Einführung in die Programmierung

Andreas Hildebrandt

Kontrollflussmanipulation mit if

- Python bietet viele Möglichkeiten, den Kontrollfluss zu manipulieren
- Werden wir nun nach und nach kennenlernen
- Einfachstes Mittel zur Kontrollflussmanipulation: **bedingte Anweisungen** (**Conditionals**)
- **Ziel:** zusammenhängende Gruppe (**Block**) von Statements nur ausführen, wenn bestimmte **Bedingung** (**Condition**) erfüllt ist
- Dazu dient **if** - Statement
- **Verwendung:**

```
if <Boolescher Ausdruck>:
    <statement>
    <statement>
    ...
```

- **"Template-Notation:"** Terme in spitzen Klammern stehen für beliebige Instanzen von Booleschen Ausdrücken bzw. Anweisungen

Kontrollflussmanipulation mit if

```
if <Boolescher Ausdruck>:
  <statement>
  <statement>
  ...
```

- **Hinweise:**
  - if selbst muss gleiche Einrückung haben wie umgebender Block
  - nach Booleschem Ausdruck **muss** Doppelpunkt stehen
  - Beliebig viele Anweisungen beliebiger Art im abhängigen Block
  - Zuordnung zu abhängigem Block durch Einrückung:
    - muss für alle abhängigen Statements **gleich** sein
    - muss **größer** (weiter rechts) sein als Einrückung des if
  - Beliebig viele Statments im Block erlaubt, mindestens ein Statement benötigt
  - Nach abhängigem Block wieder ursprüngliche Einrückung

Kontrollflussmanipulation mit if

- **Beispiel:**

In [3]:
```
#%%tutor -b http://localhost:8003

x = -42 # Hier als Beispiel; könnte z.B. Resultat von input() sein

if x < 0:
    x = -x
```

- Berechnet Betrag $|x|$ der Variablen x:
  - Falls Wert von x positiv ist muss nichts getan werden
  - Falls Wert von x negativ ist wird mit -1 multipliziert

Kontrollflussmanipulation mit if

- **Weitere Beispiele:**

In [4]:
```
#%%tutor -b http://localhost:8003

x = 4711
y = 42   # hier als Beispiel; könnten z.B. Resultate von input() sein

if x > y:
    temp = x
    x = y
    y = temp
```

- Sortiert x und y so, dass in jedem Fall $x \leq y$ gilt:
  - Falls x schon kleiner gleich y muss nichts getan werden
  - Falls x größer y:
    - Erzeuge neue Variable temp und binde an Wert, auf den x zeigt
    - Binde x an Wert auf den y zeigt
    - Binde y an Wert auf den temp zeigt (= alter Wert von x)

Kontrollflussmanipulation mit if

- **Hinweis:** bei mehrzeiligen abhängigen Blöcken ist Einrückung entscheidend!

In [5]:
```
x = 4711
y = 42   # hier als Beispiel; könnten z.B. Resultate von input() sein

# korrekt
if x > y:
    temp = x
    x = y
    y = temp

# inkorrekt
if x > y:
    temp = x
x = y
y = temp
```

Flowcharts

- Verwzeigungen können beliebig verschachtelt und mit anderen Sprachmitteln kombiniert werden
- Oft nützlich: graphische Darstellung
- Geht auf verschiedene Arten; wir verwenden hier **Flowcharts** (**Flussdiagramme**)
- **Idee:** Programm als Graph dargestellt
  - Anweisungen stehen in Knoten
  - Verzweigungen repräsentiert durch Kanten
- Oft verschiedene Formen für verschiedene Anweisungsarten verwendet, z.B.
  - Runde Knoten für Start und Ende
  - Schräge Kästen für I/O
  - Rauten für Verzweigungen
  - Kästen für alles andere

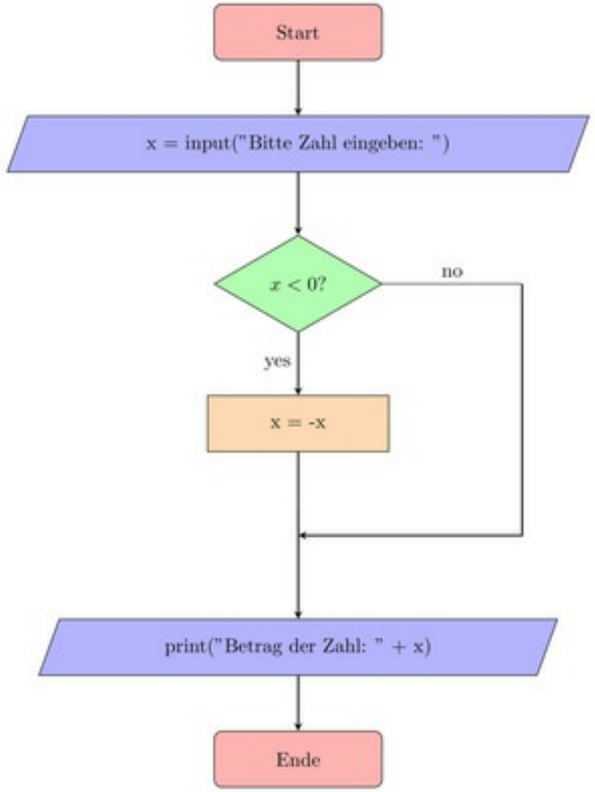
Flowcharts

- **Beispiel:**

```
x = input("Bitte Zahl eingeben: ")

if x < 0:
    x = -x

print("Betrag der Zahl: " + x)
```



if-then-else

- Oft benötigt: unterschiedliche Anweisungsblöcke je nach Wert eines Booleschen Ausdrucks
- Könnten wir erreichen mit:

```
if condition:
    <statement>
    <statement>
    ...
if not condition:
    <statement>
    <statement>
    ...
```

- **Idee:**
  - condition ist entweder True oder False (*tertium non datur*)
  - im ersten Fall wird abhängiger Block des ersten if ausgeführt
  - im zweiten Fall wird abhängiger Block des zweiten if ausgeführt

⇒ es wird immer **genau ein** abhängiger Block ausgeführt, nie keiner, nie zwei

if-then-else

- Konstrukt wird so oft benötigt, dass es von Python direkt angeboten wird
- **Schema:**

```
if <Boolescher Ausdruck>:
    <condition>
    <condition>
    ...
else:
    <condition>
    <condition>
    ...
```

- erster abhängiger Block wird auch als **then**-Block bezeichnet, zweiter als **else**-Block

if-then-else

- **Beispiel:**

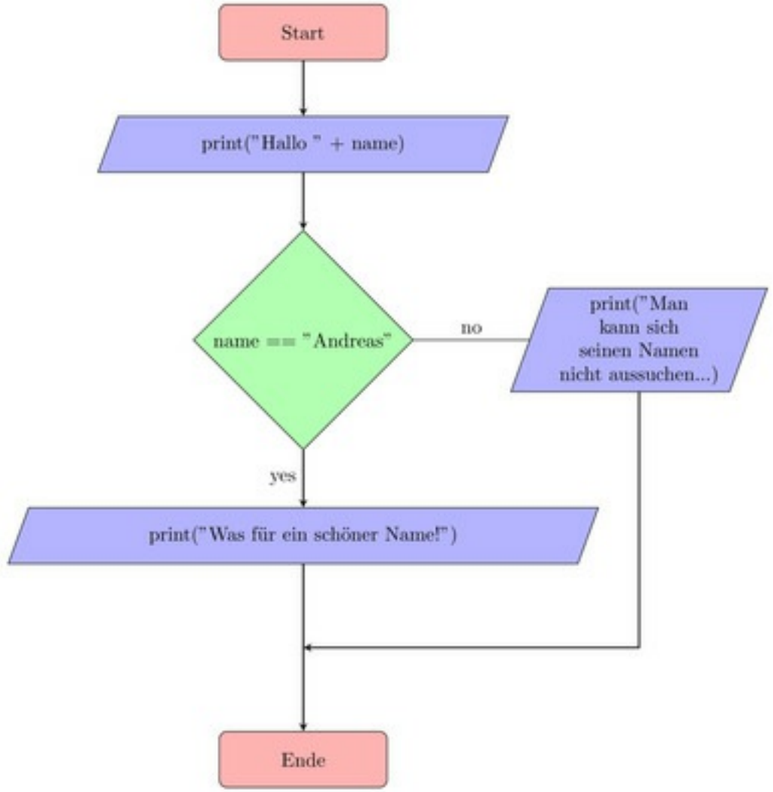
In [6]:
```
#%%tutor -b http://localhost:8003

name = "Andreas"

print("Hallo " + name)

if (name == "Andreas"):
    print("Was für ein schöner Name!")
else:
    print("Man kann sich seinen Namen nicht aussuchen...")
```

```
Hallo Andreas
Was für ein schöner Name!
```


Start
print("Hallo " + name)
name == "Andreas"
no
print("Man kann sich seinen Namen nicht aussuchen...)
yes
print("Was für ein schöner Name!")
Ende

if-then-else

- **Beispiel:**

```
if x > y:
    maximum = x
else:
    maximum = y
```

- Berechnet maximum der beiden Zahlen x und y

if-then-else

- **Beispiel:** Schaltjahrberechnung
  - Falls Jahr durch 4 teilbar ist, aber nicht durch 100 ist es ein Schaltjahr
  - Falls Jahr durch 4 und 100, aber nicht durch 400 teilbar ist, ist es kein Schaltjahr
  - Falls Jahr durch 4, 100 und 400 teilbar ist, ist es ein Schaltjahr

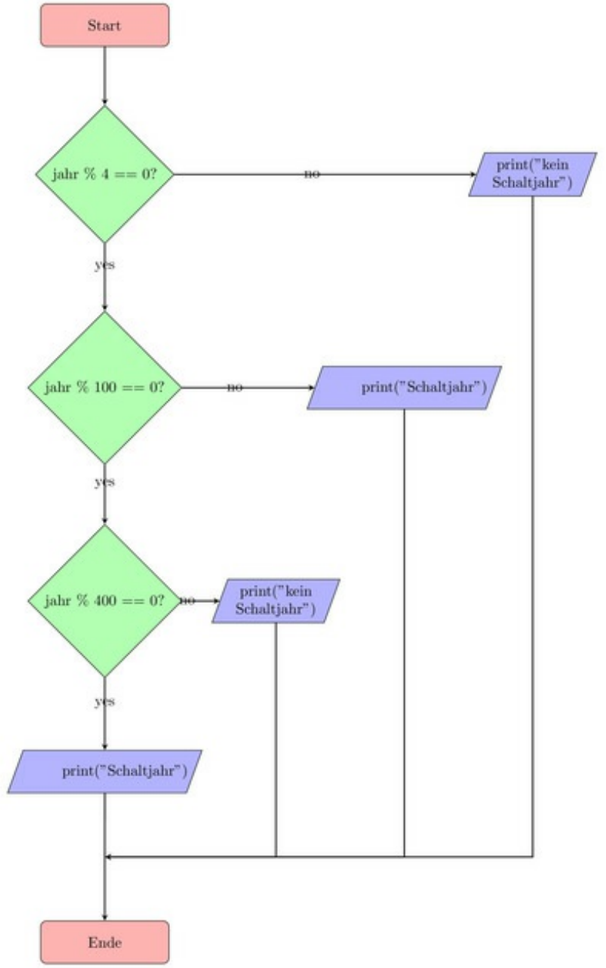
In [7]:
```
jahr = 2016

if jahr % 4 == 0: # Rest bei Division durch 4 ist 0 => teilbar durch 4
    if jahr % 100 == 0: # teilbar durch 100
        if jahr % 400 == 0: # teilbar durch 400
            print("Schaltjahr")
        else:
            print("Kein Schaltjahr")
    else:
        print("Schaltjahr")
else:
    print("Kein Schaltjahr")
```

```
Schaltjahr
```

if-then-else

- Falls Jahr durch 4 teilbar ist, aber nicht durch 100 ist es ein Schaltjahr
- Falls Jahr durch 4 und 100, aber nicht durch 400 teilbar ist, ist es kein Schaltjahr
- Falls Jahr durch 4, 100 und 400 teilbar ist, ist es ein Schaltjahr



if-then-else

- **Beispiel:** Lösung der quadratischen Gleichung $ax^2 + bx + c$ mit $a \neq 0$
  - Lösungen bekannt aus der Schule:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

  - Reelle Lösungen für $b^2 - 4ac > 0$

```
import math
discriminant = b**2 - 4.0*a*c

if discriminant >= 0:
    sqrt_d = math.sqrt(discriminant)
    print("x_1: " + str((-b + sqrt_d) / (2.0*a)))
    print("x_2: " + str((-b - sqrt_d) / (2.0*a)))
else:
    print("Keine reellen Wurzeln!")
```

if-then-else

- **Alternativ:** Lösung der quadratischen Gleichung $ax^2 + bx + c$ mit $a \neq 0$ über den komplexen Zahlen

In [8]:
```
import cmath

a = 2
b = 1
c = 1

discriminant = b**2 - 4.0*a*c
sqrt_d = cmath.sqrt(discriminant)
print("x_1: " + str((-b + sqrt_d) / (2.0*a)))
print("x_2: " + str((-b - sqrt_d) / (2.0*a)))
```

```
x_1: (-0.25+0.6614378277661477j)
x_2: (-0.25-0.6614378277661477j)
```

if-then-else

- Eher selten genutzte **Sonderregel:**

  enthält abhängiger Block von if oder else nur eine Anweisung, kann diese in gleicher Zeile wie if bzw. else stehen

- **Beispiel:**

```
if x > y: maximum = x
else:     maximum = y
```

- Nur dann einsetzen, wenn dadurch Code lesbarer wird

Kurzer Exkurs: (Pseudo-)Zufallszahlen

- Oft gebraucht: zufällig gezogene Zahlen
- Wirklicher Zufall auf dem Rechner schwer zu erreichen
- Statt dessen oft **Pseudozufallszahlen** (**pseudo-random numbers**) - Folge von Zahlen, die
  - **deterministisch** bestimmt wird...
  - ...aber viele Eigenschaften von Zufallszahlen hat (*"zufällig aussieht"*)
- Generierung verwendet oft einen Startwert, der vom Benutzer vorgegeben werden kann
- Nicht-Determinismus dann erreichbar, indem z.B. mit Uhrzeit gestartet wird

Kurzer Exkurs: (Pseudo-)Zufallszahlen

- Python liefert Pseudozufallszahlgeneratoren (PRNGs) im Modul random mit
- Werden standardmässig (="**per Default**") mit aktueller Systemzeit intialisiert ⇒ neue Werte bei jedem Programmstart
- Verschiedene Funktionen enthalten, für uns im Moment nützlich:

  random.randrange(start, stop)

  - Liefert gleichverteilt zufällig gezogenes int...
  - ...im halboffenen Intervall $[start, stop)$ (d.h. zwischen $start$ und $stop - 1$)

Kurzer Exkurs: (Pseudo-)Zufallszahlen

- **Beispiel:** Simulation des Wurfes einer fairen Münze

In [9]:
```
import random

if random.randrange(0, 2) == 0:
    print("Kopf")
else:
    print("Zahl")
```

```
Kopf
```

if-then-else

- if-Konstrukte schnell tief verschachtelt
- **Beispiel:** Berechnung der Note basierend auf Punktzahl für hypothetische Notenskala

In [10]:
```
points = 85

if points < 40:
    note = 5
else:
    if points < 50:
        note = 4
    else:
        if points < 60:
            note = 3
        else:
            if points < 70:
                note = 2
            else:
                note = 1
```

if-then-else

- Verschachtelte Alternativen so häufig, dass Python Kurzschreibweise anbietet (syntaktischer Zucker)

In [11]:
```
points = 85

if points < 40:
    note = 5
elif points < 50:
    note = 4
elif points < 60:
    note = 3
elif points < 70:
    note = 2
else:
    note = 1
```

Schleifen

- Können mit if Programme schreiben, die ohne Verzweigungen nicht möglich wären
- **Aber:** Rechner wird von uns bislang kaum ausgenutzt
- Was bringt Performance von 300 Billionen (300×10^{12}) Operationen pro Sekunde, wenn wir jede Anweisung einzeln eingeben müssen?
- **Schleifen** (**Loops**) dienen dazu, Blöcke von Anweisungen wiederholt auszuführen
- Mehrere Schleifentypen in Python umgesetzt
- Betrachten zunächst die **while-Schleife** mit Python-Schema:

```
while <Boolescher Ausdruck>:
    <statement>
    <statement>
    ...
```

Schleifen

```
while <Boolescher Ausdruck>:
    <statement>
    <statement>
    ...
```

- Schema ganz analog zu if-Anweisung, nur Schlüsselwort if durch while ersetzt
- **Bedeutung:** führe Block von Anweisungen **solange** aus, wie Boolescher Ausdruck Wert True lieft
- **Hinweise:**
  - erste Zeile nennt man **Schleifenkopf**, den Rest **Schleifenrumpf**
  - jede Wiederholung nennt man **Schleifendurchlauf** oder **Iterationsschritt**
  - beenden der Wiederholung nennt man auch **terminieren** der Schleife
  - nach jedem Schleifendurchlauf wird Boolescher Ausdruck erneut ausgewertet
  - ist Ausdruck zu Beginn bereits False wird die Schleife nie durchlaufen
  - wird der Ausdruck nie False läuft die Schleife ewig weiter (**Endlosschleife**, **infinite loop**), terminiert nie

Schleifen

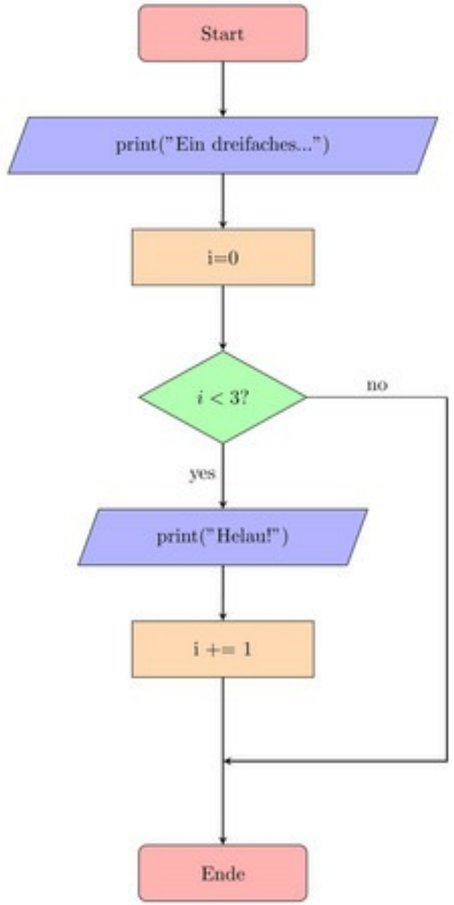
- Schleifen sind oft **Zählschleifen**:
  - Verwenden int-Variable als **Schleifenzähler** (Name meist i)
  - Erhöhen Zähler in jedem Durchlauf
  - Prüfen, dass Wert kleiner als gegebene Schranke ist
- **Beispiel:**

In [12]:
```
#%%tutor -b http://localhost:8003

print("Ein dreifaches...")
i = 0
while i < 3:
    print("Helau!")
    i += 1
```

```
Ein dreifaches...
Helau!
Helau!
Helau!
```

Schleifen



Schleifen

- **Wichtig:** in der Informatik zählt man ab **0**, nicht ab **1**
- Zählschleifen mit n Wiederholungen zählen dann von **0** bis **n-1**
- Unbedingt angewöhnen, großes Verwirrungspotential, oft Quelle von Fehlern bei Programmieranfängern
- Programm würde sich immer noch korrekt verhalten, wenn von 1 bis n gezählt würde
- Ist aber absolut unüblich und **wird zu Verwirrung führen**

Schleifen

- **Wichtig:** in der Informatik zählt man ab **0**, nicht ab **1**
- Zählschleifen mit n Wiederholungen zählen dann von **0** bis **n-1**
- Unbedingt angewöhnen, großes Verwirrungspotential, oft Quelle von Fehlern bei Programmieranfängern
- Programm würde sich immer noch korrekt verhalten, wenn von 1 bis n gezählt würde
- Ist aber absolut unüblich und **wird zu Verwirrung führen**
- ...

Schleifen

- **Wichtig:** in der Informatik zählt man ab **0**, nicht ab **1**
- Zählschleifen mit n Wiederholungen zählen dann von **0** bis **n-1**
- Unbedingt angewöhnen, großes Verwirrungspotential, oft Quelle von Fehlern bei Programmieranfängern
- Programm würde sich immer noch korrekt verhalten, wenn von 1 bis n gezählt würde
- Ist aber absolut unüblich und **wird zu Verwirrung führen**
- ...
- Ok, es gibt Ausnahmen. Aber die ignorieren wir hier noch!

Schleifen

- Schleifen und Conditionals sind ewiger Quell von Programmierfehlern
- Hier werden Entscheidungen getroffen $\Rightarrow$ Potential, etwas falsch zu machen
- Besonders beliebt:
  - **Off-by-one** - error: Bedingungen in if oder while um ± 1 zur richtigen Variante verschoben
  - **Endlosschleife** - Variable wird nicht korrekt erhöht, Bedingung stimmt nicht, ...

Schleifen

- Schleifen sind auch Ursache für lange Programmlaufzeiten!
- Analyse der Anzahl von Anweisungen, die ausgeführt werden müssen, wichtige Aufgabe der Informatik
- Werden wir später noch ansprechen

Schleifen

- Bisherige Zählschleifen wären noch ohne Schleife umsetzbar:
  - kopiere Schleifenrumpf so oft hintereinander, wie Schleife durchlaufen werden soll
  - unpraktisch, aber machbar
- Ist aber kein Ersatz, wenn Anzahl Schleifendurchläufe noch nicht bekannt ist
- **Beispiel:**

In [13]:
```
import random

x = random.randrange(1, 10)
y = random.randrange(1, 10)

guess = input("Was ist " + str(x) + " * " + str(y) + "? ")

while int(guess) != x*y:
    guess = input("Ähh nein! Probier's nochmal... ")

print("Ja, das ist korrekt.")
```

```
Was ist 3 * 3? 1
Ähh nein! Probier's nochmal... 4
Ähh nein! Probier's nochmal... 9
Ja, das ist korrekt.
```

Schleifen

- **Beispiel:**

In [14]:
```
import random

done = False

while not done:
    x = random.randrange(1, 10)
    y = random.randrange(1, 10)

    guess = input("Was ist " + str(x) + " * " + str(y) + "? ")

    while int(guess) != x*y:
        guess = input("Ähh nein! Probier's nochmal... ")

    done = input("Ja, das ist korrekt. Noch ein Versuch? (J/N)") == "N"
```

```
Was ist 8 * 3? 2
Ähh nein! Probier's nochmal... 42
Ähh nein! Probier's nochmal... 24
Ja, das ist korrekt. Noch ein Versuch? (J/N)J
Was ist 4 * 9? 36
Ja, das ist korrekt. Noch ein Versuch? (J/N)blarg
Was ist 7 * 4? 28
Ja, das ist korrekt. Noch ein Versuch? (J/N)N
```

Schleifen

- **Beispiel:**

In [15]:
```
# Berechne Tabelle der ersten n Zweierpotenzen
n = int(input("n? "))

power = 1
i = 0
while i <= n:
    print(str(i) + " " + str(power))
    power = 2*power
    i += 1
```

```
n? 5
0 1
1 2
2 4
3 8
4 16
5 32
```

Schleifen

```
# Berechne Tabelle der ersten n Zweierpotenzen
n = int(input("n? "))

power = 1
i = 0
while i <= n:
    print(str(i) + " " + str(power))
    power = 2*power
    i += 1
```

- Warum ist das Programm eigentlich korrekt?
- Wird ab jetzt immer wichtigere Frage (Programme werden immer komplexer)
- Idealerweise können wir Korrektheit **beweisen**
- Beweise oft nur in einfachen Fällen möglich, aber oft nützlich für Programmteile

Schleifen

- Korrektheitsbeweise für Schleifen oft möglich mit **Schleifeninvarianten**, ähnlich vollständiger Induktion
- Wir nennen eine Aussage Schleifeninvariante, falls Sie
  - vor der ersten Schleifenausführung
  - beim Übergang von einem in den nächsten Schleifendurchlauf
  - nach Beendigung der Schleife

  erfüllt ist
- nach der Schleife gilt dann:
  - Abweisende Bedingung (Schleife wurde beendet)
  - Schleifeninvariante
- kann bei geeigneter Schleifeninvariante Korrektheitsbeweis ermöglichen
- **Achtung:** nicht jede Invariante hilft (2 == 2 ist immer eine Invariante...)

Schleifen

```
power = 1
i = 0
while i <= n:
    print(str(i) + " " + str(power
    power = 2*power
    i += 1
```

- Schleifeninvariante: power == 2^i
  - Vor Schleifendurchlauf hat i den Wert 0, power Wert $1 = 2^0$
  - In der Schleife geht $i \rightarrow i + 1$, power $\rightarrow 2 \times 2^i = 2^{i+1}$
  - Nach Beendigung der Schleife ist i == n; aus Invariante folgt power == 2^n

Schleifen

- Beweisidee von eben führt zu **partiellem Korrektheitsbeweis**
- Für **totalen Korrektheitsbeweis** noch benötigt: Programm **terminiert** immer
- Beweis hier einfach:
  - Ganze Zahl i wächst in jedem Durchlauf
  - Schleife terminiert, wenn Wert von i $> n$ wird

Schleifen

- Weiteres Beispiel für Korrektheitsbeweis: **ggT-Berechnung** (**größter gemeinsamer Teiler**, **greatest common divisor** (**gcd**))

In [16]:
```
x = 234
y = 126

while x != y:
    if x > y:
        x -= y
    else:
        y -= x

print(x)
```

```
18
```

Schleifen

```
while x != y:
    if x > y:
        x -= y
    else:
        y -= x

print(x)
```

- Vor Schleifendurchlauf definieren wir g := ggT(x, y)
- Schleifeninvariante: g == ggt(x, y)
  - Vor Schleifendurchlauf per Definition erfüllt
  - In der Schleife gilt: x != y. o.B.d.A sei x > y (anderer Fall analog)
    - g == ggT(x, y) vor Ausführung des Schleifenrumpfes ⇒ $x = ag, y = bg$
    - Damit gilt $x - y = (a - b)g$ mit $(a - b) > 1$ und $ggT(a - b, b) <= g$ (weil g == ggT(x, y)
    - Also haben auch $x - y$ und y gleichen ggT: g == ggT(x-y, y)
  - Nach Beendigung der Schleife ist x == y und g == ggT(x, y) ⇒ x == y == g == ggT(x, y)

Schleifen

- Für totale Korrektheit fehlt noch Beweis der Terminierung
- **Beweisidee:** x, y starten als natürliche Zahlen (int und > 0)
  - Falls x == y wird Schleife nie durchlaufen (Terminierung trivial)
  - Ansonsten wird x oder y echt kleiner, bleibt aber positiv (andere Zahl bleibt unverändert)
  - Muss irgendwann terminieren, sonst müsste Zahl $<= 0$ werden oder nicht mehr kleiner werden

Schleifen

- Zählschleifen sehr häufig verwendet
- Schema mit while in etwa:

```
i = start
while i < stop:
    <statement>
    <statement>
    ...
    i += 1
```

- Alternative: **for**-Schleife

Schleifen

- for-Schleifen gibt es in mehreren Varianten für verschiedene Zwecke
- **Hier:** einfache Zählschleifen mit Schema

```
for i in range(start, stop):
    <statement>
    <statement>
    ...
```

- start und stop müssen vom Typ int sein
- Schleife wird $stop - start$ - mal durchlaufen
- Schleifenzähler i nimmt in Schleife Werte $start, start + 1, \ldots, stop - 1$ an

Schleifen

- **Beispiel:**

In [17]:
```
print("Ein dreifaches...")
for i in range(0, 3):
    print("Helau!")
```

```
Ein dreifaches...
Helau!
Helau!
Helau!
```

Schleifen

- Zählschleifen beginnen meist bei 0
- Python erlaubt daher Kurzschreibweise (syntaktischer Zucker) ohne Angabe der unteren Grenze

In [18]:
```
print("Ein dreifaches...")
for i in range(3):
    print("Helau!")
```

```
Ein dreifaches...
Helau!
Helau!
Helau!
```

Schleifen

- Weitere Beispiele: **Fakultät**
- $n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$
- **Achtung:** unübliche Zählschleife, nicht von 0 bis $n-1$ sondern von 1 bis n

In [19]:
```
n = 5
fac = 1
for i in range(1, n+1):
    fac *= i
print(fac)
```

```
120
```

Schleifen

- Weitere Beispiele: **Fakultät**
- $n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots \times n$
- Alternativ:

In [20]:
```
n = 5
fac = 1
for i in range(n):
    fac *= i+1
print(fac)
```

```
120
```

Break, continue, pass

- Oft soll Teil des Schleifenrumpfs nur für manche Durchläufe ausgeführt werden
- **Beispiel:**

In [27]:
```
sum = 0
for i in range(3):
    for j in range(3):
        if i != j:
            frac = 1./(i-j)
            print(i, end=" ")
            print(j, end=" ")
            print(frac)
            sum += frac
print(sum)
```

```
0 1 -1.0
0 2 -0.5
1 0 1.0
1 2 -1.0
2 0 0.5
2 1 1.0
0.0
```

Break, continue, pass

- Oft soll Teil des Schleifenrumpfs nur für manche Durchläufe ausgeführt werden
- Python bietet dafür **Kurzschreibweise** (syntaktischen Zucker) durch Schlüsselwort **continue**
- Ausführung von continue
  - beendet aktuellen Schleifendurchlauf (der untersten Schleife die gerade ausgeführt wird)
  - überprüft Schleifenkopf neu
  - fährt normal weiter fort

In [26]:
```
sum = 0
for i in range(3):
    for j in range(3):
        if i == j:
            continue
        frac = 1./(i-j)
        print(i, end=" ")
        print(j, end=" ")
        print(frac)
        sum += frac
print(sum)
```

```
0 1 -1.0
0 2 -0.5
1 0 1.0
1 2 -1.0
2 0 0.5
2 1 1.0
0.0
```

Break, continue, pass

- Noch radikaler: Schlüsselwort **break**
- Beendet (unterste derzeit ausgeführte) Schleife sofort vollständig
- Keine weitere Überprüfung des Schleifenkopfs
- Ermöglicht oft einfachere Formulierung schwieriger Bedingungen

Break, continue, pass

- **Beispiel:** (verwendet random.random() - zieht zufälliges float gleichverteilt aus $[0.0, 1.0)$

In [23]:
```
# Ziehe zufälligen Punkt im Kreis mit Radius $1$
import random

while True: # wäre ohne break Endlosschleife!
    x = random.random()
    y = random.random()

    if x*x + y*y <= 1: # falls ja liegt Punkt im Kreis, sonst nochmal versuchen
        break

print("(" + str(x) + ", " + str(y) + ")")
```

```
(0.9139481471267971, 0.3714684203872354)
```

Break, continue, pass

- Abhängige Blöcke von if, else, for, while **müssen** immer mindestens ein Statement enthalten
- Sonst: Compile-Fehler
- Manchmal soll aber nichts getan werden (z.B. während Entwicklung, Rest des Programms soll schnell getestet werden, ...)
- Dazu dient sogenannte **no op** **pass**
- Tut bei Ausführung gar nichts...

Break, continue, pass

- **Beispiel:**

In [24]:
```
if True: # dieses if hat keinerlei Auswirkung
    pass
```

Break, continue, pass

- pass wirkt zunächst völlig sinnlos...
- ... ist aber immer wieder mal nützlich

Gambler's ruin

- Komplexeres Beispiel: Simulation des **Gambler's ruin** (Sedgewick et al., Introduction to programming in Python)
- Spieler wettet immer wieder $1\times$ auf fairen Münzwürf
- Führt unweigerlich irgendwann in den Ruin, wenn Spieler nicht aufhört
- Aber was passiert, wenn sich Spieler ein Ziel setzt und aufhört, sobald es erreicht ist?
- Können wir simulieren!

Gambler's ruin

- **Idee:**
  - führe möglichst große Anzahl Zufallsexperimente durch
  - lasse Spieler immer auf Kopf setzen (spielt für WSK bei fairer Münze keine Rolle)
  - höre auf, wenn Spieler ruiniert ist oder Ziel erreicht hat
- Verhältnis von Gewinn zu Gesamtzahl Experimente ergibt dann Gewinnwahrscheinlichkeit
- Bestimmen außerdem, wie lange im Durchschnitt gespielt wurde

Gambler's ruin

In [25]:
```
import random

stake  = 100 # Startkapital
goal   = 250 # Ziel
trials = 100 # Anzahl Zufallsexperimente

bets = 0 # wie lange gespielt
wins = 0 # wie oft gewonnen
for t in range(trials): # Schleife über Experimente
    cash = stake # Fülle Geld wieder auf
    while (cash > 0) and (cash < goal): # Schleife über Wetten
        bets += 1
        if random.randrange(0, 2) == 0: # Kopf geworfen
            cash += 1
        else: # Zahl geworfen
            cash -= 1
    if cash == goal: # gewonnen oder verloren?
        wins += 1

print(str(100 * wins // trials) + '% Gewinnwahrscheinlichkeit')
print('Durchschnittliche Spieldauer: ' + str(bets // trials))
```

```
36% Gewinnwahrscheinlichkeit
Durchschnittliche Spieldauer: 12853
```

Gambler's ruin

- Konvergiert für große Wiederholungsanzahl gegen klassisches Resultat aus der Statistik:

  Gewinnwahrscheinlichkeit = Startkapital / Ziel

  für dieses Spiel
- Wozu simulieren, wenn Ergebnis aus Mathematik bekannt?
- Simulation kann leicht angepasst werden um leicht andere Fragen zu beantworten
- Theoretische Herleitung oft erheblich komplexer
- **Beispiel:** wie ändern sich Wahrscheinlichkeiten, wenn Spieler nach einer vorgegebenen Anzahl Wiederholungen aufhört?