänen

- schon bei mittelgroßen Algorithmen schnell unübersichtlich ... viele, viele Seiten
- Verführt zur Verwendung von expliziten Sprunganweisungen (GOTO's) und damit die Produktion von "Spaghetti-Code" fördern und




Programmablaufpläne: Kritik und Nutzung

- gut strukturierter Programmcode ist genauso übersichtlich (oder gar übersichtlicher!)
- Korrigiert man z.B. einen Fehler in einem Programm müsste dieses ggf. im Ablaufplan „nachgezogen“ werden
- Programmablaufpläne werden heute in der Softwareentwicklung **nicht mehr eingesetzt**.
- Sie erleben allerdings in etwas modifizierter Form eine Renaissance in Multimedia-Entwicklungsprozessen
- sind gut geeignet, elementare Strukturen der Programmierung zu verdeutlichen.

Realisierung in Python if – elif – else

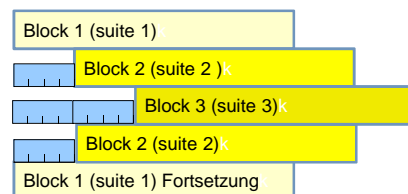
```

if expression:
    suite
elif expression:
    suite
elif expression:
    suite
...
else:
    suite
    
```

 indent
 4 Blanks (Leerzeichen) oder
 1 Tab(ulator) **laut Programmier-richtlinien nicht verwenden!**

„expression“ ist hierbei ein Ausdruck, der sich zu einem booleschen Wert auswerten lässt.
 „suite“ kennzeichnet eine beliebige Anweisungsfolge (das ist ein Block).

In Python: Strukturierung durch „Einrücken“



Achtung: Eingeleitet wird ein Block (suite) immer durch ein “:”
 Übrigens: In Python ist eine **sonstige Nutzung verboten** und führt zum:
SyntaxError: unexpected indent

Beispiel

```
a = 3
b = 2
if a < b:
    pass
elif a == b:
    a += 1
else: a -= 1
print(a)
```

```
>>> ===== RESTART =====
>>>
2
```

Was ist hierzu entscheidend wichtig? – Vergleiche, die sich zu True oder False auswerten lassen

Operator	Beschreibung
X < Y	echt kleiner als
X <= Y	kleiner oder gleich als
X > Y	echt größer als
X >= Y	größer oder gleich als
X == Y	gleicher Wert wie
X != Y	Ungleicher Wert wie
X is Y	Gleiches Objekt (Variable)
X is not Y	Negierte Objektgleichheit

Das Ergebnis einer dieser Vergleichsoperatoren ist immer
ein Boolean: True oder False.

„Specials“ mit Boolean: True oder False

(noch nicht versuchen, dass zu verstehen – aber nicht mit hereinfallen!)

Boolean ist ein **Untertyp von Integer**:

- = 0 oder leer bedeutet False
- 0 bedeutet True

... Konsequenzen:

```
>>> True + True
2
>>> True - False
1
>>> True * False
0
```

*Tricky oder? ...Vergessen Sie es!
Ich hab's nie gesagt! ;-)*

Alle Expressions (= Ausdrücke) können zu einem Boolean ausgewertet werden!

Sie können alle Ausdrücke in einem boolean context benutzen und Python wird einen Wahrheitswert berechnen.

In einigen Situationen (wie in **if statements**), erwartet Python ein Ergebnis True oder False. Dieses nennt man *boolean context*. Hier erfolgt eine **coercion**!

Zum Schluss noch ein "Bonbon":

Conditional Expressions (1)

Nennt man auch conditional operator, inline if (iif), oder ternary if ... findet man in fast jeder modernen Programmiersprache.

Vergleichsweise häufig kommt folgende Abfragestruktur vor:

```
if condition:
    x = true_value
else:
    x = false_value
```

Conditional Expressions (2)

Dies kann in Python auch wie folgt formuliert werden:

```
x = true_value if condition else false_value
```

... und ist damit ein Einzeiler. **Aufpassen: keine Doppelpunkte!** –

- **else-Zweig** muss immer vorhanden sein.
- **Operator Präzedenz** geringer als **or**

Zuerst wird `condition` ausgewertet, dann abhängig vom Ergebnis `true_value` oder `false_value`.

Zwischen-Zusammenfassung

- Mit `if` – `elif` – `else` sind alle Verzweigungstypen:
 - einfach
 - zweifach
 - Mehrfach realisierbar.
- Eine lange Abfragekette `i(elif)` ist wenig elegant.
- Ggf. an die Conditional Expressions denken!

Übersicht

1. Fallunterscheidungen: Verzweigungen

Prinzipien
Graphische Repräsentationen
Realisierung in Python: if – else – elif

2. Iterative Grundstrukturen

Prinzipien
Schleifen: Realisierungsformen der Iteration
Realisierungen in Python:
for und while
break – continue – else

Iterative Grundstrukturen (1)

Die **Iteration** (von lateinisch *iterare*, "wiederholen"; engl. *iteration*) ist ein grundlegender Lösungsansatz sowohl in der Mathematik als auch der Informatik mit zwei verschiedenen Ausprägungen:

1. **Iteration (in Mathematik und Informatik)** ist eine Methode, sich der Lösung eines „Rechenproblems“ schrittweise, **aber zielgerichtet** anzunähern. Sie besteht in der **wiederholten Anwendung desselben Rechenverfahrens**.

Meistens iteriert man mit Rückkopplung: Die Ergebnisse eines Iterationsschrittes (oder alle bisher erzielten Ergebnisse) werden als Ausgangswerte des jeweils nächsten Schrittes genommen - bis das Ergebnis zufrieden stellt.

Iteration (Fortsetzung)

- Dazu muss man sicher sein (z.B. beweisen!), dass die Iterationsfolge **konvergiert** und dass der Grenzwert mit der gesuchten Lösung übereinstimmt.
- Die Geschwindigkeit der Konvergenz ist ein Maß dafür, wie brauchbar die Iterationsmethode ist.
- Z.B. wird ein iteratives Lösungsverfahren dann eingesetzt, wenn das Ergebnis **nicht geschlossen berechenbar** ist (zum Beispiel Gleichungen mit transzendenten Funktionen: $\sin x + \cos x = x$ oder Bestimmung der Nullstellen ab dem Polynomgrad 5, etc.).
- Häufig ist eine gute Näherung schon befriedigend.

Beispiele

- Summen
$$a = \sum_{i=0}^n a_i = a_0 + a_1 + a_2 + \dots + a_n$$
- Produkte
$$a = \prod_{i=0}^n a_i = a_0 \cdot a_1 \cdot a_2 \cdot \dots \cdot a_n$$
- Regula Falsi Nullstellenberechnung bei Polynomen

Iterative Grundstrukturen (2)

2. In der Informatik wird auch von Iteration gesprochen, wenn ein **schrittweiser Zugriff auf Einzelemente eines zusammengesetzten Datentyps (Sammlungen, in Python Sequenztypen)**, z.B. auf jedes Zeichen eines Strings, erfolgt.

Z.B. wird wiederholt auf einen String mit veränderten Indexwert zugegriffen: in einem String "otto" sollen alle Kleinbuchstaben durch Großbuchstaben ersetzt werden: "OTTO".

Schleifen: Realisierungsformen der Iteration

In Programmiersprachen werden iterative Lösungen beider Art **durch Schleifen realisiert**:

Eine **Schleife** ist eine Kontrollstruktur in imperativen Programmiersprachen.

Sie wiederholt einen Teil des Codes – den so genannten **Schleifenrumpf** oder **Schleifenkörper** – so lange, bis eine **Abbruchbedingung** eintritt.

Schleifen, die ihre Abbruchbedingung niemals erreichen oder Schleifen, die keine Abbruchbedingungen haben, nennen wir **Endlosschleifen**.

Endlosschleifen

können nur von „außen“ unterbrochen werden durch
ein Zurücksetzen (engl. *reset*), durch eine Unterbrechung (engl. *interrupt*),
durch Ausnahmen (engl. *exceptions*),
Abschalten des Gerätes oder ähnliches
(Achtung, sind dann streng genommen kein Algorithmus)

Oft, aber nicht immer, ist eine Endlosschleife ein Programmierfehler, weil das Programm nicht normal beendet werden kann.

Ist die Aufgabe des Programms jedoch z.B. eine Überwachung und Reaktion auf einen externen (gemeldet z.B. durch einen Interrupt) oder internen Fehlerzustand (gemeldet durch eine Exception), so kann dieses Verhalten ggf. gewollt sein! Grundsätzlich aber **VORSICHT!**

Schleifenarten (Grundformen)

- die **kopfgesteuerte** oder **vorprüfende** Schleife, bei der erst die Abbruchbedingung geprüft wird, bevor der Schleifenrumpf durchlaufen wird (meist durch das Schlüsselwort **WHILE** (= *solange-bis*) angezeigt.
- die **fußgesteuerte** oder **nachprüfende** Schleife, bei der nach dem Durchlauf des Schleifenrumpfes die Abbruchbedingung überprüft wird, z.B. durch ein Konstrukt **REPEAT-UNTIL** (= *wiederholen-bis*).
- die **Zählschleife**, eine Sonderform der kopfgesteuerten Schleife, meist als **FOR** (= *für*) -Schleife implementiert.
- die **foreach-Schleife**, eine Sonderform für Sequenzdatentypen: Meint: „für jedes Element in der Sequenz führe „Block“ genau einmal aus.“

Schleifen: Realisierungsformen der Iteration (1)

Schleifenart	Ablaufplan
vorprüfend (Kopfgesteuert) while (B) do Block	<pre> graph TD Entry(()) --> B{B} B -- wahr --> Block[Block] Block --> B B -- falsch --> Exit(()) </pre>
nachprüfend (Fußgesteuert) repeat Block until (B)	<pre> graph TD Entry(()) --> Block[Block] Block --> B{B} B -- wahr --> Entry B -- falsch --> Exit(()) </pre>

✓ **in Python**
„Block“ wird ggf.
nie ausgeführt.

Hier wird „Block“
mindestens
einmal ausgeführt

Schleifen: Realisierungsformen der Iteration (3)

Schleifenart	Ablaufplan
foreach - Schleife for <Iterativvariable> in sequenz do Block Meint: „für jedes Element in der Sequenz führe „Block“ genau einmal aus.“	<pre> graph TD Entry(()) --> B{next Item from sequenz} B -- leer --> Exit(()) B --> Block[Block] Block --> B </pre>

✓ **in Python**
Wir kennen bisher
erst einen
Sequenzdatentyp:
String, aber es
kommen noch
einige hinzu.

Schleifen: Realisierungsformen der Iteration (2)

Schleifenart	Ablaufplan
Zählschleife for(C=1;B;C++)do Block Meint: "Die Schleifenvariable C beginnt bei 1 und läuft durch Inkrementieren mit 1 bis B."	<pre> graph TD Start(()) --> C1[C=1] C1 --> B{B} B -- falsch --> End(()) B -- wahr --> Block[Block] Block --> Cinc[C++] Cinc --> B </pre>

✓ in Python mit
for - Schleife und
zusätzlicher Funktion
range ()

Schleifen in Programmiersprachen

Bis auf die rein funktionalen Programmiersprachen (diese haben überhaupt keine Schleifen) realisieren **alle** modernen Programmiersprachen eine Auswahl der hier dargestellten Grundstrukturen.

Sie unterscheiden sich in den benutzten Schlüsselwörtern, der Art der Klammerung der Programmblöcke {}, begin ... end, etc.) und des Typs der Laufvariablen sowie deren „Inkrementierung“.

Mit Python werden wir sehr leistungsfähige Varianten kennenlernen:
while-Schleife, foreach-Schleife und die range-Funktion

Historische Notiz (1)

Noch in den 60er-Jahren waren Sprunganweisungen (GOTO <Sprungziel>) in Programmen üblich, was bei größeren Programmen nahezu zur Unwartbarkeit führte, da sie schnell kaum noch überschaubar wurden.

Das „GOTO <Sprungziel>“ ist eine direkte Abbildung des Maschinenbefehls „JUMP <Adresse des Sprungziel>“ – lediglich musste das Sprungziel jetzt keine Programmadresse mehr sein, sondern konnte symbolisch als Zahl oder Name angegeben werden.

Historische Notiz (2)

Schon im Mai 1966 publizierten **Böhm und Jacopini** einen Artikel, in dem sie zeigten, dass jedes Programm, das GOTO-Anweisungen enthält, in ein GOTO-freies Programm umgeschrieben werden kann, das nur mit Verzweigung (IF <Bedingung> THEN ... ELSE ...) und einer Schleife (WHILE <Bedingung> DO xxx) arbeitet (gegebenenfalls unter Zuhilfenahme von etwas Kodeduplikation und der Einführung von booleschen Variablen (true/false))

Im März 1968 veröffentlichte *Edsger W. Dijkstra* seinen legendären Aufsatz „**Go To Statement Considered Harmful**“ (Dieser kleine Aufsatz ist das **READING nächster Woche!**)

Ein Beispiel für so genannten „Spaghetti-Code“

mit GOTO-Anweisungen:

```
GOTO 40
20 UmgeheDasProblem
   GOTO 70
40 if (Durcheinander < TötlicheDosis) then GOTO 60
   GOTO 20
60 RetteJungfrau
70 ...

IF (Durcheinander < TötlicheDosis)
    THEN RetteJungfrau
    ELSE UmgeheDasProblem
```

Kontrollstrukturen in Python

while-Anweisungen = kopfgesteuerte Schleife

```
while_stmt ::= "while" expression ":" suite
             ["else" ":" suite]
```

(Dies ist übrigens eine Syntaxdefinition in erweiterter BNF, direkt lesbar, oder? - Betrachten wir später noch)

Das **“while-Statement”** benutzt man zur wiederholten Ausführung des Code-Blöcken (suite), solange die “expression” zu “True” ausgewertet wird.

Wenn die “expression” zu “False” ausgewertet wird, wird die “suite” des else-Zweigs ausgeführt, sofern vorhanden.

In suite muss der „Wahrheitswert“ von expression irgendwann geändert werden, sonst Gefahr einer Endlosschleife.

Beispiel Endlosschleife

```
while True:
    pass
```

```
Traceback (most recent call last):
  File "<pyshell#100>", line 2, in <module>
    pass
KeyboardInterrupt
```

Was macht man?

Rechner herunterfahren
und neu booten? NEIN!

„Control-C“ drücken, das ist
der `KeyboardInterrupt`

Menuepunkt „Restart“

In Python for- Anweisung (1)

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
          ["else" ":" suite]
```

Das for_stmt kann relativ kompliziert werden: Wir betrachten zunächst einen einfachen Fall, die **Variante 1** (for each Schleife):

- In `target_list` steht als target ein **Name** (identifizier) **als sogenannte Schleifenvariable**
- In `expression_list` muss dann stehen: ein iterierbarer Typ (ein iterable – alle Sequenztypen: List, Tuple, String) aber auch Sets, Frozensets, Dictionaries. kurz alle Objekte, die ihre Elemente eins nach dem anderen zurückgeben können und dafür die Methode `__next__()` haben

In Python for- Anweisung - Ablauf

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
          ["else" ":" suite]
```

Der Schleifenvariablen **target** wird vor dem ersten Durchlauf von **suite** das erste Element des iterierbaren Typs zugewiesen.

Wenn **suite** durchgelaufen ist, wird **target** das folgende Element aus dem iterierbaren Typ zugewiesen, usw. bis alle Elemente behandelt wurden.

Dann wird (falls vorhanden) die else-suite einmal ausgeführt.

Dies ist also eine **foreach** Schleife.

In Python for- Anweisung - Beispiel

```
for letter in "Python":
    print (letter)
else: print("ist topp!")
```

erzeugt

```
P
y
t
h
o
n
ist topp!
```

Ein Hinweis für die Zukunft

Durch Teilbereichsbildung (*Slicing*), die zusammenhängende Bereiche einer Sequenz extrahiert: Die Bereichsgrenzen sind mit 0 und der Sequenzlänge vorbelegt:

- `S[1:3]` geht von Index 1 bis ausschließlich 3 (also 2)
- `S[:]` geht vom Index 0 bis zum Ende; `len(S)`
- `S[::-1]` nimmt alles bis auf das letzte Element (negative Schrittweite zählen die Indizes vom Ende).
- **Durch Slicing kann man nicht nur den Teilbereich auswählen sondern auch die Bearbeitungsreihenfolge anpassen.**

Beispiele zur for-Schleife (Zukunft)

```
>>> for x in 'hallo':
    print(x)
```

```
h
a
l
l
o
```

```
>>> for x in 'hallo'[::-1]:
    print(x)
```

```
o
l
l
a
h
```

So wird die „foreach-Schleife“ zur Zählschleife Die Funktion range()

Die foreach-Schleife wird im Zusammenhang mit der **range-Funktion** (als virtuelle „expression“) zu einer **Zählschleife**

```
range ([start,]stop [,step])
```

```
for x in range ([start,]stop [,step]):
```

interessant. **range** liefert (virtuell) eine Sequenz von aufeinander folgenden Integern zwischen *start* und *stop-1*.

Mit nur einem Parameter (*stop*) ergeben sich die Zahlen **0...stop-1**.

step ist optional und ist die Schrittweite – kann negativ sein. Verhält sich aber anders als das Slicing! – Sorry.

Beispiele zur Zählschleife mit range([start,]stop [,step])

```
>>> for x in range(0,5,2):  
    print(x)
```

```
0  
2  
4
```

```
>>> a = '0123456789'  
>>> for x in a[0:5:2]:  
    print(x)
```

```
0  
2  
4
```

Die Regeln für das Slicing und für die Parameter der range()-Funktion sind sehr ähnlich

aber: Slicing: Bereichsgrenzen durch **:** getrennt!
range (Bereichsgrenzen durch **,** getrennt

Die Funktion `range()` ist eine spezielle Funktion, die einen "Iterator" erzeugt, eine "virtuelle Sequenz"

- Der Iterator muss ja mit `__next__()` nur das nächste Element der virtuellen Sequenz liefern und dann, wenn keins mehr vorhanden ist eine `StopIteration` exception (Ausnahme)
- Das kann auch ein Programm, man muss also die Sequenz nicht vollständig erzeugen und speichern, sondern man kann dies on-demand liefern.

Solche Programme kann man auch selbst schreiben.

Aber da führte jetzt zu weit!

Beispiele zur Zählschleife mit `range([start,]stop [,step])` und Slicing

```
>>> for x in range(6,-5,-2):
      print(x)
```

```
6
4
2
0
-2
-4
```

```
>>> a = '0123456789'
>>> for x in a[6:-5:-2]:
      print(x)
```

```
6
```

... aber eben doch auch verschieden: AUPASSEN!

Ausblick (denn das führt hier zu weit): Als target kann auch eine target_list stehen (behandeln wir im Zusammenhang mit Composite Strukturen)

```
for_stmt ::= "for" target_list "in" expression_list ":" suite
          ["else" ":" suite]
```

```
namensliste = [('Max', 'Meyer'), ('Marlene', 'Müller')]

for vorname, nachname in namensliste:
    print(vorname, nachname)
```

erzeugt:

Max Meyer
Marlene Müller

- Die Anzahl der Elemente in **expression list** muss genau der in **target-list** entsprechen.
 - Die Anzahl der Elemente in den expressions muss gleich groß sein.
- ... wir sollten zuerst die Containertypen behandeln.

„Abbrechen“ von Schleifen

Um eine **Schleife** (while oder for) ohne Erreichen der Bedingung (expression == False) zu **beenden**, verwendet man die

- **break**-Anweisung.

Um in die **nächste Schleifeniteration zu springen** (den Rest des Schleifenrumpfes (der suite) zu überspringen) verwendet man die

- **continue**-Anweisung.

Ein „Nichtstuer“

- **pass**-Anweisung
- Macht garnichts – geht zur nächsten Anweisung weiter
- Kann benutzt werden, wenn die Syntax eine Anweisung erfordert, doch die Programmlogik benötigt keine Aktion.
- Während der Code-Entwicklung als Platzhalter
(`# remember to implement this`).

Zusammenfassung

Wieder viel Stoff: Kontrollstrukturen: **Verzweigungen und Schleifen**
(ermöglichen die Iteration)

Kontrollstrukturen verändern den **Programmfluss** (nicht mehr nur eine Abfolge)

Wir haben diese Konzepte eingeführt

... Die Praxis, das Programmieren damit müssen Sie jetzt üben! – Sorry.

An die Abgabe des **EPR_01** spätestens morgen denken!

Auf jeden Fall die zwei Quizzes machen:

- Verzweigungen
- Schleifen

Vielleicht auch Fehler melden, wenn Sie welche entdecken!

Ausblick ... Die nächsten drei Wochen

Montag: V04 Datenstrukturen Integer, Bool, None
Freitag: V05 Datenstruktur String

Montag: V07 Kontrollstrukturen –
Prozeduren und Funktionen – import
Freitag: V06 Datenstruktur Float

Montag: V08: Aggregierte Datentypen in Python (Builtins)
Freitag: V09 Iteration vs. Rekursion

... und, danke für Ihre Aufmerksamkeit!