Grundlagen der Programmierung

Vom Algorithmus zum Programm:

Imperative Programmierung ♦ Python

Algorithmisches Denken: Vom Problem zur Lösung

1. **Identifizieren des Problems**
2. **Formulieren des Problems**
3. **Entwurf des Algorithmus**
4. **Implementierung des Algorithmus**
5. **Anwendung des Algorithmus**

→ Problemlösung

Programmiersprache

- Bisher: Formulierung der Algorithmen
  - verbal (*informal*) oder
  - in Pseudocode (*semi-formal*)
- Jetzt: Formulierung der Algorithmen in einer Programmiersprache (*formalen Sprache*)
  - exakte Definition der **Syntax** („*Rechtschreibung und Grammatik*“)
  - und von deren Bedeutung (**Semantik**)
- Dadurch
  - auf Rechenanlage *automatisch* ausführbar
  - keine Mehrdeutigkeiten mehr enthalten

Mehrdeutigkeit in der deutschen Sprache

Von Eva K. / Eva K. - Eva K. / Eva K., FAL, https://commons.wikimedia.org/w/index.php?curid=748457

Programmiersprache Python

- frei verfügbar für die meisten Betriebssysteme
- http://www.python.org
- Version für diesen Kurs: Python 3 (z.B. Python 3.9.0)
- **Literatur**
  - Th. Theis: Einstieg in Python.
    Ideal für Programmiereinsteiger. 7. Auflage,
    Rheinwerk Computing, 2022
  - https://wiki.python.org/moin/GermanPythonBooks

Algorithmische Konzepte: Variablen

…
- haben einen **Namen** (*x*, *z*, *L*, *var*, *input,* ...)
- dienen zum Merken von **Werten**
  - Eingabedaten
  - Berechnungsergebnisse, Zwischenergebnisse, Zähler, ...
- Werte können durch Anweisungen verändert werden

- **Python: Variablennamen**
  - (Buchstabe + _) (Buchstabe + Ziffer + _)*
  - Groß- und Kleinschreibung wird unterschieden
  - Wörter mit Sonderbedeutungen sind **reserviert** (dürfen nicht als Variablennamen verwendet werden)

Werte

- Mögliche Werte gehören zu einem **Datentyp**.
- Datentypen sind bestimmt durch
  1. eine Menge von gültigen Werten (*Wertebereich*)
  2. *Operationen* auf diesen Werten
     (z.T. mehrsortig)

Datentypen

- **Python: Zahlentypen**

Syntax (*Literale*)	Beispiele	Semantik
Dezimalziffernfolge (hinter optionalem '+' oder '-' ohne führende 0)	1234, -17, 0	Ganze Dezimalzahl: **Integer**, Typ **int**
'0o' gefolgt von Integer '0x' gefolgt von Integer	0o224, 0o10 0x224, 0x10	int als Oktalzahl int als Hexadezimalzahl
Dezimalziffernfolge mit '. ' oder .-Zahl gefolgt von 'e' oder 'E' gefolgt von Integer	1.25, 4E100 3.14e-10 -2.1e+20	Dezimalbruch bzw. Basis · 10^Exponent (*bei* e/E): **Gleitkommazahl**, Typ **float**

Dezimalbruch

Integer

Datentypen

Operationen auf Zahlen

- Vorzeichenwechsel (durch + oder –)
- Addition +, Subtraktion -, Multiplikation *
- Division
  - exakte Division /
  - ganzzahlige Division //
    (auch bei Gleitkommazahlen)
  - Rest bei ganzzahliger Division %

```
5 / 2 = 2.5
5 // 2 = 2
5.0 // 2 = 2.0
5 % 2 = 1
```

weitere Operationen s. Rechnerübungen

Datentypen

- **Python: Datentyp String (Zeichenketten)**, Typ **str**
  - Folge von **Zeichen**:
    Buchstaben, Ziffern, Sonderzeichen, Leerzeichen
  - in einfache, doppelte oder dreimal doppelte Hochkommata gesetzt
  - Beispiele:

```
"Hallo Python"
'auch das ist ein "String"! ^^ '
""" "So definierte Zeichenketten dürfen auch
    Zeilenumbrüche als Zeichen enthalten. """
```

Datentypen

Operationen auf Strings

`s1 = ""`	leerer String
`s2 = "***"`	
`s3 = "-$$-"`	
`s4 = s2 + s3`	Konkatenation; ergibt ***-$$-
`s5 = s3*3`	Vervielfachung; ergibt -$$--$$--$$-
`len(s4)`	Länge; ergibt 7

weitere Operationen s. Rechnerübungen

Algorithmische Konzepte: Listen

- **indizierte Listen**
  - sind spezielle Variablen
  - zum Merken einer Vielzahl von Werten
  - Zugriff auf Listenelemente über Indizes: *Liste*[*Index*]
  - können verschachtelt werden (Listen von Listen ...)

- **Python: Datentyp Liste**

`L1 = []`	leere Liste
`L2 = [1,4,8]`	drei Elemente, Indizes **0**, **1**, **2** (**null-indizierte Liste**)
`L3 = [1,[5,9]]`	verschachtelte Liste mit `L3[0]=1`, `L3[1][0]=5`, `L3[1][1]=9`

Datentypen

- **Python: Operationen auf Listen**

`len(L2)`	Länge (Anzahl der Elemente)
`L2+L3`	Verkettung
`L2[i:j]`	Slice (Teilliste von `i` bis vor `j`)
`L2[i:]`	Teilliste ab `i`
`L2[:j]`	Teilliste bis vor `j`

weitere Operationen s. Rechnerübungen

- Strings sind spezielle Listen (von Zeichen, unveränderlich)
  → **Slice-Operationen auch für Strings**

Datentypen

- **Python: Datentyp Wahrheitswert**, Typ **bool**
  - **True** (wahr)
  - **False** (falsch)

  } *Literale*

- **Operationen:**
  - `and` (logisches UND / Konjunktion)
  - `or` (logisches ODER / Disjunktion)
  - `not` (logisches NICHT / Negation)

Datentypen

- Weitere Datentypen in Python:
  - Tupel
  - Menge
  - Dictionary
  - Datei
  - vom Programmierer definierte

- *z.T. in den Rechnerübungen*

Ausdrücke

- setzen sich zusammen aus Variablen, Werten und Operationen auf Werten und Variablen,
  z.B.
  $n + 4$, $m - (3k + 1)$, $|L|/2$, *- var*
  → *Operationen wirken auf Teilausdrücke (Terme)*

  - *Term Operation Term* (bei zweistelligen Operationen)
  - *Operation Term* (bei einstelligen Operationen)

- haben einen Wert

Ausdrücke (1)

- **Arithmetische Ausdrücke**

  ((3+n)*(m-1))//2

  Syntax:
  - „gültige“ Kombination aus Operatoren und Operanden
  - Kombination ist „gültig“, wenn der Definitionsbereich (*insbesondere die Stelligkeit*) der Operation beachtet ist
  - Operanden können selbst Ausdrücke sein
  - Klammerungen sind erlaubt

  Semantik:
  - **Wert** nach *Auswertung* des Ausdrucks
  - Operationen haben *Prioritäten* (z.B. Punktrechnung vor Strichrechnung)

Assoziativität

- $(a \circ b) \circ c = a \circ (b \circ c)$
- Gilt nicht für jede Operation:

  $(5 - 4) - 1 = 0 \neq 2 = 5 - (4 - 1)$

- **Python:**
  - Nicht assoziative Operationen werden in der Regel *linksassoziativ* ausgewertet: `5-4-1` ergibt 0
  - Sonst: Klammern verwenden: `5-(4-1)` ergibt 2

Algorithmische Konzepte: Bedingungen

- *Beispiele:* $z > x$, $L[i] > x$, $i \leq |L|$, L nicht leer, ...
- enthalten Variablen
- werden in Abhängigkeit vom Wert der Variablen wahr oder falsch
- mehrere Bedingungen können zu einer Bedingung zusammengesetzt werden
  - aussagenlogische Operationen:
    UND, ODER, ENTWEDER-ODER, ...
  - *z.B.* $x > 0$ UND $x < y$
  - *z.B.* $i > |L|/2$ UND $i \leq |L|$

Ausdrücke (2)

- **Boolesche Ausdrücke (Bedingungen)**

  Syntax:

  - Vergleich zweier arithmetischer Ausdrücke mit

`>`	größer als	`>=`	größer als oder gleich
`<`	kleiner als	`<=`	kleiner als oder gleich
`==`	gleich	`!=`	ungleich

  - `and` (UND), `or` (ODER), `not` (NICHT)-Verknüpfung
  - Klammerungen sind erlaubt

  Semantik:

  - **Wert** nach *Auswertung* des Ausdrucks: `True` oder `False`

Prioritäten

Operator	Bedeutung
`+ -` (einstellig)	positives/negatives Vorzeichen
`* / % //`	Punktrechnung
`+ -` (zweistellig)	Strichrechnung
`< <= > >= == !=`	Vergleich arithmetischer Ausdrücke
`not`	logische Verneinung (Negation)
`and`	logisches UND (Konjunktion)
`or`	logisches ODER (Disjunktion)

in abnehmender Priorität geordnet

Algorithmische Konzepte: Anweisungen

- ~~Anweisungen können im Prinzip beliebig formuliert werden, solange klar ist, was zu tun ist~~

- *insbesondere für Listen L:*

  füge … *L* hinzu — **`L.append(...)`** — Einfügen am Ende

  und

  entferne … aus *L* : **`L.remove(...)`** — Löschen des ersten Vorkommens des Elements

  `del L[0]`, **`del [1:3]`** — Löschen der genannten El.

- **Ausgabeanweisung**

  gib … aus: *zum Beispiel mit* **`print(...)`**

Algorithmische Konzepte: Anweisungen

0. Ausdruck

- Berechnung des Wertes eines Ausdrucks:
  „Ein Ausdruck gibt seinen Wert zurück.“
- relativ nutzlos, wenn nirgends gespeichert/verwendet

1. Zuweisung

- Zuweisung eines Wertes an eine Variable
- *Variable* ⟵ *Ausdruck*

Rechtsausdruck

- **Python:** ***variable*** **=** ***ausdruck***, *Beispiele:*

```
a = 5                    b = b + 3
L = [0,4,12]             s = "Hallo"
```

Algorithmische Konzepte: Anweisungen

2. Sequenz

- Folge von Anweisungen; der Reihe nach abzuarbeiten
- **Python: Anweisungsblock (Block)**
  eine oder mehrere Anweisungen,
  getrennt durch Semikolon oder Zeilenumbruch
  mit gleicher Einrücktiefe
- *Beispiel:*

```
a = 5
b = a + 3
print(b)
```

in Pseudocode:

$a \longleftarrow 5$

$b \longleftarrow a + 3$

gib b aus

Algorithmische Konzepte: Anweisungen

3. Fallunterscheidung

- **falls** *Bedingung*
  Anweisungsblock

  (***bedingte Anweisung***)

- **falls** *Bedingung*
  Anweisungsblock
  sonst
  Anweisungsblock

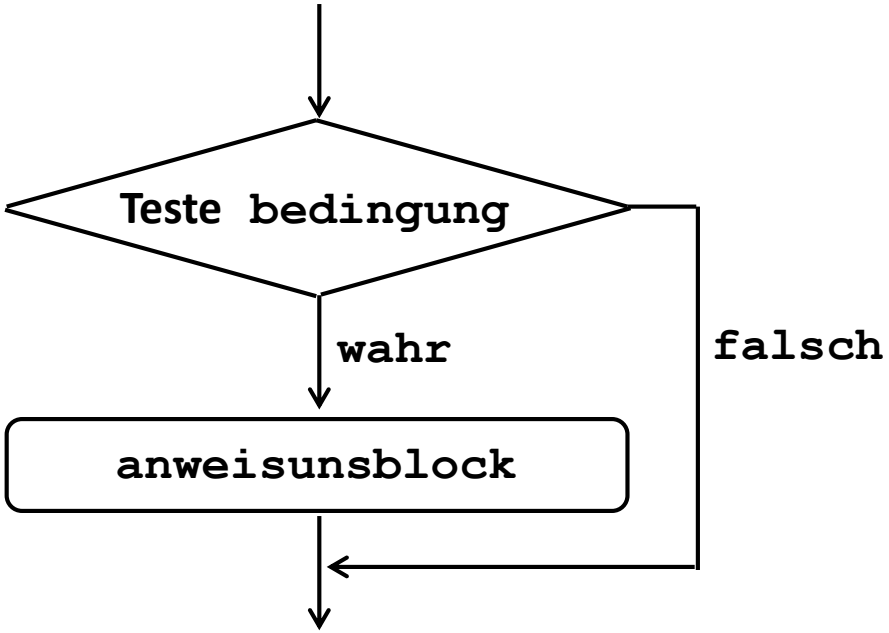
  (***alternative Anweisung***)

- ohne **sonst:** weiter mit nächster Anweisung
- *Beispiel:*
  falls $z > x$
  $x \longleftarrow z$
  gib x aus
- eingerückte Anweisungen gehören zu **falls** bzw. **sonst**

Bedingte Anweisung

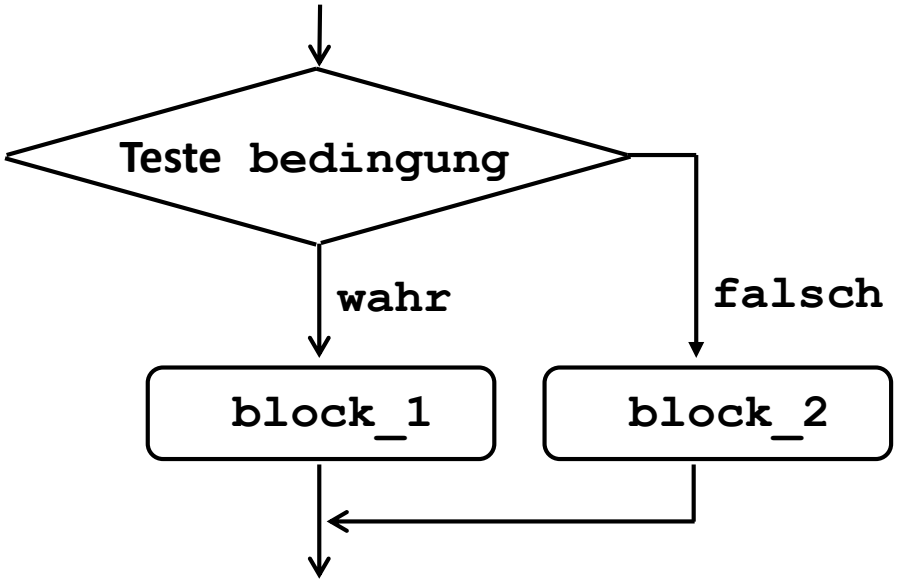
```
if bedingung:
    anweisungsblock
```

Eine oder mehrere Anweisungen mit gleicher Einrücktiefe



Fallunterscheidung

```
if bedingung:
      block_1
else:
      block_2
```



Mehrfache Verzweigung mit `elif`

```
if bedingung1:
      block_1
elif bedingung2:
      block_2
else:
      block_3
```

mehrere **`elif`**-Anweisungen sind möglich

Algorithmische Konzepte: Anweisungen

4. Wiederholung

- *Beispiele:*
  für alle *z* in *L*
  Anweisungsblock

  für *i* ⟵ 1 bis |*L*|
  Anweisungsblock

- allgemein:
  Wiederholungssteuerung
  Anweisungsblock

- Die Anweisungen im *Anweisungsblock* werden wiederholt ausgeführt, solange es die *Wiederholungssteuerung* verlangt; dann weiter mit nächster Anweisung

Arten der Wiederholungsteuerung (1)

- **für alle** *Variable* **in** *Kollektion*
- **Python: for-in-Schleife**

```
for variable in liste:
    block
```

- *Beispiel:*

```
L = [0,4,8]
for x in L:
    print(x)
```

0
4
8

`variable` nimmt nacheinander die Werte an, die in der Liste enthalten sind. Für jeden Wert in der Liste werden die Anweisungen im eingerückten `block` ausgeführt.

Arten der Wiederholungsteuerung (1)

- **für alle** *Variable* **in** *Kollektion*
- **Python: for-in-Schleife**

```
for variable in liste:
    block
```

- Listenelemente können auch hinter `in` direkt aufgezählt werden:

```
for x in 0,4,8:
    print(x)
```

Arten der Wiederholungsteuerung (2)

- **für** *Variable* ⟵ *Ausdruck* **bis** *Ausdruck*
  - dabei Variablenwert schrittweise um 1 erhöhen
  - **Python: Zählerschleife**

```
for i in range(1,4):
    print("Zahl: ", i)
```

```
Zahl: 1
Zahl: 2
Zahl: 3
```

Verkettung mehrerer Ausgaben auf derselben Zeile

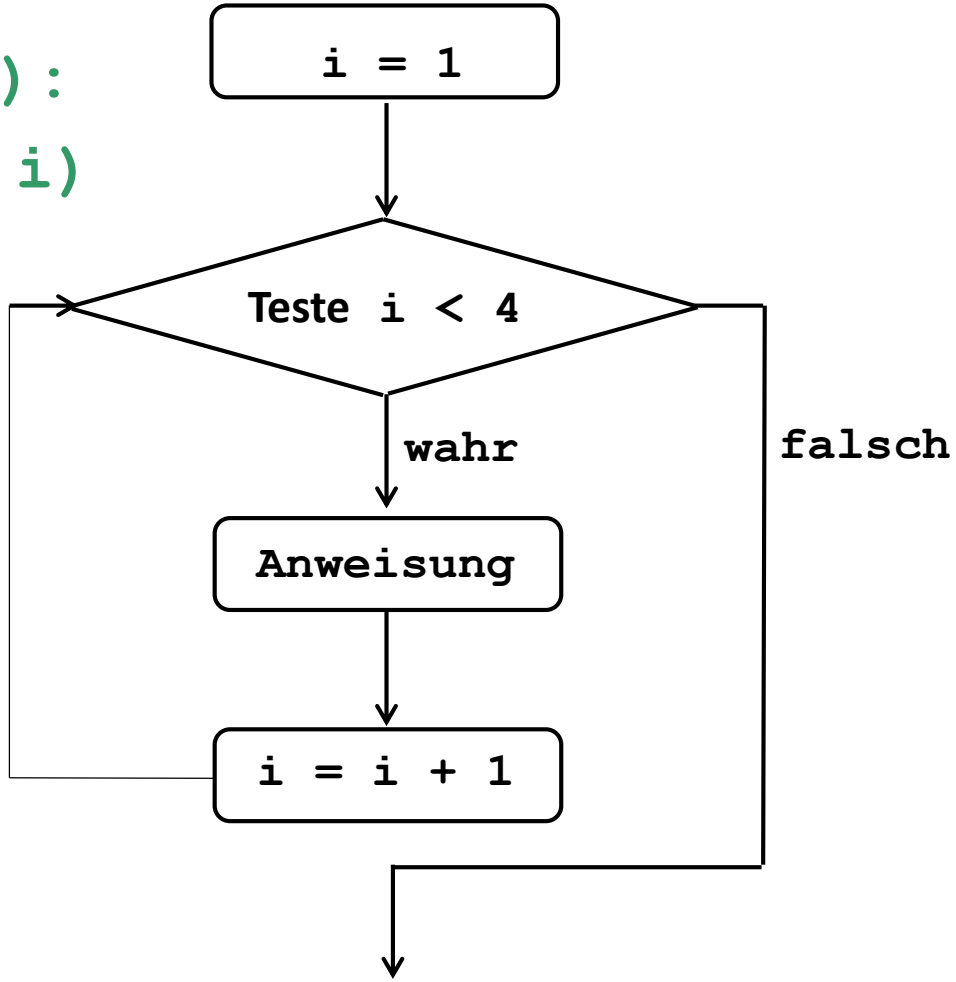
  - `range(beginn, ende)`

    `beginn` ist der erste Wert, für den die Anweisung ausgeführt wird

    `end` ist der erste Wert, für den die Anweisung nicht mehr ausgeführt wird

Arten der Wiederholungsteuerung (2)

```
for i in range(1,4):
    print("Zahl: ", i)
```



Arten der Wiederholungsteuerung (2)

- allgemeiner:
  `range(beginn,ende,schrittweite)`

```
for i in range(1,10,2):
    print(i," Quadrat ist ",i*i)
```

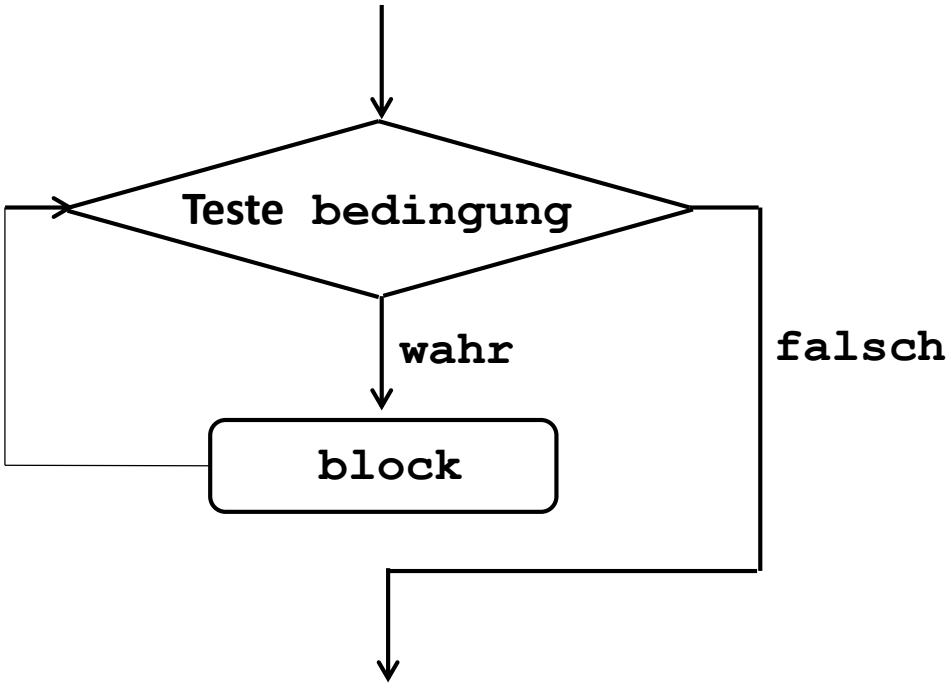
```
1 Quadrat ist 1
3 Quadrat ist 9
5 Quadrat ist 25
7 Quadrat ist 49
9 Quadrat ist 81
```

Arten der Wiederholungsteuerung (3)

solange *Bedingung*

$i \longleftarrow 1$
solange $i \leq |L|$

```
while bedingung:
    block
```



Arten der Wiederholungsteuerung (3)

- **solange** *Bedingung*

  $i \longleftarrow 1$
  solange $i \leq |L|$

- **Python: bedingte Schleife**

```
while bedingung:
        block
```

- *Beispiel:*

```
i = 4
while i>0:
    print(i)
    i = i-1
```

4
3
2
1

Zuerst wird `bedingung` geprüft. Solange dies `True` ergibt, werden die Anweisungen im **block** ausgeführt und dann die Bedingung vor jeder Wiederholung erneut geprüft.

Python-Programm

- Ein Algorithmus ist eine **Sequenz** (Anweisungsfolge), die Eingabedaten in Ausgabedaten überführt.

- Ein Python-Programm ist ein Anweisungsblock, also eine Folge von Python-Anweisungen.
  - Diese werden nacheinander von einer Maschine ausgeführt.
  - Das Programm endet, wenn es keine nächste Anweisung gibt.
  - Programmieren durch Angabe einer Sequenz von Anweisungen heißt **imperative Programmierung.**

Python-Interpreter

- Der Python-Interpreter ist das Programm, das die Python-Anweisungen (zeilenweise)
  - liest und
  - für deren Ausführung auf der Maschine sorgt

  (*interpretiert*).

- UNIX-Kommando:
  - **`python`** (startet interaktiven Modus)
  - **`python`** *datei.py* (Ausführung des Programms in *datei.py*)

Von Pseudocode zu Python

Name: **größtes Listenelement**
Eingabe: Liste *L* ganzer Zahlen
Ausgabe: größte Zahl in der Liste

x ⟵ größtes Element der Liste *L*
gib *x* aus

Benötigen offenbar eine Variante, die stark genug verfeinert ist.

```
x ⟵ - ∞
für alle z in L
    falls z > x
        x ⟵ z
gib x aus
```

```
x ⟵ - ∞
für i ⟵ 1 bis |L|
    falls L[i] > x
        x ⟵ L[i]
gib x aus
```

Von Pseudocode zu Python

$x \longleftarrow -\infty$
für $i \longleftarrow 1$ bis $|L|$
    falls $L[i] > x$
        $x \longleftarrow L[i]$
gib x aus

```
x = -float('inf')
for i in range(0,len(L)):
        if L[i] > x:
                x = L[i]
print(x)
```

```
float('inf') :  ∞
-float('inf'): -∞
```

Von Pseudocode zu Python (2)

$x \longleftarrow -\infty$
für $i \longleftarrow 1$ bis $|L|$
    falls $L[i] > x$
        $x \longleftarrow L[i]$
gib x aus

```
x = -float('inf')
for i in range(0,len(L)):
        if L[i] > x:
                x = L[i]
print(x)
```

$x \longleftarrow -\infty$
für alle z in L
    falls $z > x$
        $x \longleftarrow z$
gib x aus

```
x = -float('inf')
for z in L:
        if z > x:
                x = z
print(x)
```

Kommentare

- Kommentare sind Texte im Programm,
  die bei der Ausführung ignoriert werden.
- Sie dienen der Erläuterung der Anweisungen im Programm.

```
x = -float('inf') # minus unendlich
for i in range(0,len(L)):
      if L[i] > x:
            x = L[i]
print(x) # Ausgabe des größten Listenelements
```

- von `#` bis zum Zeilenende (*einzeilige Kommentare*)
- von `"""` bis `"""` (*mehrzeilige Kommentare*)

Beispiel 2 – erster Versuch

Name: **Liste aller Teiler**
Eingabe: eine positive ganze Zahl *x*
Ausgabe: Liste mit allen Teilern von *x*

$i \longleftarrow 1$
für $k \longleftarrow 1$ bis x
    falls x MOD $k = 0$
        $L[i] \longleftarrow k$
        $i \longleftarrow i + 1$
gib L aus

```
i = 0
for k in range(1,x+1):
        if x % k == 0:
                L[i] = k
                i = i + 1
# Ausgabe von L
```

Liste existiert nicht!
Benötigte Länge der Liste ist unbekannt!

Beispiel 2 – zweiter Versuch

Name: **Liste aller Teiler**
Eingabe: eine positive ganze Zahl *x*
Ausgabe: Liste mit allen Teilern von *x*

```
L = []
for k in range(1,x+1):
      if x % k == 0:
            L.append(k)
# Ausgabe von L
```