Grundlagen der Programmierung

Vom Algorithmus zum Programm:

Prozedurale Programmierung mit Python
Wiederverwendung, Funktionen, Aufrufstack

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

Imperative Programmierung mit Python

- Ein Python-Programm ist eine Sequenz, 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.**
- Vorgehen bisher:
  - Programm als in sich geschlossene Folge von Anweisungen
  - „größere Aufgaben“ müssen verfeinert werden
  - **Verfeinerungen „einsetzen“**

Beispiel: g.g.T.

Name: **größter gemeinsamer Teiler (ggT)**
Eingabe: zwei positive ganze Zahlen x und y
Ausgabe: ggT von x und y

falls $x > y$
  vertausche x und y
$L_1 \longleftarrow$ Liste aller Teiler von x (aufsteigend sortiert)
$L_2 \longleftarrow$ Liste aller Teiler von y (aufsteigend sortiert)
$i \longleftarrow |L_1|$
solange $i \geq 1$
  falls $L_1[i]$ ist in L_2
    gib $L_1[i]$ aus
    STOP
  $i \longleftarrow i - 1$

$tmp \longleftarrow x$
$x \longleftarrow y$
$y \longleftarrow tmp$

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

Beispiel: Code für Verfeinerungen

$tmp \leftarrow x$
$x \leftarrow y$
$y \leftarrow tmp$

```
tmp = x
x = y
y = tmp
```

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

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

Imperatives Python-Programm (g.g.T.)

falls $x > y$
    vertausche x und y
$L_1 \longleftarrow$ Liste aller Teiler von x
$L_2 \longleftarrow$ Liste aller Teiler von y
$i \longleftarrow |L_1|$
solange $i \geq 1$
    falls $L_1[i]$ ist in L_2
        gib $L_1[i]$ aus
        STOP
    $i \longleftarrow i - 1$

```
if x > y:
        tmp = x
        x = y
        y = tmp
L1 = []
for k in range(1,x+1):
        if x % k == 0:
                L1.append(k)
L2 = []
for k in range(1,y+1):
        if y % k == 0:
                L2.append(k)
```

Imperatives Python-Programm (g.g.T.)

falls $x > y$
    vertausche x und y
$L_1 \longleftarrow$ Liste aller Teiler von x
$L_2 \longleftarrow$ Liste aller Teiler von y
$i \longleftarrow |L_1|$
solange $i \geq 1$
    falls $L_1[i]$ ist in L_2
        gib $L_1[i]$ aus
        STOP
    $i \longleftarrow i - 1$

```
i = len(L1) - 1
while i >= 0:
    if L1[i] in L2:
        print(L1[i])
        break
    i = i - 1
```

`break`-Anweisung bewirkt unmittelbares Beenden der Schleife.

Imperatives Python-Programm (g.g.T.)

```
if x > y:
        tmp = x
        x = y
        y = tmp
L1 = []
for k in range(1,x+1):
        if x % k == 0:
                L1.append(k)
L2 = []
for k in range(1,y+1):
        if y % k == 0:
                L2.append(k)
```

```
i = len(L1) - 1
while i >= 0:
        if L1[i] in L2:
                print(L1[i])
                break
        i = i - 1
```

Beobachtung:

Das Teilprogramm zur Bestimmung aller Teiler tritt für verschiedene Variablennamen mehrfach aus.

Wiederverwendung

- Teilprogramme (*wie Liste aller Teiler*) definieren **Prozeduren**.
- Prozeduren lösen oft typische Aufgabenstellungen bei der Programmierung.
- Prozeduren eignen sich, in demselben oder anderen Programmen wiederverwendet zu werden.
- **Prozedurale Programmierung** erlaubt es Programmen, definierte Prozeduren aufzurufen.
- Prozeduren können in einem Programm oder in einer Programm-Bibliothek verwaltet werden.

Prozeduren mit Parametern

- Für die Wiederverwendung müssen die Variablennamen für die Eingabewerte angepasst werden.
- Eingabewerte werden als Parameter übergeben:

```
prozedur(parameter)
prozedur(param_1, param_2, …, param_k)
```

Bespielprozeduren

```
def trennung(s,n):    # s String, n Integer
      s = s*n
      print(s)
```

Schlüsselwort **def** für die Definition von Prozeduren

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

```
return L
```

Rückgabewert

Prozedurales Python-Programm (g.g.T.)

```
if x > y:
        tmp = x
        x = y
        y = tmp
L1 = teilerlist(x)
L2 = teilerlist(y)
i = len(L1)- 1
while i >= 0:
        if L1[i] in L2:
                print(L1[i])
                break
        i = i - 1
```

Prozeduraufrufe

Rückgabewerte werden ggf. an Variablen zugewiesen.

Name `teilerlist` muss dem Programm durch die **`def`**-Anweisung bekannt gegeben sein.

Funktionen

- Funktionen sind Prozeduren mit Rückgabewert.

Eine k-stellige **Funktion** $f: M_1 \times M_2 \times \ldots \times M_k \longrightarrow N$
ist eine Relation, die jedem Tupel aus $M_1 \times M_2 \times \ldots \times M_k$
eindeutig ein Element aus N zuordnet: $f(x_1, x_2, \ldots, x_k) = y$.

- Es entsteht also zu jedem Tupel aus k Eingabewerten ein eindeutig zugehöriger Ausgabewert.

`teilerlist(x)` ↦ `L` ist einstellige Funktion

Eingebaute Prozeduren und Funktionen

- in Python vordefiniert
- Beispiele

`print(s)`	mit einem String als Parameter
`input()`	ohne Parameter, Rückgabe **str** *(s. Rechnerübungen)*
`float(s)`	mit einem String als Parameter, Rückgabe **float**
`int(s)`	mit einem String als Parameter, Rückgabe **int**
`len(L)`	mit einem String oder einer Liste als Parameter, Rückgabe **int**

Prozedurales Python-Programm (g.g.T.)

Hauptprogramm definiert auch eine Funktion → *möglich*:

```
def ggT(x,y):
    if x > y:
        tmp = x
        x = y
        y = tmp
    L1 = teilerlist(x)
    L2 = teilerlist(y)
    i = len(L1)- 1
    while i >= 0:
        if L1[i] in L2:
            print(L1[i])   return L1[i]
            break
        i = i - 1
```

`return`-*Anweisung beendet die Prozedur und übergibt den Ausgabewert*

Berechenbare Funktion

- Eine Funktion heißt (algorithmisch) **berechenbar**, wenn sie von einem Algorithmus (einem Python-Programm) berechnet wird.
  - unter Beachtung des Definitionsbereichs
    → *Berechnung muss nur für zulässige Eingabewerte (aus dem Definitionsbereich) korrekt sein*
  - Algorithmus muss aber entscheiden können, ob Eingabewerte zulässig sind

Formale und aktuelle Parameter

- Definition der Funktion

```
def teilerlist(x)
        L = []
        for k in range(1,x+1):
                if x % k == 0:
                        L.append(k)
        return L
```

- Aufruf der Funktion

```
L1 = teilerlist(x)
L2 = teilerlist(y)
```

Formale Parameter:
sind Variablen, die nur innerhalb der Funktionsdefinition benutzt werden

Aktuelle Parameter:
sind Ausdrücke, deren Werte den entsprechenden formalen Parametern zugewiesen werden

Funktionsaufrufe bei der Interpretation

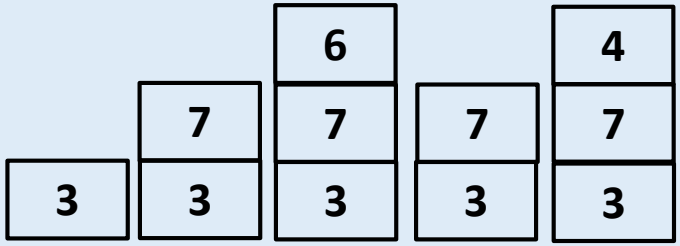
- *Wie kann der Interpreter die Variablen der Funktion von den sonstigen Programmvariablen unterscheiden?*

➢ Verwaltung der Variablen in getrennten Speicherbereichen: **Stackframe**

- ➢ Zu jedem Zeitpunkt werden die Anweisungen von immer nur einer Funktion ausgeführt (*der Funktion, die zuletzt aufgerufen wurde*).
- ➢ Der zugehörige Stackframe verwaltet die Werte der Variablen, die zu dieser Funktion gehören (s. Funktionsdefinition).
- ➢ Die aufrufende Funktion pausiert, bis die Anweisungen der aufgerufenen Funktion abgearbeitet sind.
- ➢ Der Stackframe der aufrufenden Funktion muss gespeichert werden.

Stack

```
st = []
st.append(3)
st.append(7)
st.append(6)
st.pop()
st.append(4)
```



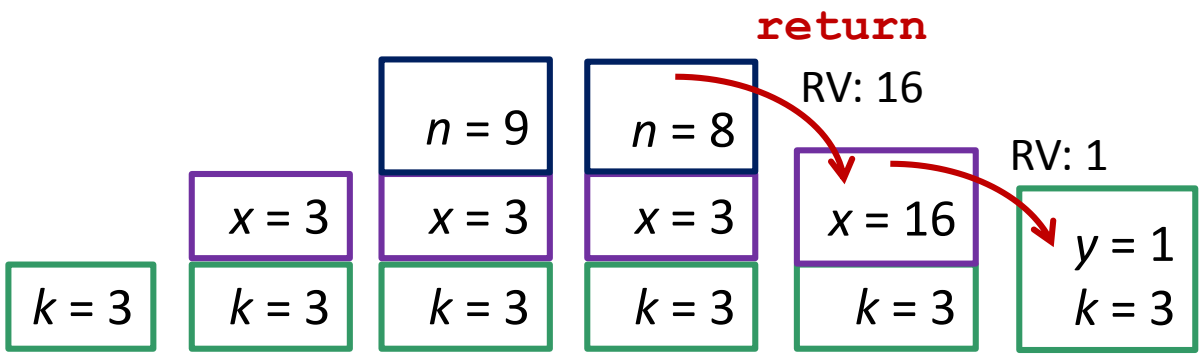
Stack

- Ein **Stack** ist eine Datenstruktur, die eine Kollektion von Daten nach dem **Last-In-First-Out (LIFO)**-Prinzip verwaltet.
- *Liste*, bei der neue Elemente immer nur an derselben Seite eingefügt und gelöscht werden können:
  - Einfügen:
    ```
    st.append(x)          # push
    st[len(st):] = [x]
    ```
  - Löschen:
    ```
    st.pop()
    del st[len(st)-1]
    # und Rückgabe von st[len(st)-1]
    ```
- Eine **Queue (Warteschlange)** ist eine Kollektion, die die Daten nach dem **First-In-First-Out (FIFO)**-Prinzip verwaltet.

Aufrufstack (vereinfacht)

- Stack, der die Stackframes der aufgerufenen Funktionen speichert
- Stackframe des Hauptprogramms zuerst einfügen

```
def f(n):
    n = n - 1
    return 2*n
def g(x):
    x = f(x*x)
    return x%3
k = 3
y = g(k)
```



- Aufruf einer Funktion: **push** neuer Stackframe
- Ende der Funktionsanweisungen: **pop** Stackframe und ggf. Übergabe des Rückgabewerts an Aufrufer

Vermeidung von Namenskollisionen

- Verwendung derselben Variablen mit unterschiedlichen Werten in verschiedenen Funktionen möglich

```
def f(x):
    x = x // 7
    return x
x = 23
y = x - f(x)          # y = 23 - 3
                      # y = 20
```

- Erklärung mit dem Aufrufstack?

Referenzen

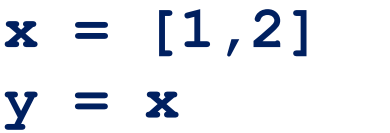
- **Python:** Identifier ist **Referenz** (Verweis) auf die Speicherstelle für den Wert.
- Zugriff auf den Wert einer Variablen `x` :
  1. Nachschlagen der Speicherstelle, die `x` referenziert
  2. Auslesen/Eintragen des Wertes an dieser Speicherstelle
- Wertzuweisung übergibt die Referenz (*keine Kopie der Werte*)
- *Beispiel Liste:*

```
x = [1,2]
y = x                # Übergabe der Referenz
y.append(3)
print(x)             # Ausgabe: [1,2,3]
```

Änderungen der Variablenwerte

1. Zuweisung an neue Variable
   erzeugt zweite Referenz auf die Speicherstelle

```
x = [1,2]
y = x
```



2. Wert über zweite Referenz ändern

```
y.append(3)
```


- Änderung ist auch über die erste Referenz sichtbar

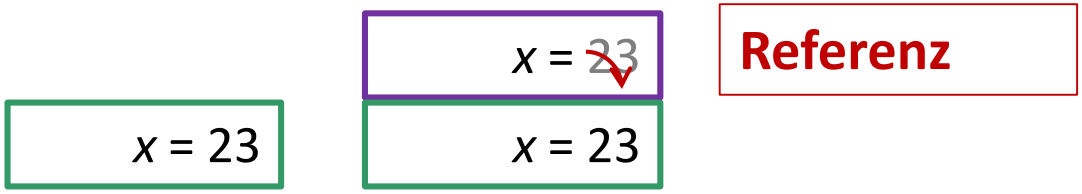
```
print(x)  # [1,2,3]  # [1,2]
```

- Wird aber ein neuer Wert mit dem =-Operator zugewiesen, dann wird dieser an einem anderen Speicherplatz abgelegt
  → Änderung über erste Referenz *nicht* sichtbar

Vermeidung von Namenskollisionen

- Verwendung derselben Variablen mit unterschiedlichen Werten in verschiedenen Funktionen möglich

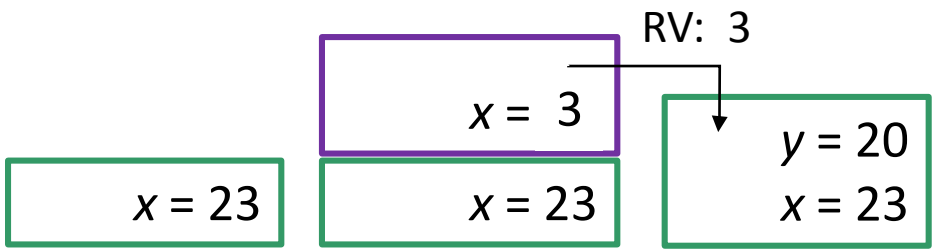
```
def f(x):
    x = x // 7
    return x
x = 23
y = x - f(x)
```



Vermeidung von Namenskollisionen

- Verwendung derselben Variablen mit unterschiedlichen Werten in verschiedenen Funktionen möglich

```
def f(x):
    x = x // 7
    return x
x = 23
y = x - f(x)
```



- Wird einer als Parameter übergebenen Variablen ein neuer Wert (mit `=`) zugewiesen, wird im Stackframe eine neue Variable angelegt.
- Die Variablen der aufgerufenen Funktionen *verdecken* dann gleichnamige Variablen von zuvor aufgerufenen Funktionen.

Verdeckung

- *Beispiel Zahl:*

```
def f(x):
    x = 3
    x = x + 1
x = 1
f(x)
print(x)      # 1
```

- *Beispiel Liste:*

```
def g(L):
    L = [3,4]
    L.append(5)
L = [1,2]
g(L)
print(L)      # [1,2]
```

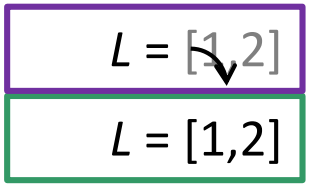
- **ABER:**

```
def h(L):
    L[0] = 3
L = [1,2]
h(L)
print(L)      # [3,2]
```

Seiteneffekt

Seiteneffekte

- **Wirkungen** an Werten von Variablen ohne direkte Veränderung des Wertes



- Bei Funktionsaufruf:
  - Übergabe einer **Referenz** (Verweis) auf den Parameter.
  - Jede Änderung einer **Eigenschaft** der übergebenen Variablen wirkt direkt an der *Speicherstelle* dieser Variablen.
    → **Seiteneffekte** (*z.B.* *L*[0] = 3)
  - Erst bei Zuweisung eines neuen Werts an eine Parameter-Variable wird eine neue Variable im Stackframe angelegt.
- *Dies ist eine Besonderheit von Python.*
  Die meisten anderen Programmiersprachen zeigen ein etwas anderes Verhalten.
- Unbemerkte Seiteneffekte sollten vermieden werden!

Erwünschte Seiteneffekte: Globale Variablen in Python

- Erklärung, dass Funktionen/Prozeduren
  - eine Variable nicht als lokale Variable definieren,
  - sondern eine außerhalb der Funktion definierte verwenden sollen:
- Schlüsselwort **global**

```
def f():
    global x
    x = 7
```

```
x = 3
f()
print(x)   # 7
```

Globale Variablen *versus* Parameter

- Nutzung globaler Variablen ermöglicht Seiteneffekte

  Der Aufrufer einer Prozedur kann unerwünschte Änderungen an seinen Variablen erfahren.

- Guter Programmierstil:
  - Globale Variablen nur einsetzen, wenn dies dem Zweck der Variablen entspricht.
  - Sonst: Parameter und Rückgabewerte
    → *Seiteneffekte vermeiden*

Module

- Prozeduren können in einem Programm oder in einer Programm-Bibliothek verwaltet werden.
- **Modul**: Textdatei, mit Python-Definitionen und -Anweisungen
  - Dateiname endet auf `.py`
  - Name vor `.py` ist der Modulname
  - *Beispiel:* `ggt.py` definiert das Modul `ggt`
- Python-Programme (z.B. andere Module) nutzen ein Modul durch `import` *Modulname*, z.B. `import ggt`
  - gibt nur den Namen bekannt
  - Zugriff auf Variablen und Funktionen: *Modulname*`.`*Bezeichner* z.B. `ggt.x` oder `ggt.ggt(x,y)`

Module nutzen

- **`import`**-Anweisungen üblicherweise am Anfang von Moduldefinitionen
- **`from`**-Anweisung zum Einbinden von Variablen oder Funktionen aus einem Modul, *z.B.*

```
from ggt import ggt
k = ggt(12,16)
```

- Importieren aller Namen eines Moduls:

```
from ggt import *
```