TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Thiemo Leonhardt
Professur für Didaktik der Informatik

Von der Idee zum Programm

2. Vorlesung Programmierung für das Lehramt

2019-04-23

Organisatorisches

Hilfestellungen:

- In der Übung
- Bei Ihren Kommilitonen
- Slub
- Internetlinks im OPAL

- Per Mail:
  - programmierung-la@mailbox.tu-dresden.de

Lernziele

Grundlegenden Teilbereiche der imperativen Programmierung:

- **Wiederholung**: Was Sie bis jetzt können sollten!
  - Algorithmen verstehen und transformieren
    - Text in Code, Formel zu Code, 2D-Grafik zu Code
  - Algorithmische Grundkonstrukte in Python
    - Konkatenation (Programmblöcke)
    - Alternative (if-Bedingung)
    - Iteration (while, for..in)

- ***Effiziente*** *Gestaltung* des ***Problemlöseprozesses*** durch ***modulares*** Arbeiten mit **Funktionen** und **Prozeduren**

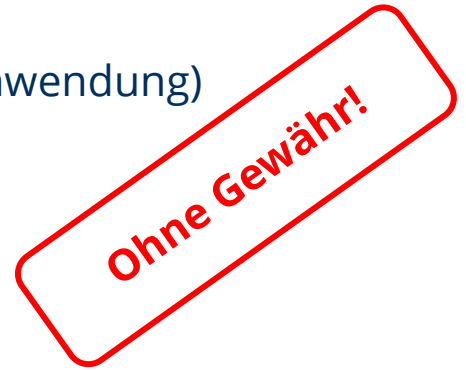
Klausurinformationen

Klausur:

- Voraussichtlich am 18.07.2019 (**noch nicht final**)
- 90 Minuten

Inhalte:

- Konzepte (MP)
- Text/Fomel/Grafik in Struktogramm in Python Code
- Spielen Sie Debugger / Variablen Gültigkeit
- Referenzen auf Objekte
- Problemlösestrategien
- MVC – Modularisierung
- Effizienzbetrachtungen (Anwendung)



Klausurübungen

- Am **30.4.2019** und **02.05.2019** wird es eine **Mini-Klausur 45 Minuten** über den bisherigen Inhalt der Übung und der Vorlesung geben.
- Eine **Probeklausur 90 Minuten** gibt es am **02.07.2019**, **03.07.2019** und **04.07.2019**.

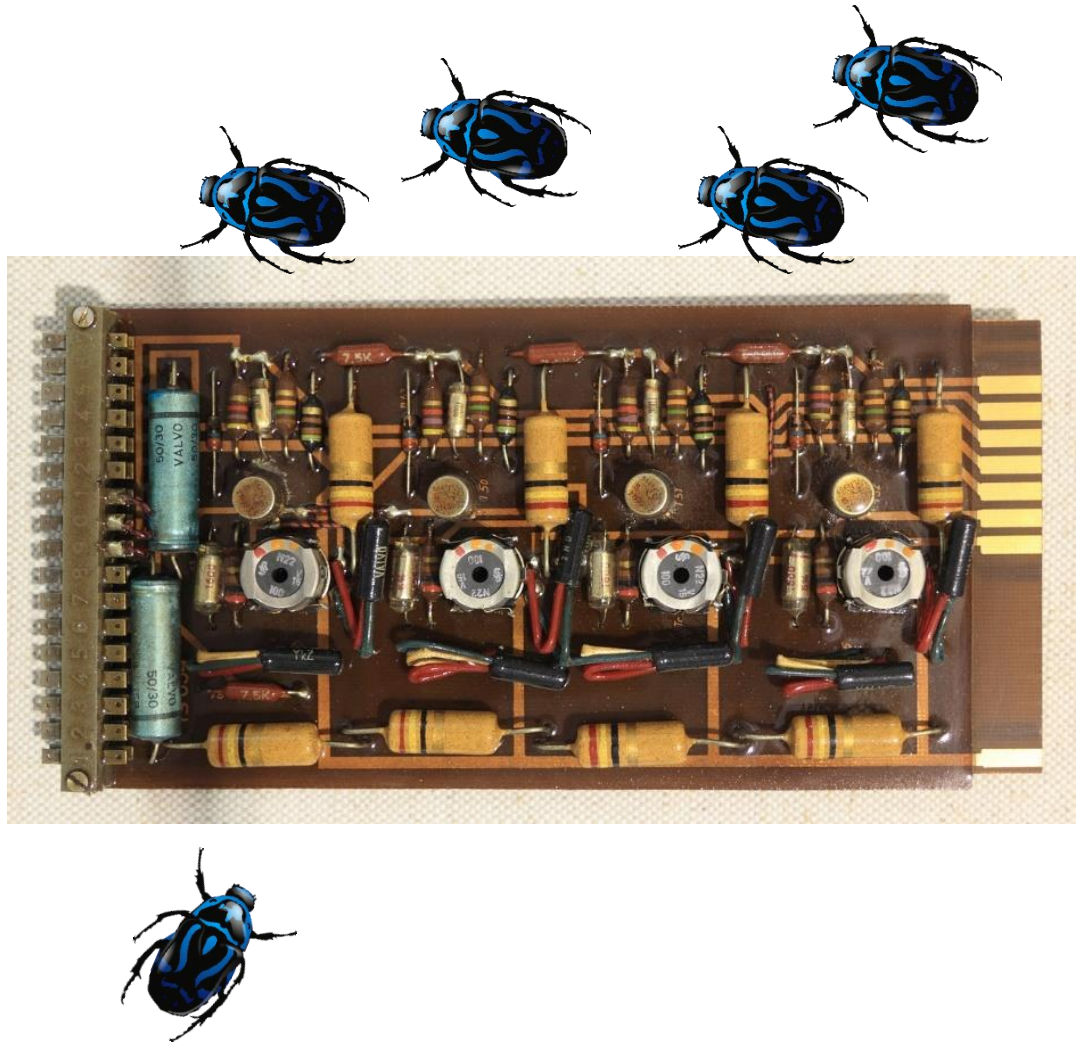
- Die Probe-Klausuren können (müssen aber nicht) abgegeben werden und werden dann korrigiert.
- Die Probe-Klausuren werden hochgeladen und ausgewählte Aufgaben in der Übung besprochen.

Debugger und Variablen Zuweisung

Debuggen = Käfer *entfernen*

- Wanzen oder Käfer (engl. Bug) sorgten für Kurzschlüsse
- Daher mussten diese gefunden und entfernt und *am Besten direkt schon vorher abgefangen werden*
  - *DEBUG(-GEN)*

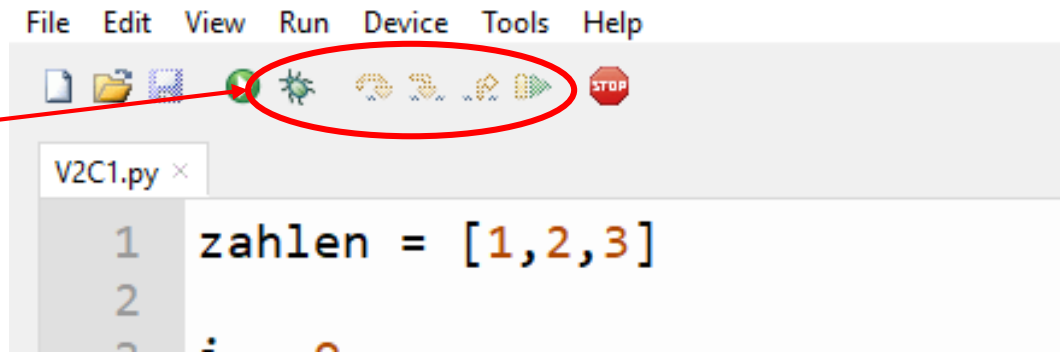
- Daher kommt der Begriff des Findens von Fehlern beim Ablauf eines Programms auf einem Computer
  - find the bug and fix the bug
- Beheben eines Fehlers wird daher mit Bugfix bezeichnet



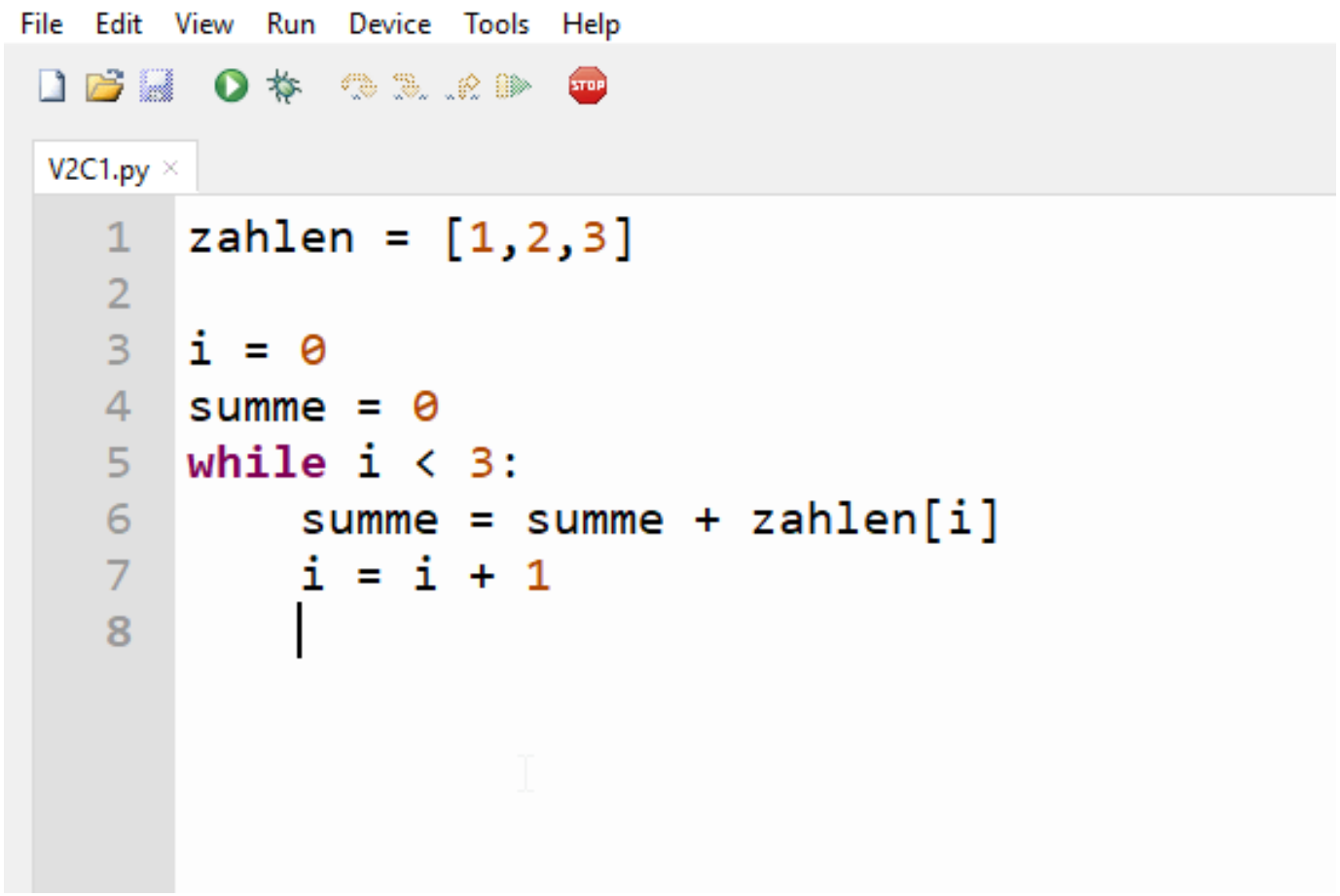
Wie funktioniert Debugging?

Für das Debugging gibt es verschiedene Methoden:

1. Benutzen Sie ihren Kopf
   - typische Fehler finden und Code danach absuchen
   - versuchen Sie selbst den Ablauf des Programmes auszuführen
2. Ausgaben im Programm
   - Variablenwerte an bestimmten Stellen ausgeben
   - print(Variable)
3. Debugger



Codebeispiel – Thonny Debugger



Casting: Variablen Typen ändern

Castbar ohne Informations-Verlust.

Implizit durch den Compiler

bool **int** **float** **complex**

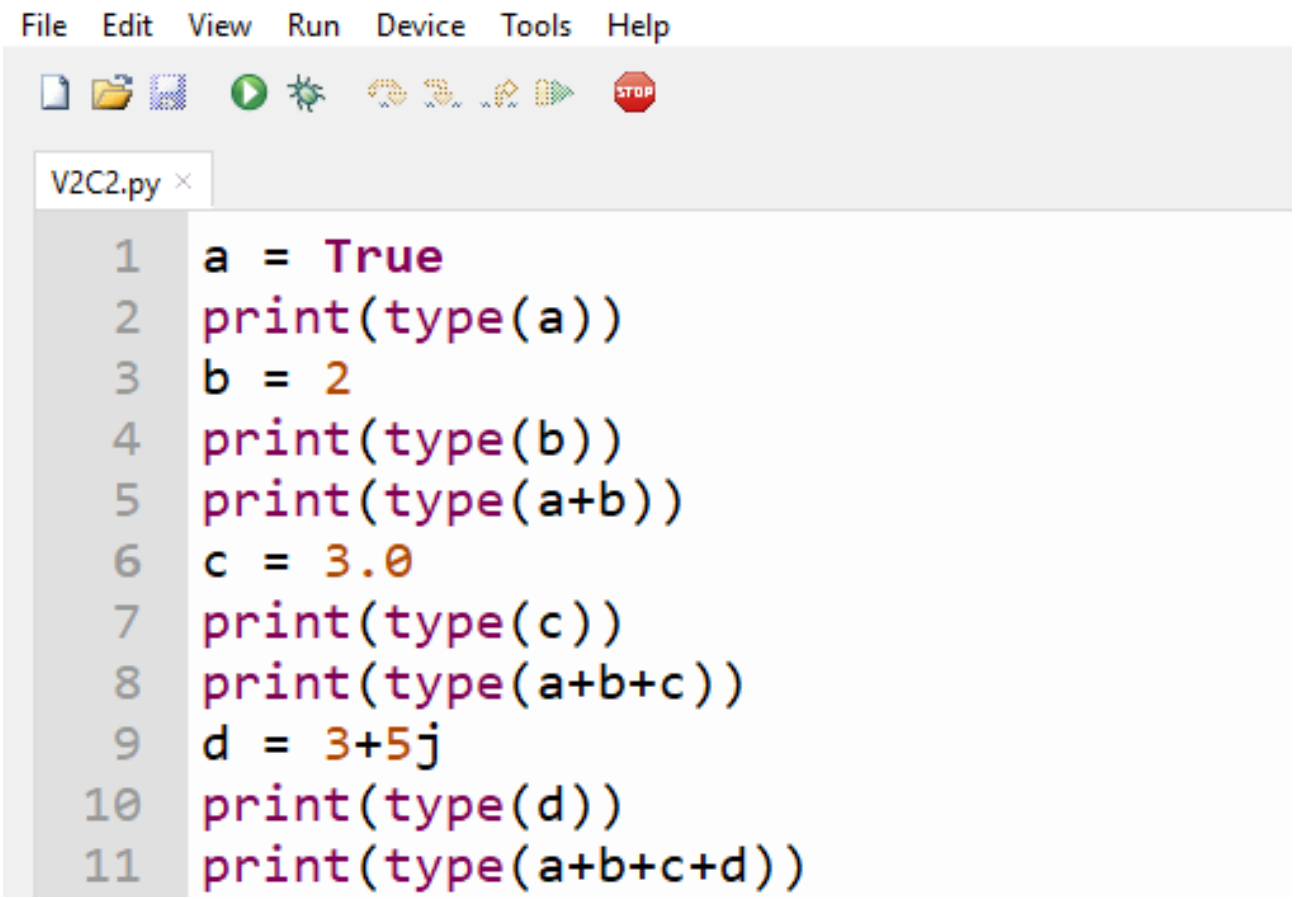
Explizit durch Sie

String

Tuple List Dictionary

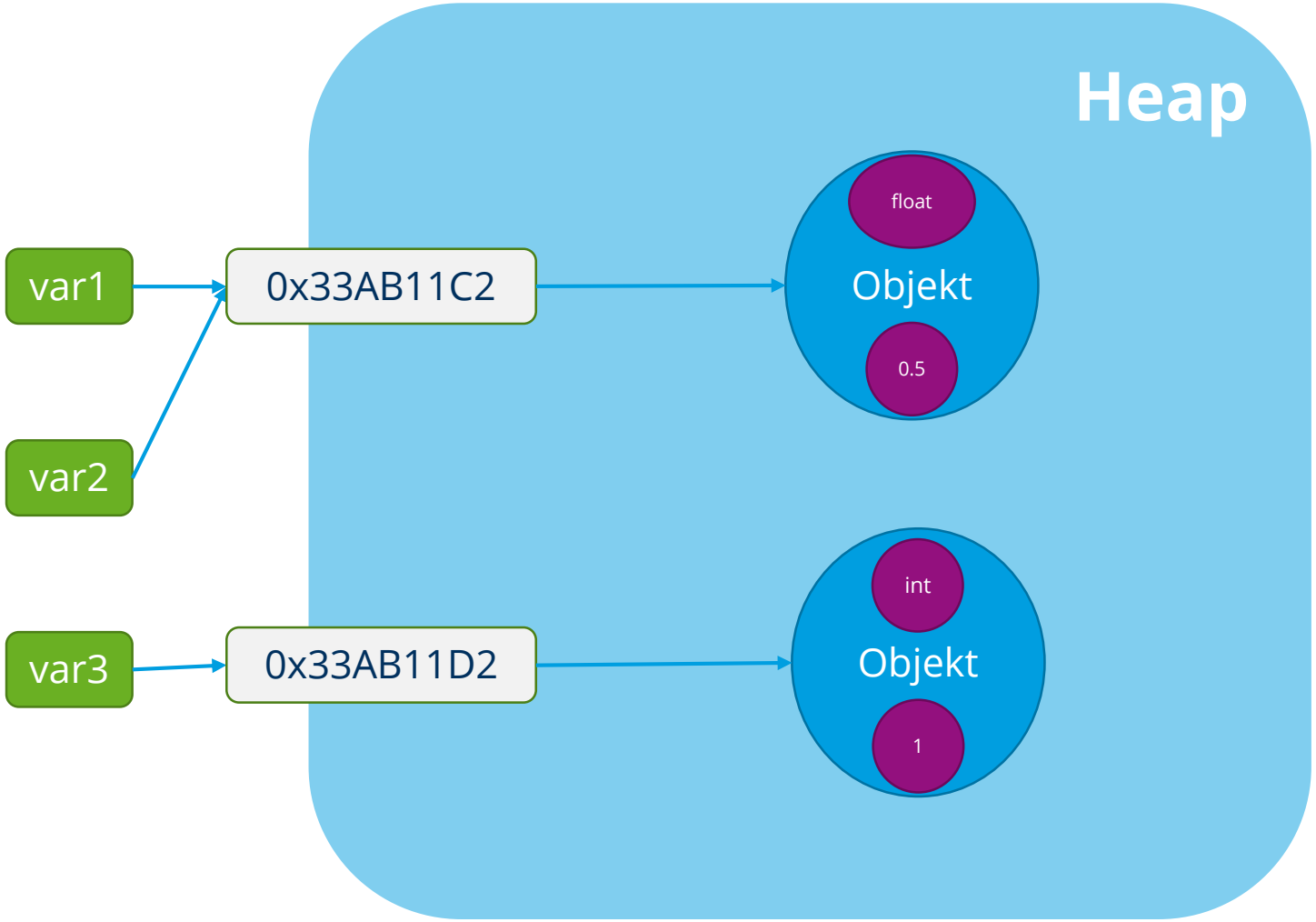
Bewusste Entscheidung.

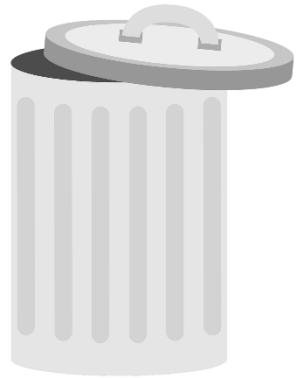
Codebeispiel – Casting von Variablen



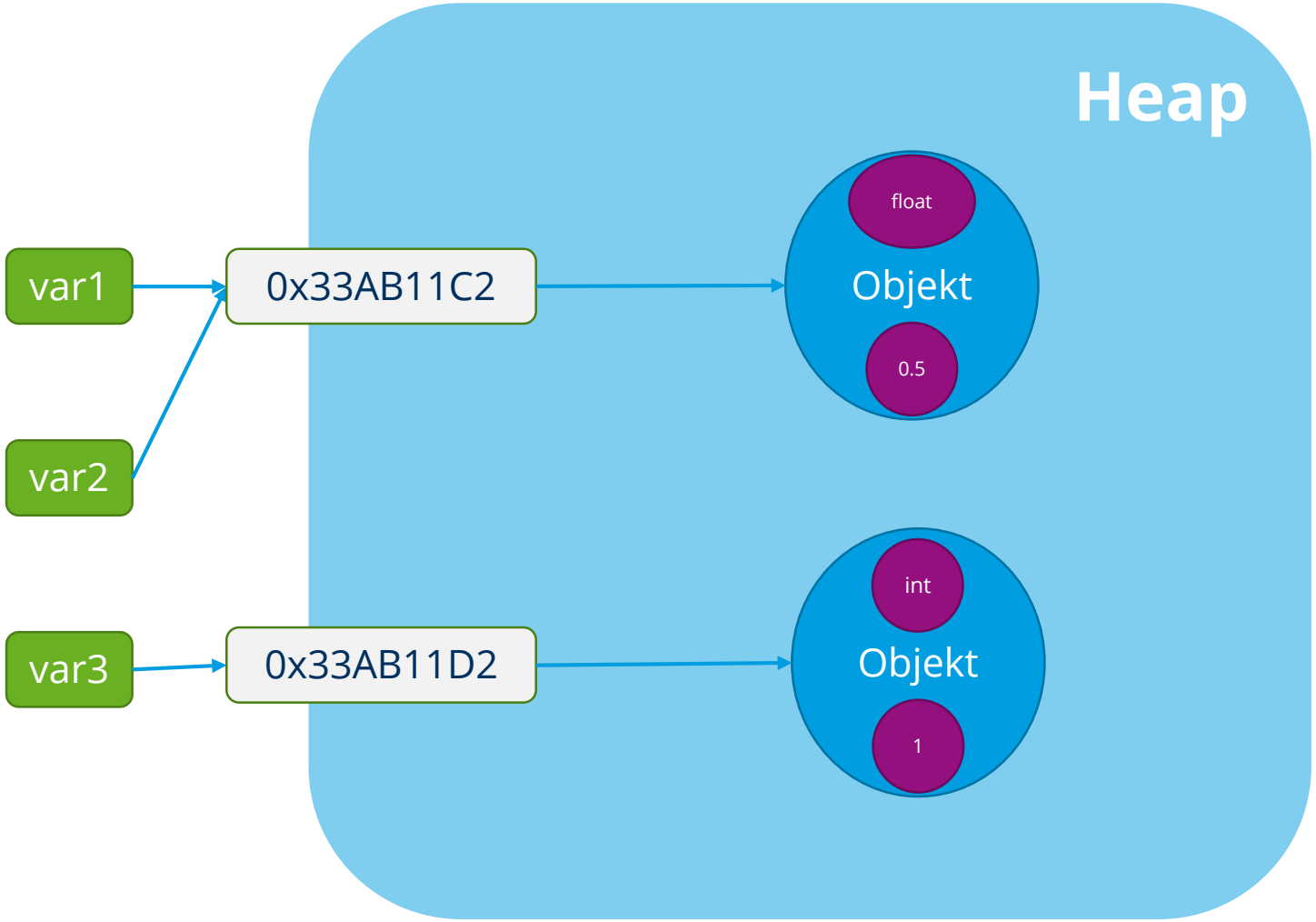
Einschub Garbage Collector

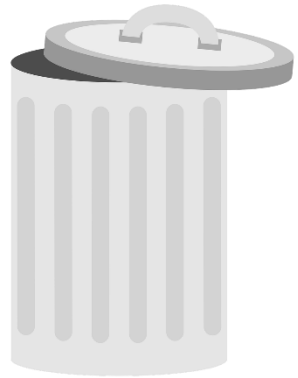
Heap (=Haufen) Speicher



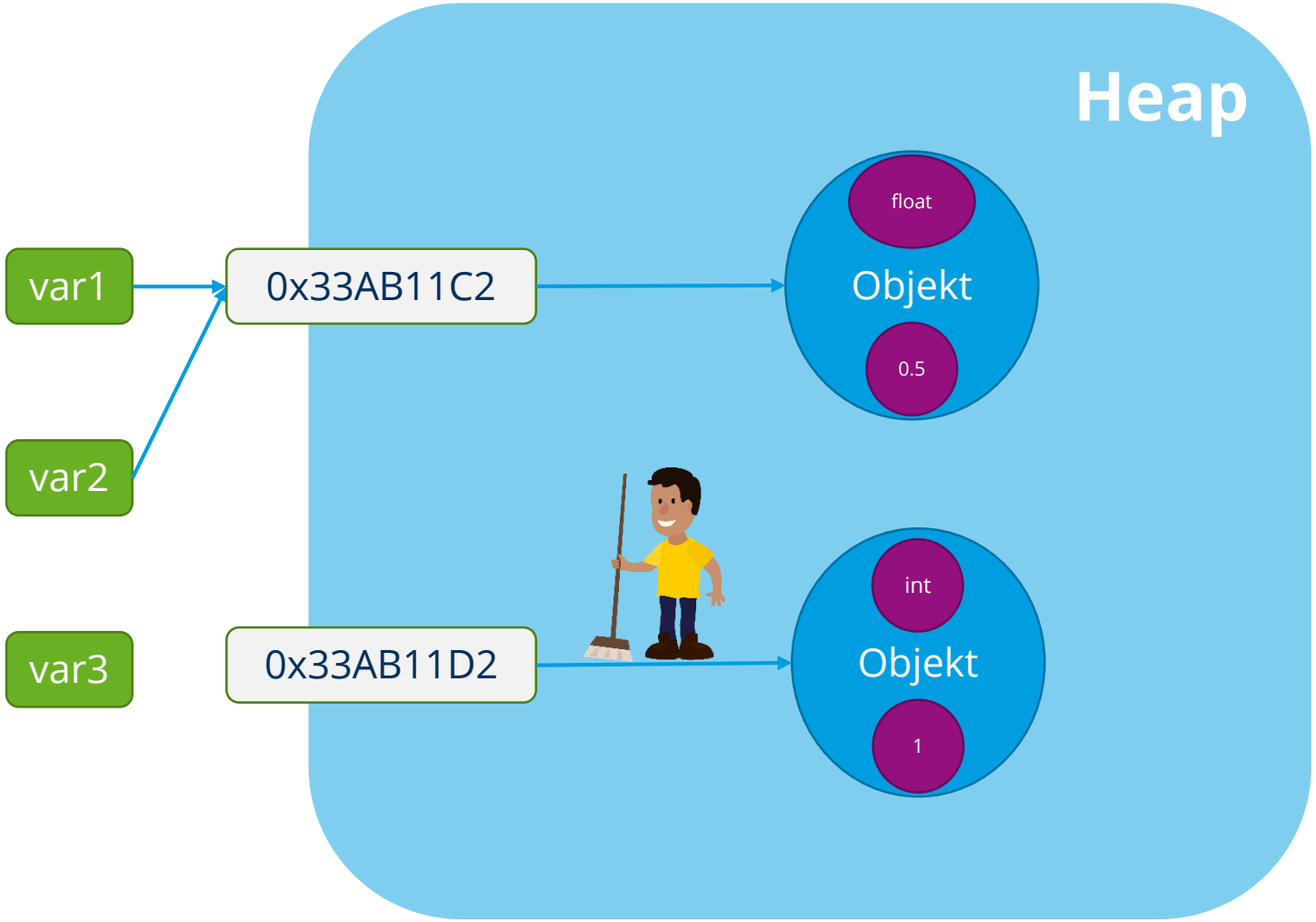


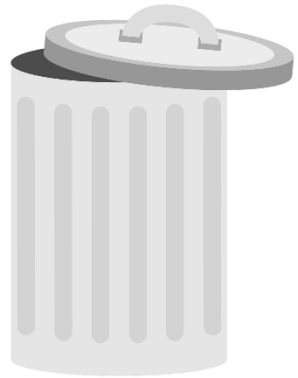
Heap (=Haufen) Speicher





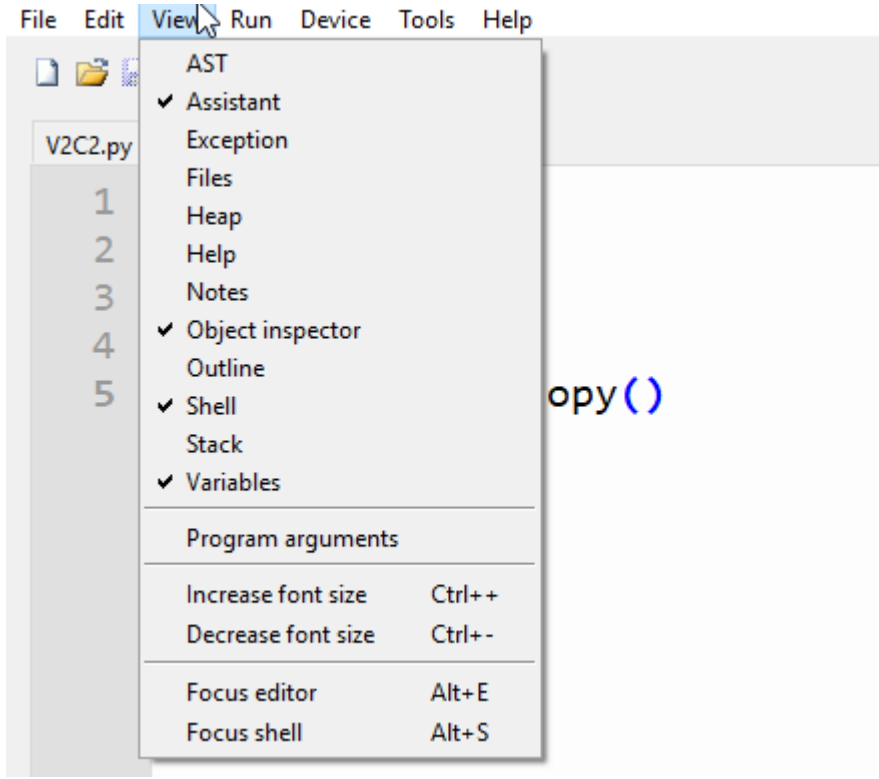
Heap (=Haufen) Speicher





Codebeispiel – Heap

- die Heapsicht kann unter View->Heap aktiviert und deaktiviert werden



Einfache Schleifen und erweiterte Schleifen

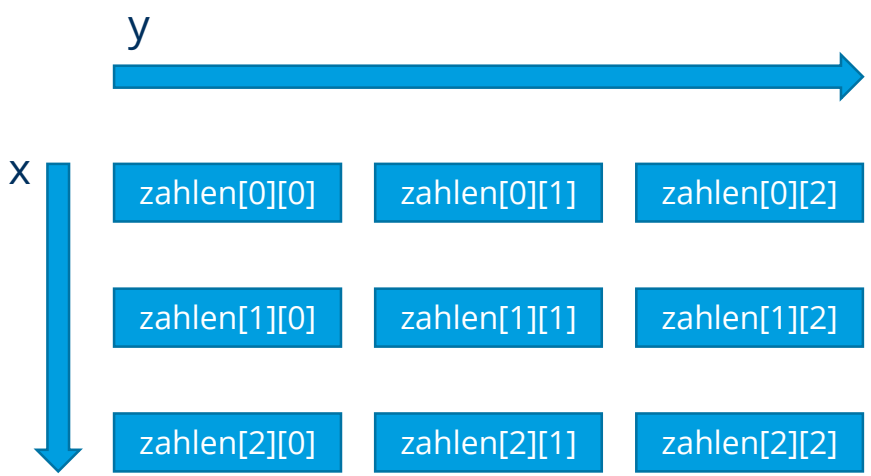
Einfache Schleifen – Lineare Datenstrukturen

File Edit View Run Device Tools Help

V2C1.py

```
1  zahlen = [1,2,3]
2
3  i = 0
4  summe = 0
5  while i < 3:
6      summe = summe + zahlen[i]
7      i = i + 1
8
```

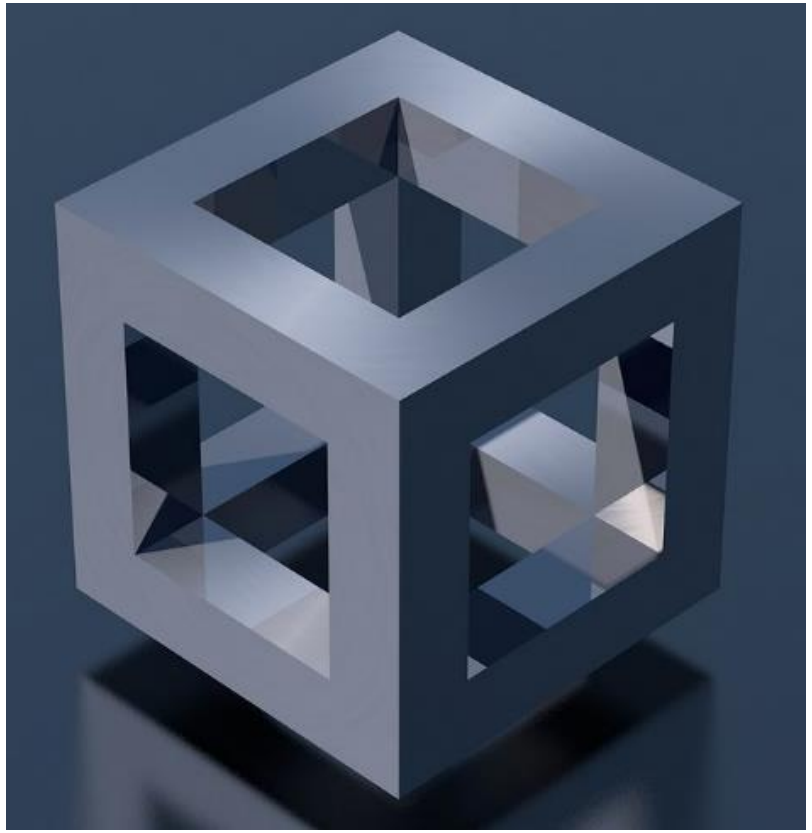
Erweiterte Schleifen – 2-Dimensionale Datenstrukturen



V2C2.py | V2C3.py | V2C1.py | V2C4.py

```
1  zahlen = [[1,2,3],[4,5,6],[7,8,9]]
2
3  x = 0
4  y = 0
5  summe = 0
6  while x < 3:
7      while y < 3:
8          summe = summe + zahlen[x][y]
9          y = y + 1
10     y = 0
11     x = x + 1
12 print(summe)
```

Erweiterte Schleifen – Mehr-Dimensionale Datenstrukturen



V2C2.py | V2C3.py | V2C1.py | V2C4.py | V2C5.py

```
1  zahlen = [[[1,2,3],[4,5,6],[7,8,9]],[[10,11,12]
2
3  x = 0
4  y = 0
5  z = 0
6  summe = 0
7  while x < 3:
8      while y < 3:
9          summe = summe + zahlen[x][y]
10         y = y + 1
11     y = 0
12     x = x + 1
13 print(summe)
14
```

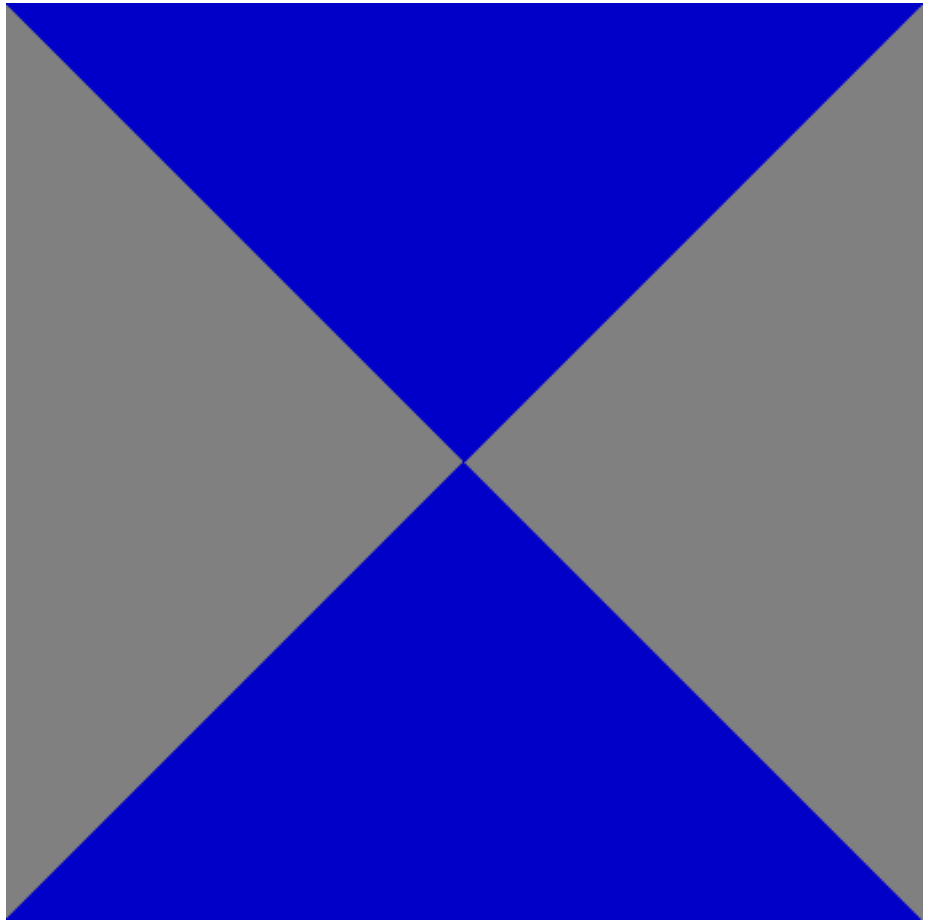
Von der Idee zum Programm

Aufgabenstellung

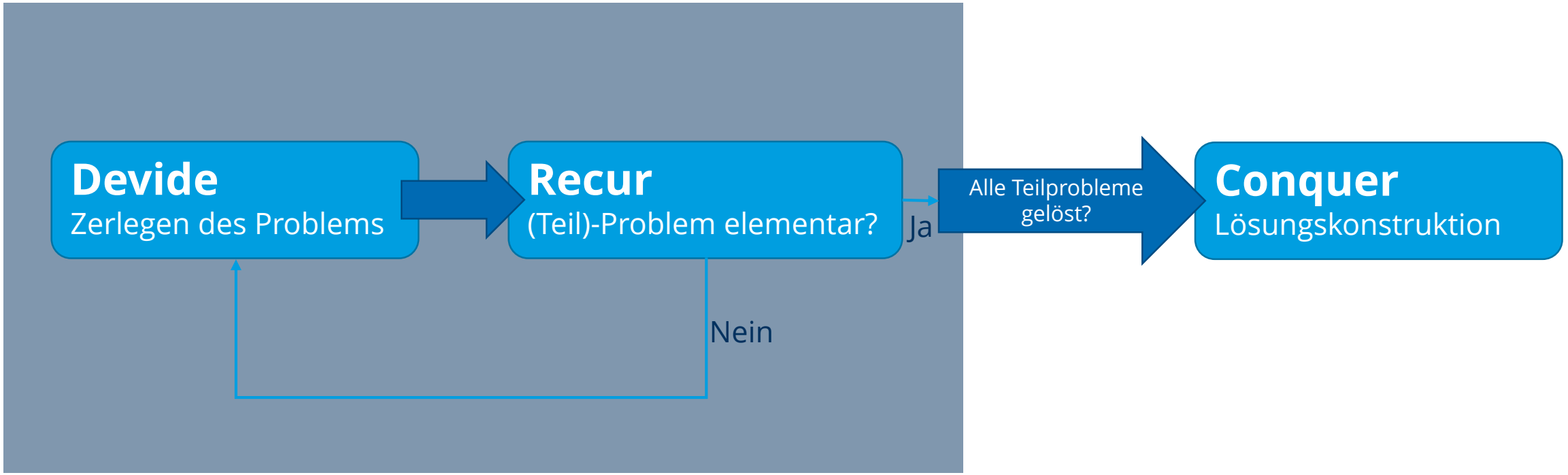
Schreiben Sie ein Programm, dass folgendes Bild zeichnet.

- Vorhandenes Wissen
  - Bilder erzeugen
  - Bild in eine Pixelliste umwandeln
  - Pixel verändern

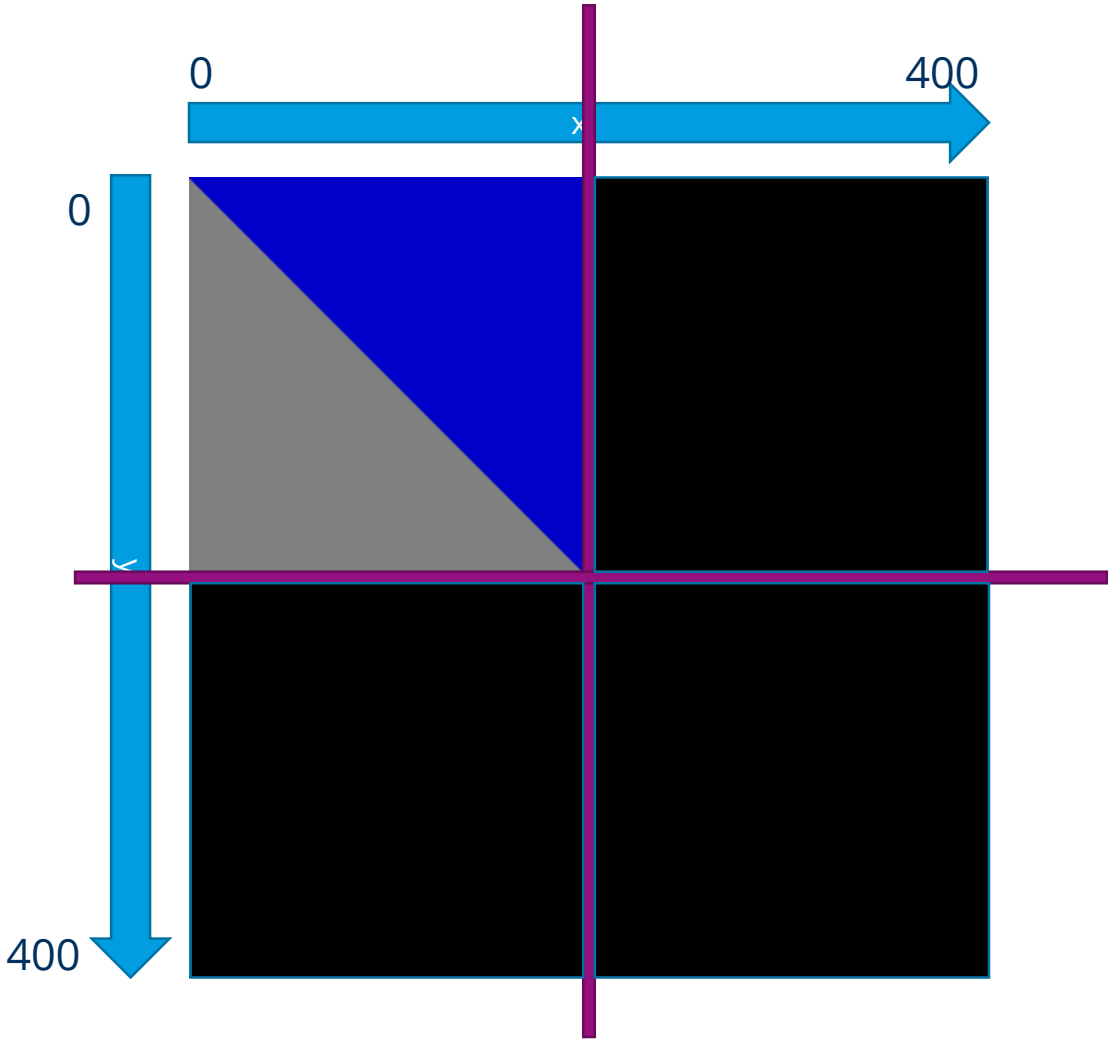
Welchen Ansatz wählen?



Divide and Conquer (Teile und (Be-)herrsche)



Zerlegen des Problems



Lösen Teilproblem 1

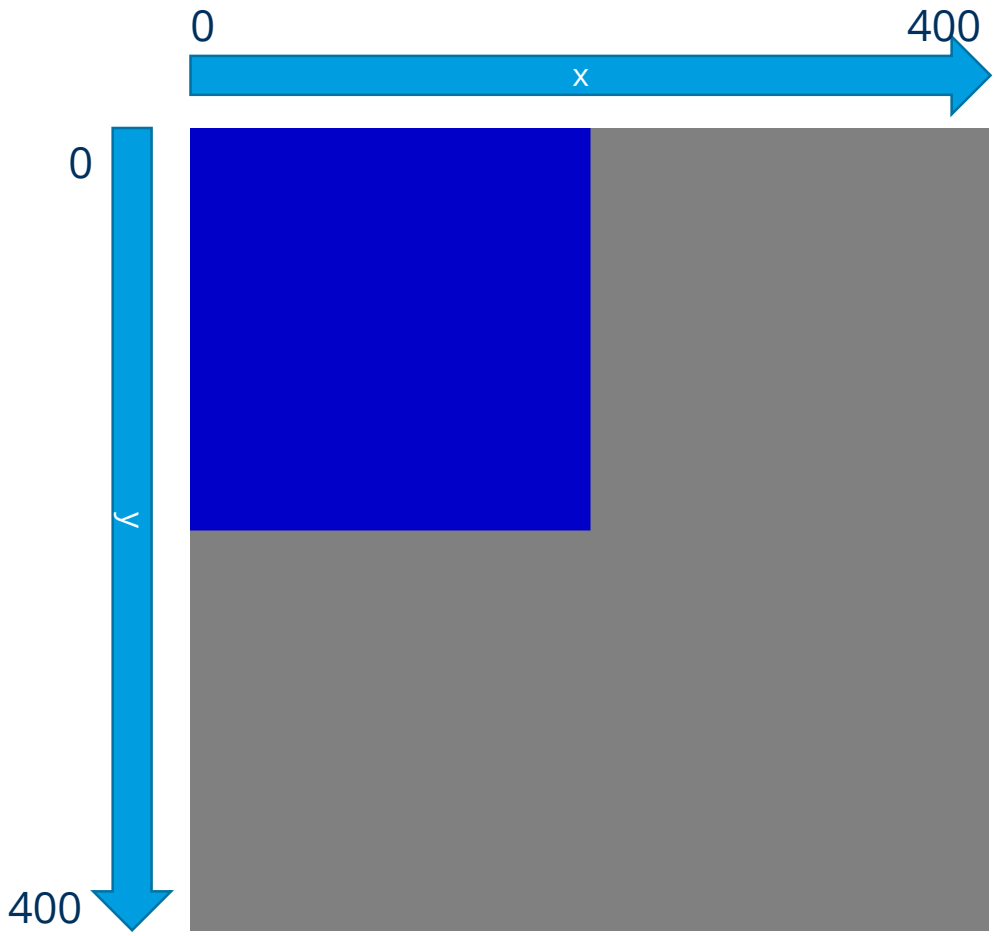
- Nehmen Sie sich 2 Minuten Zeit und versuchen Sie nachzuvollziehen was dieser Code an Ausgabe produziert.

```
1 from PIL import Image
2
3 img = Image.new("RGB", (400, 400), "grey")
4 pixel = img.load()
5
6 x,y = 0,0
7
8 while y < 200:
9     while x < 200:
10         pixel[x,y] = (0,0,200)
11         x = x + 1
12     x = 0
13     y = y + 1
14
15 img.show()
```

Lösen Teilproblem 1

Teilschritte:

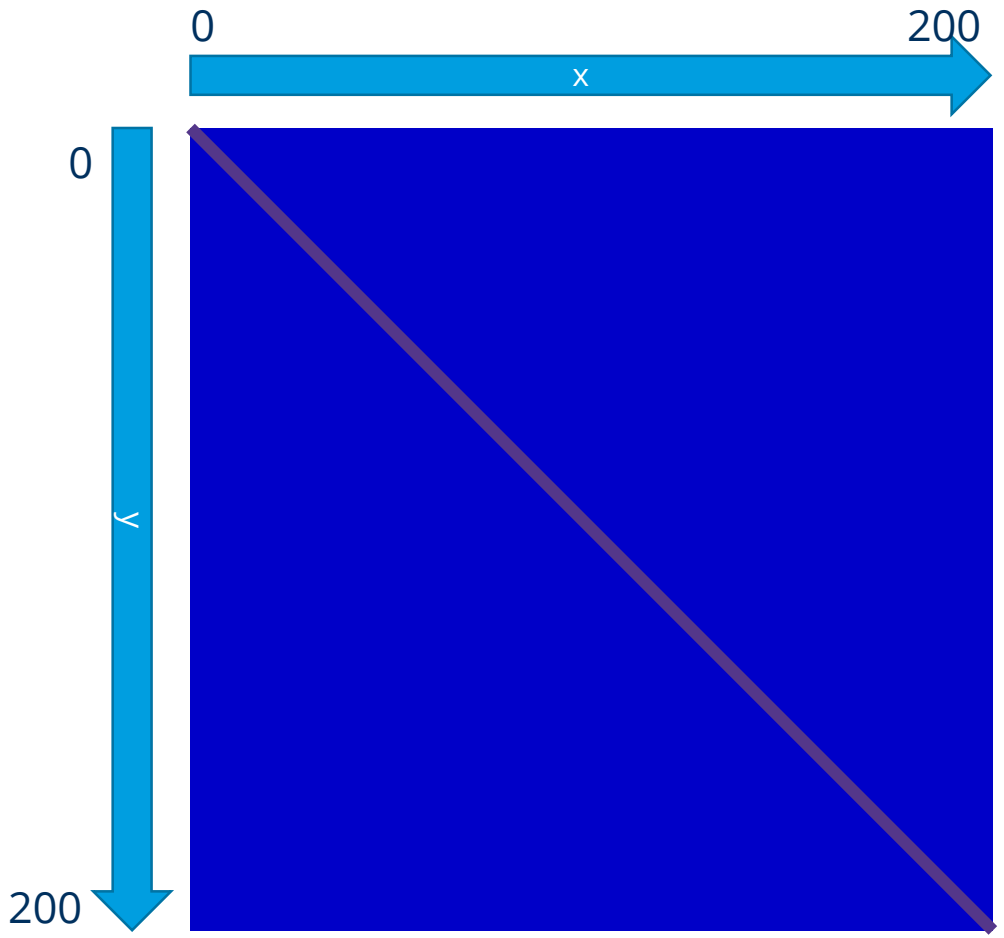
- Ausgangslage
- Betrachten des Teilproblems



Lösen Teilproblem 1

Teilschritte:

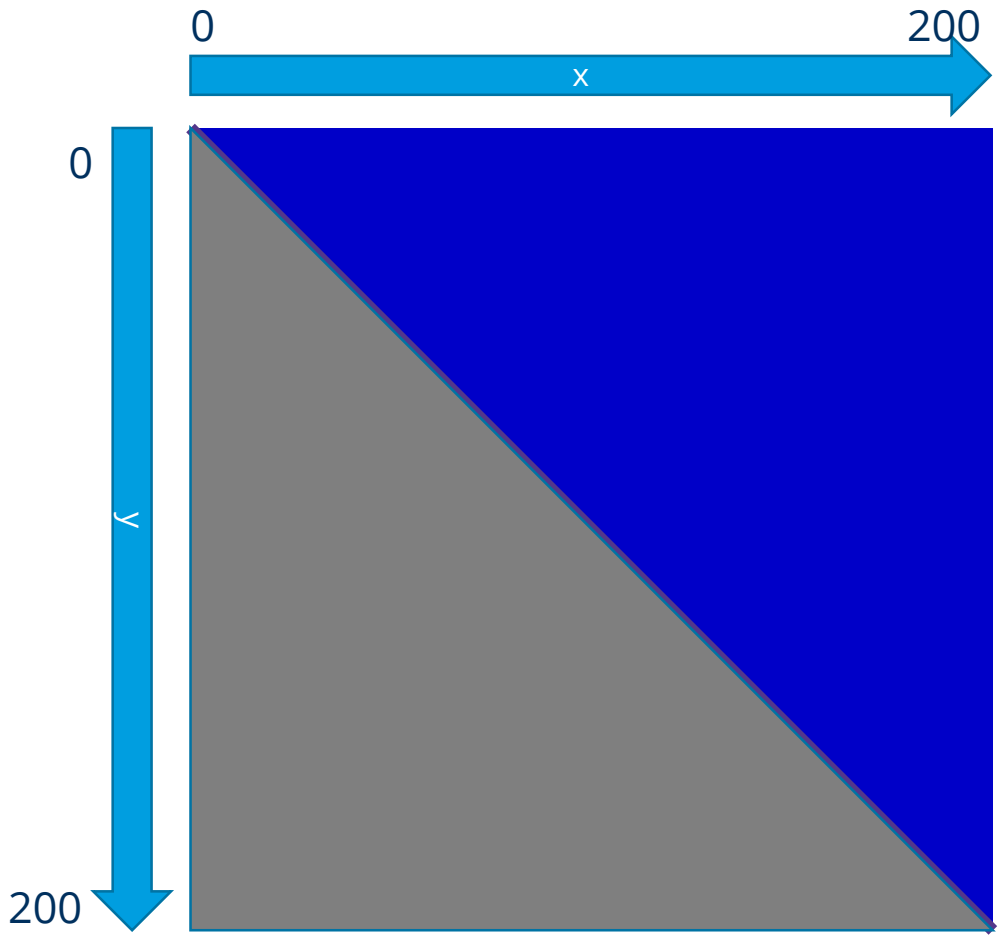
- Ausgangslage
- Betrachten des Teilproblems
  - Elementares Problem?
  - Zeilenweise oder spaltenweise Betrachtung



Lösen Teilproblem 1.1 – Zeilenweise Betrachtung

Teilschritte:

- Ausgangslage
- Betrachten des Teilproblems

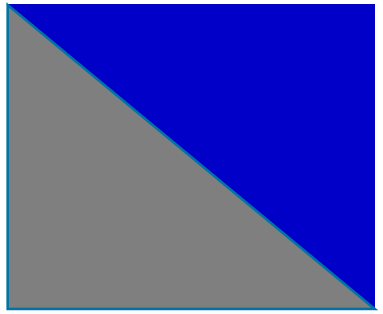


Teilproblem 1.1 – Rückblick

y = 0 erste Zeile von x = 0 bis x = 199 färben
y = 1 zweite Zeile von x = 0 bis x = 199 färben
y = 2 dritte Zeile von x = 0 bis x = 199 färben
y = 3 vierte Zeile von x = 0 bis x = 199 färben
...
y = 199 letzte Zeile von x = 0 bis x = 199 färben

```
1 from PIL import Image
2
3 img = Image.new("RGB", (400, 400), "grey")
4 pixel = img.load()
5
6 x,y = 0,0
7
8 while y < 200:
9     while x < 200:
10         pixel[x,y] = (0,0,200)
11         x = x + 1
12    x = 0
13    y = y + 1
14
15 img.show()
```

Lösen Teilproblem 1.1



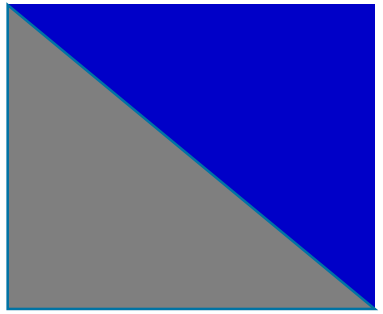
y = 0 erste Zeile von x = 0 bis x = 199 färben
y = 1 zweite Zeile von x = 1 bis x = 199 färben
y = 2 dritte Zeile von x = 2 bis x = 199 färben
y = 3 vierte Zeile von x = 3 bis x = 199 färben

...

y = 199 letzte Zeile von x = 199 bis x = 199 färben

```
1 from PIL import Image
2
3 img = Image.new("RGB", (400, 400), "grey")
4 pixel = img.load()
5
6 x,y = 0,0
7
8 while y < 200:
9     while x < 200:
10         pixel[x,y] = (0,0,200)
11         x = x + 1
12    x = 0
13    y = y + 1
14
15 img.show()
```

Lösen Teilproblem 1.1



y = 0 ist erste Zeile von x = 0 bis x = 199 färben
y = 1 ist erste Zeile von x = 1 bis x = 199 färben
y = 2 ist erste Zeile von x = 2 bis x = 199 färben
y = 3 ist erste Zeile von x = 3 bis x = 199 färben

...

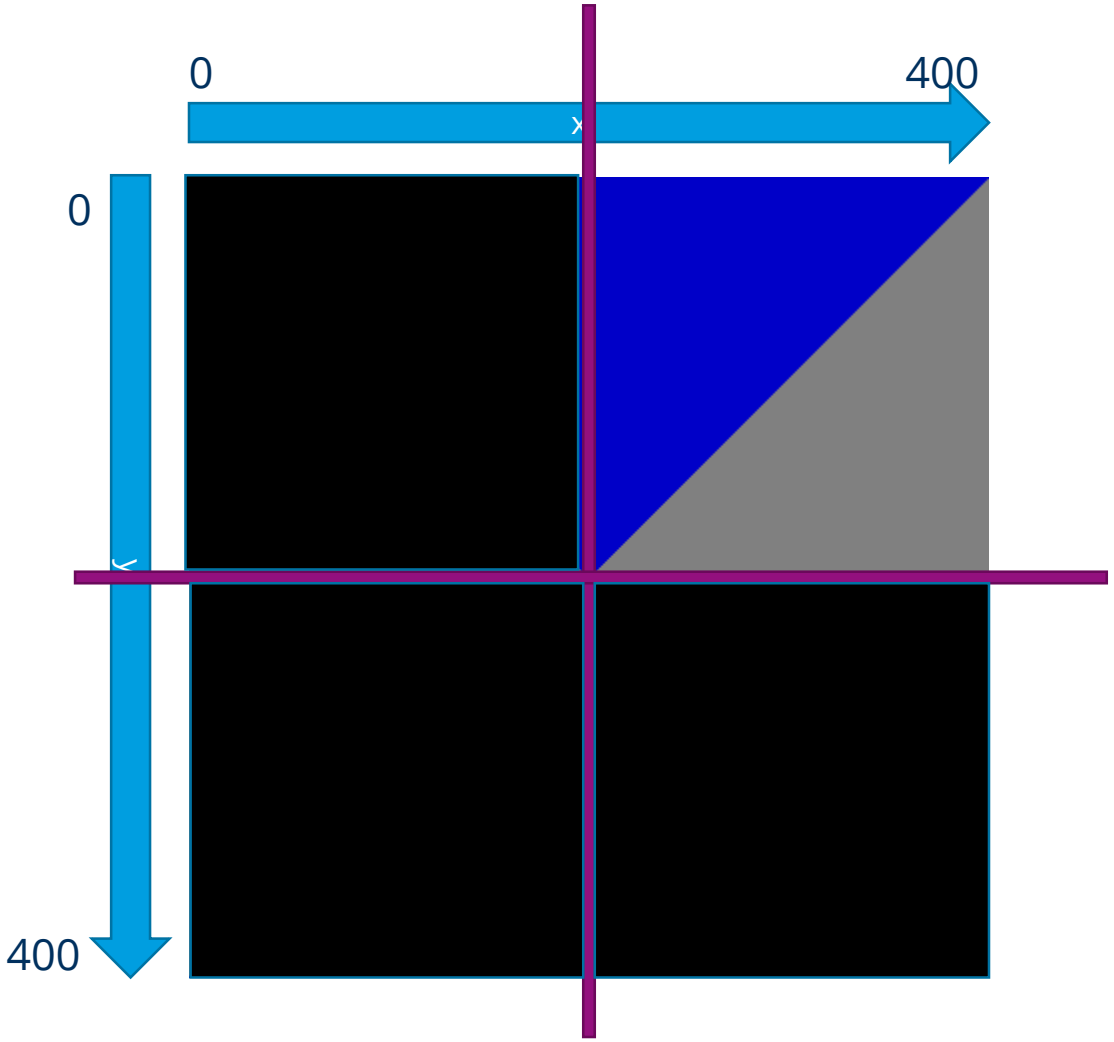
y = 199 ist erste Zeile von x = 199 bis x = 199 färben

```
1  x = 0
2  y = 0
3
4  while y < 200:
5
6      while x < 200:
7          pixel[x,y] = (0,0,200)
8          x = x + 1
9
10     x = 0
11     y = y + 1
```

Codebeispiel – Lösung Teilproblem 1.1

```
1  from PIL import Image
2
3  img = Image.new("RGB", (400, 400), "grey")
4  pixel = img.load()
5
6  x,y = 0,0
7
8  while y < 200:
9      x = y
10     while x < 200:
11         pixel[x,y] = (0,0,200)
12         x = x + 1
13     x = 0
14     y = y + 1
15
16 img.show()
```

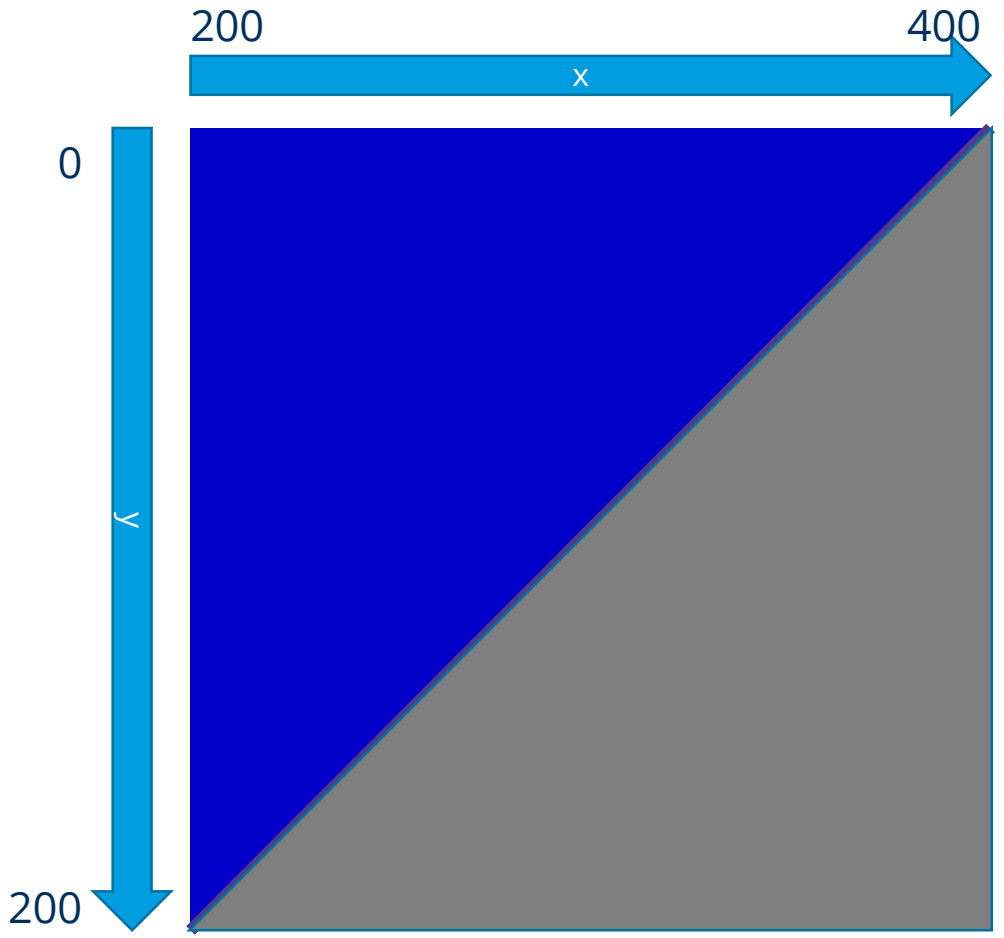
Lösen Teilproblem 2



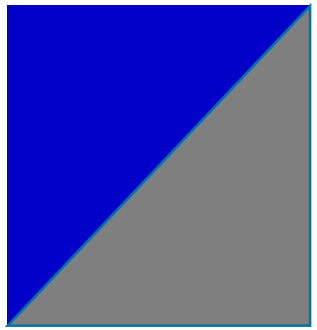
Lösen Teilproblem 2 – Zeilenweise Betrachtung

Teilschritte:

- Ausgangslage
- Betrachten des Teilproblems



Lösen Teilproblem 2



In Zeile y = 0 von x = 200 bis x = 399 färben
In Zeile y = 1 von x = 200 bis x = 398 färben
In Zeile y = 2 von x = 200 bis x = 397 färben

...

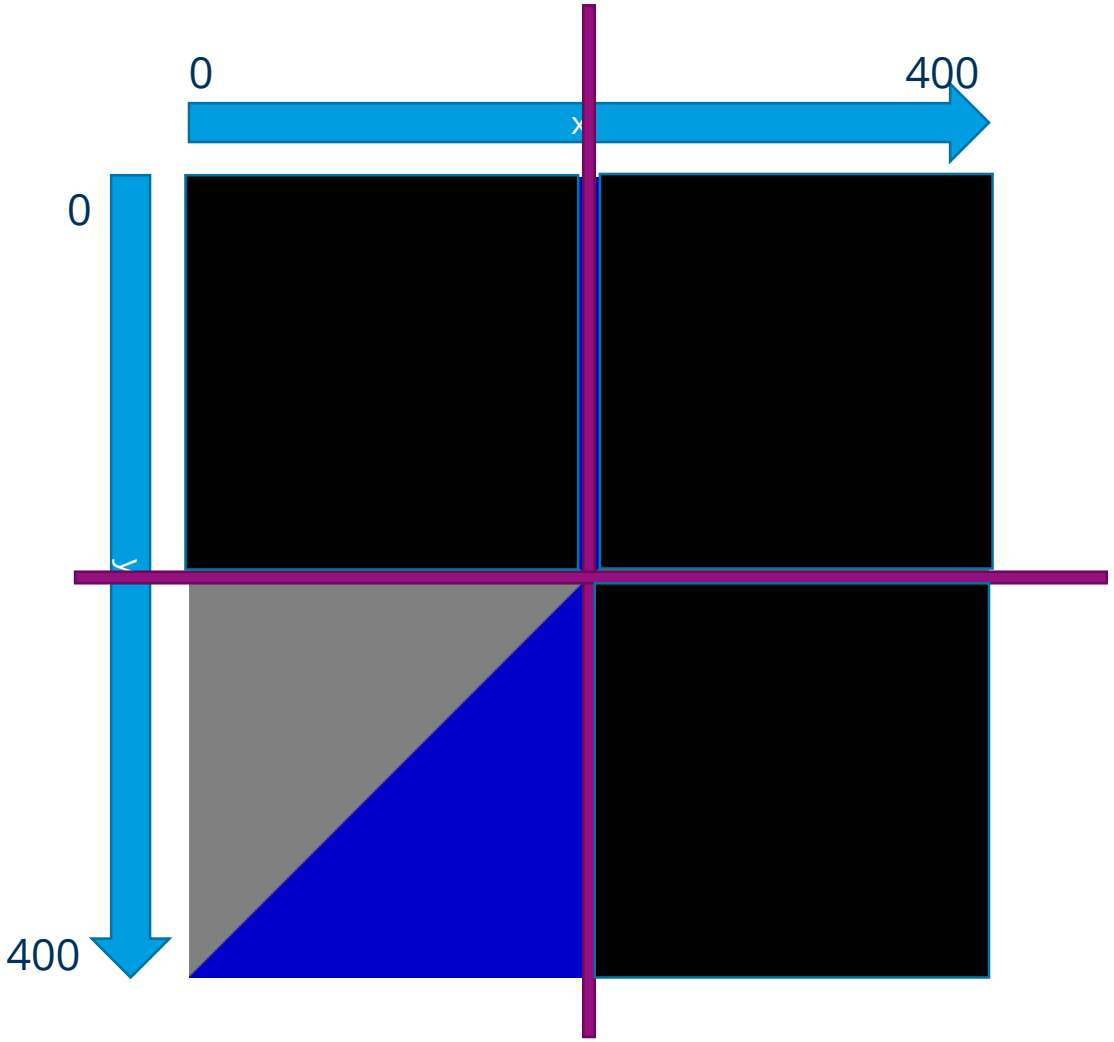
In Zeile y = 199 von x = 200 bis x = 200 färben

```
x = 0
y = 0

while y < 200:

    while x < 200:
        pixel[x,y] = (0,0,200)
        x = x + 1

    x = 0
    y = y + 1
```

Codebeispiel – Lösung Teilproblem 2

```
1  from PIL import Image
2
3  img = Image.new("RGB", (400, 400), "grey")
4  pixel = img.load()
5
6  x = 200
7  y = 0
8
9  while y < 200:
10     while x < 400 - y:
11         pixel[x,y] = (0,0,200)
12         x = x + 1
13     x = 200
14     y = y + 1
15
16 img.show()
```

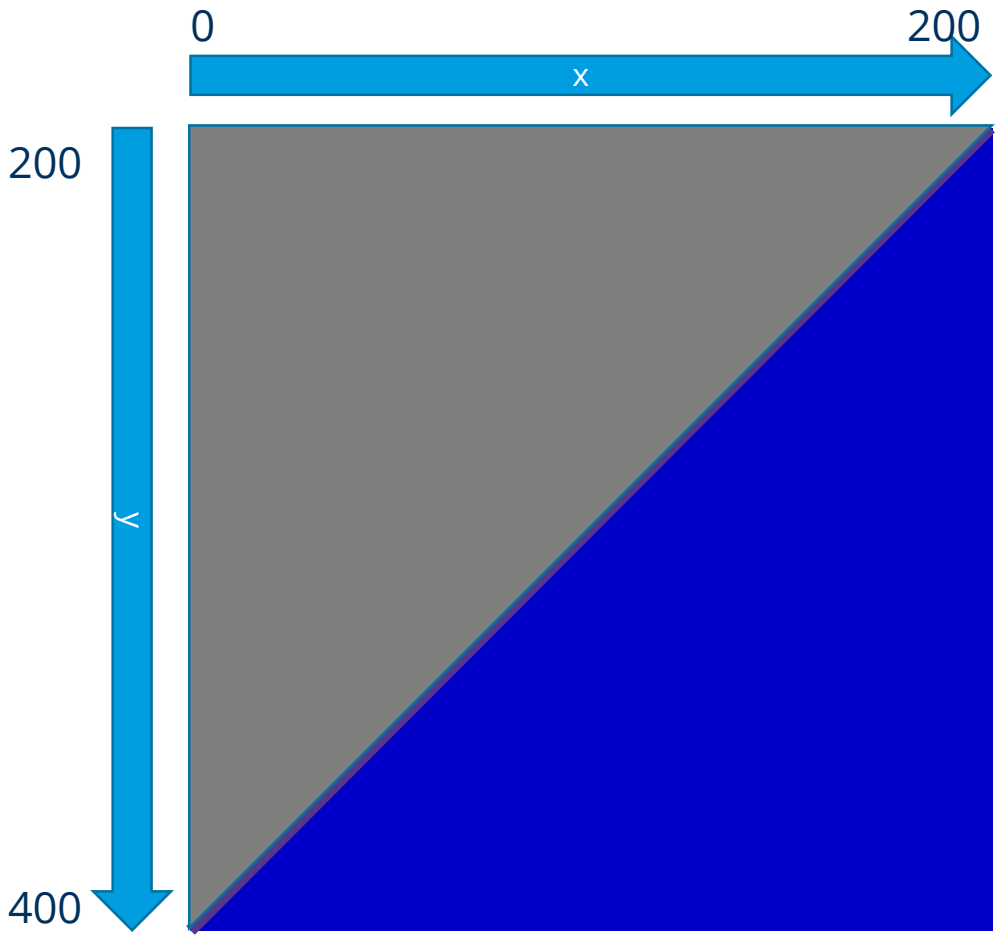
Lösen Teilproblem 3



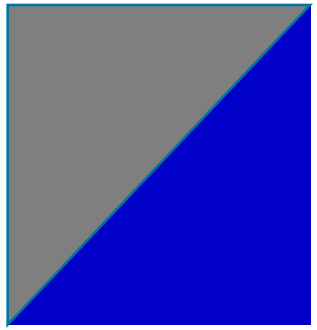
Lösen Teilproblem 3 – Zeilenweise Betrachtung

Teilschritte:

- Ausgangslage
- Betrachten des Teilproblems



Lösen Teilproblem 3



In Zeile y = 200 von x = 199 bis x = 199 färben
In Zeile y = 201 von x = 198 bis x = 199 färben
In Zeile y = 202 von x = 197 bis x = 199 färben

...

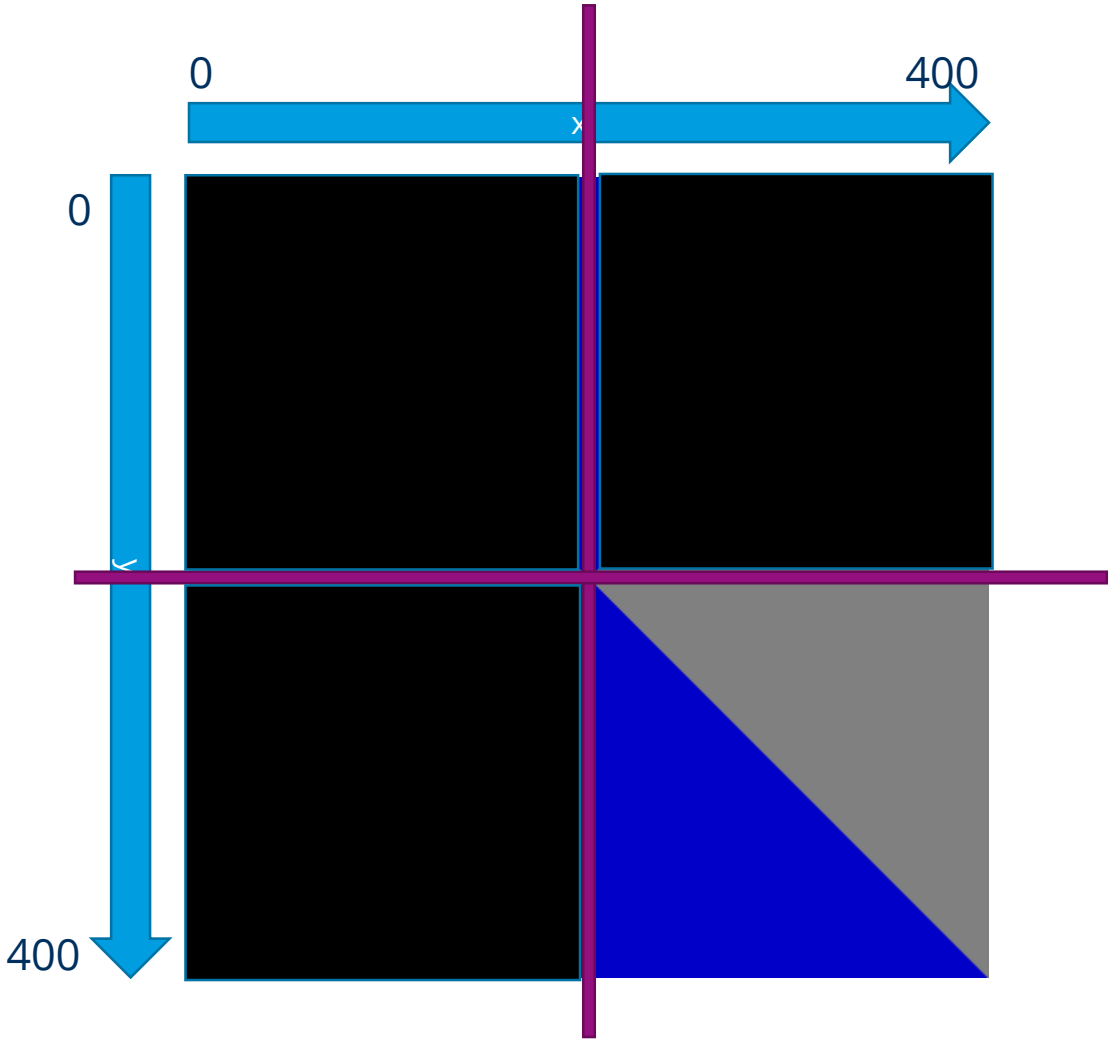
In Zeile y = 399 von x = 0 bis x = 199 färben

```
1 x = 0
2 y = 0
3
4 while y < 200:
5
6     while x < 200:
7         pixel[x,y] = (0,0,200)
8         x = x + 1
9
10    x = 0
11    y = y + 1
```

3Codebeispiel – Lösung Teilproblem 3

```
1  from PIL import Image
2
3  img = Image.new("RGB", (400, 400), "grey")
4  pixel = img.load()
5
6  x = 200
7  y = 0
8
9  while y < 200:
10     while x < 400 - y:
11         pixel[x,y] = (0,0,200)
12         x = x + 1
13     x = 200
14     y = y + 1
15
16 img.show()
```

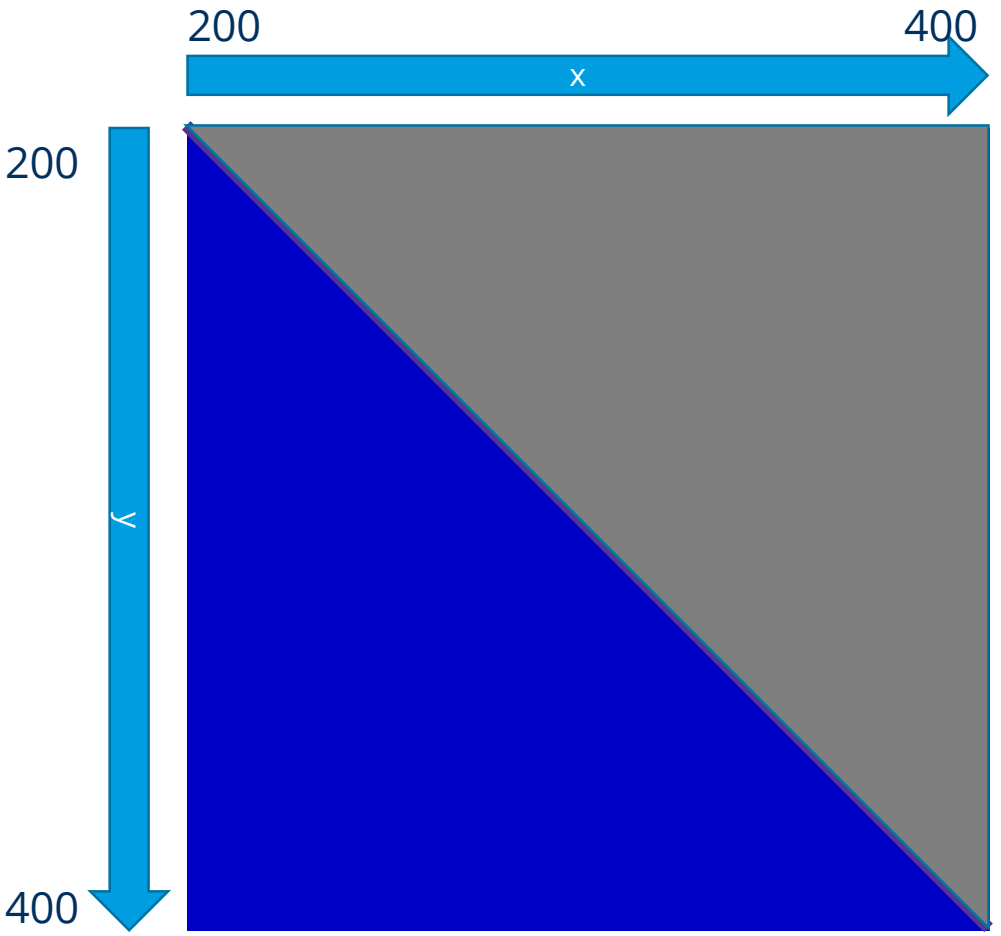
Lösen Teilproblem 4



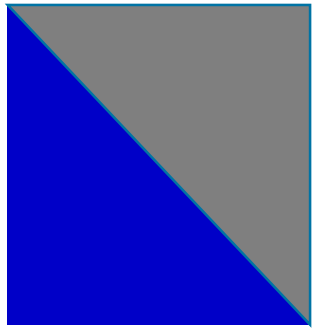
Lösen Teilproblem 4 – Zeilenweise Betrachtung

Teilschritte:

- Ausgangslage
- Betrachten des Teilproblems



Lösen Teilproblem 4



In Zeile y = 200 von x = 200 bis x = 200 färben
In Zeile y = 201 von x = 200 bis x = 201 färben
In Zeile y = 202 von x = 200 bis x = 202 färben

...

In Zeile y = 399 von x = 200 bis x = 399 färben

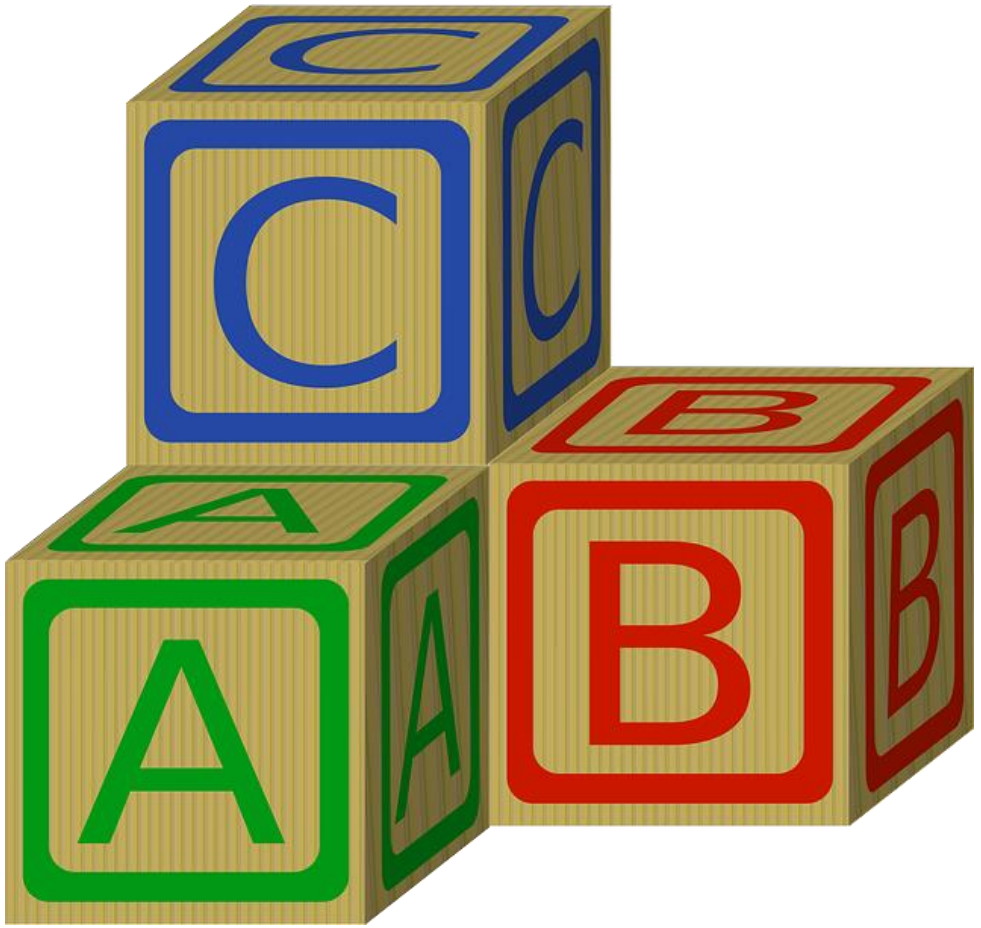
```
1 x = 0
2 y = 0
3
4 while y < 200:
5
6     while x < 200:
7         pixel[x,y] = (0,0,200)
8         x = x + 1
9
10    x = 0
11    y = y + 1
```

Codebeispiel – Lösung Teilproblem 4

```
1  from PIL import Image
2
3  img = Image.new("RGB", (400, 400), "grey")
4  pixel = img.load()
5
6  x = 200
7  y = 200
8
9  while y < 400:
10     while x < y:
11         pixel[x,y] = (0,0,200)
12         x = x + 1
13     x = 200
14     y = y + 1
15
16 img.show()
17
```

Zusammenführen der Lösungen - Conquer

- Zusammenfügen aller Lösungen in passender Reihenfolge
  - in dieser speziellen Aufgabe tatsächlich nicht relevant
  - nur relevante Code-Bausteine zusammenfügen



Codebeispiel – Komplettlösung

```
1  from PIL import Image
2
3  img = Image.new("RGB", (400, 400), "grey")
4  pixel = img.load()
5
6  x = 0
7  y = 0
8
9  while y < 200:
10     x = y
11     while x < 200:
12         pixel[x,y] = (0,0,200)
13         x = x + 1
14     x = 0
15     y = y + 1
16
17 x = 200
18 y = 0
19
20 while y < 200:
```

Funktionen

Definition

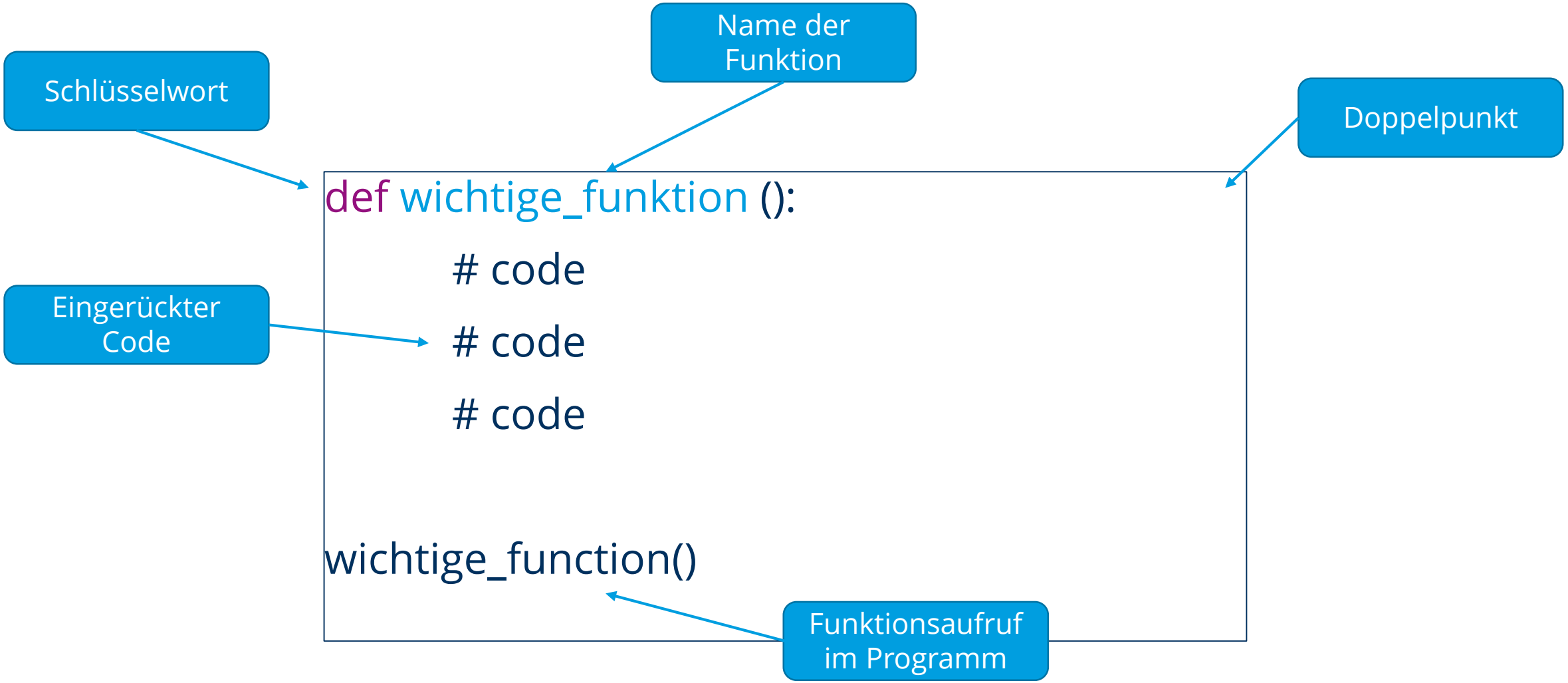
Eine Funktion ist ein Programmkonstrukt:

- dient zur Strukturierung des Codes
- sind sogenannte *Unterprogramme*
- haben den Vorteil der Wiederverwendbarkeit von Codebausteinen

Eigenschaften:

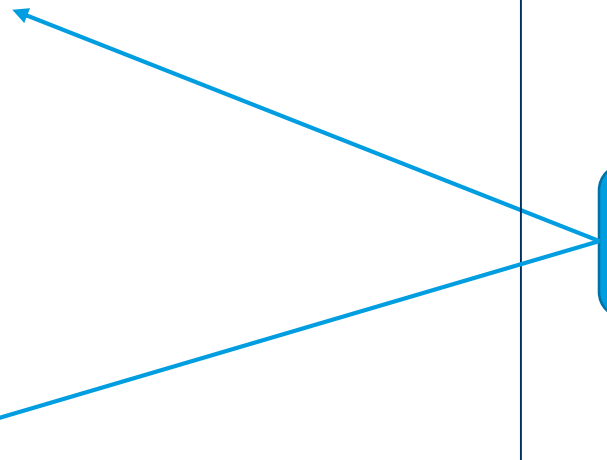
- Kann *Parameter* übergeben bekommen
- Kann ein *Resultat* der Ausführung zurückgeben

Funktionen in Python



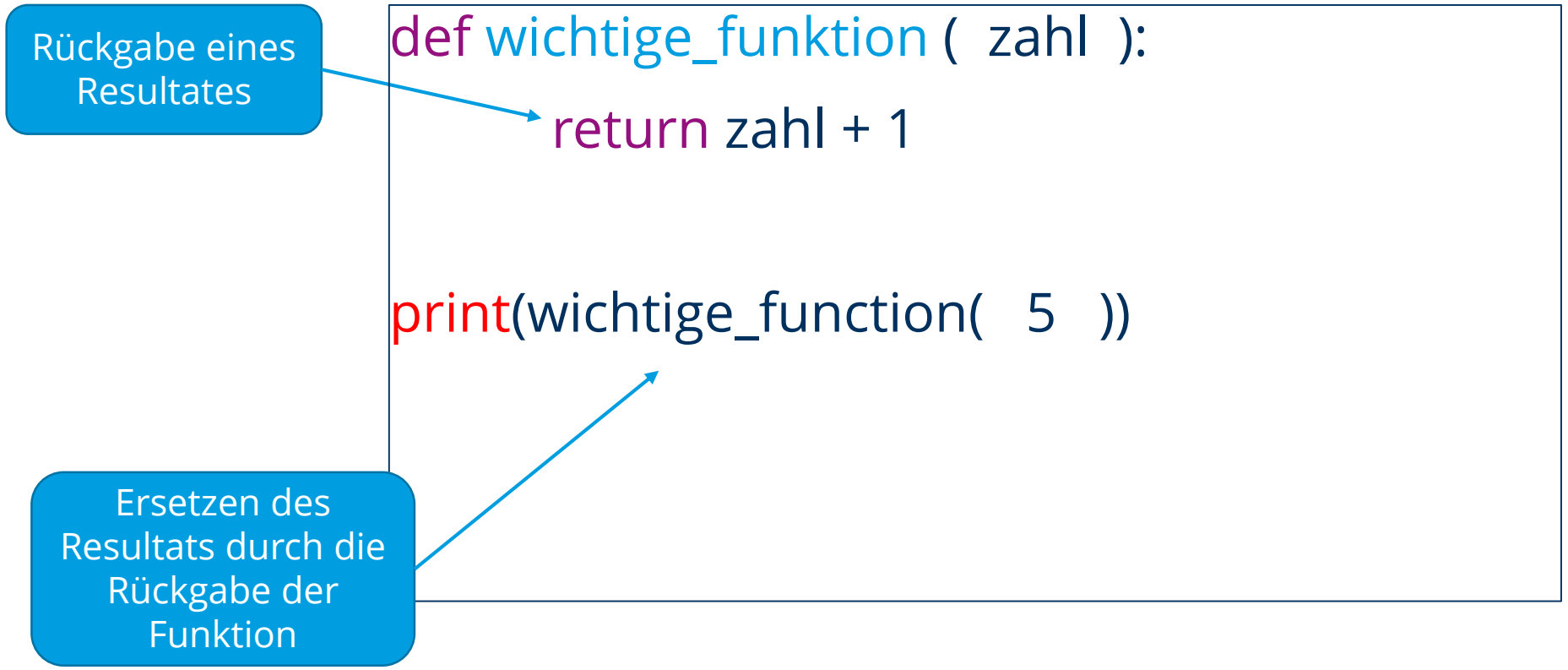
Funktionen in Python mit Parametern

```
def wichtige_funktion (      zahl      ):
        # code
        # code
        # code


wichtige_function(       5         )
```



Optionale Parameter Übergabe

Funktionen in Python mit Rückgabe



Codebeispiel – Funktionen

```python
# aufsummieren aller Elemente jeder Listen

liste1 = [1,2,3]
liste2 = [4,5,6]
liste3 = [7,8,9]

summe = 0
i = 0
while i < len(liste1):
    summe = summe + liste1[i]
    i = i + 1
print(summe)

summe = 0
i = 0
while i < len(liste2):
    summe = summe + liste2[i]
    i = i + 1
print(summe)

summe = 0
i = 0
while i < len(liste3):
    summe = summe + liste3[i]
    i = i + 1
print(summe)
```

Was Sie behalten und wiederholen sollten

- Was heißt ***Debuggen*** und wie funktioniert es?
- Wie funktioniert das ***Typcasting*** von Variablen?
- Was ist der ***Heap***?
  - Wie funktioniert er?
  - Was ist und wie funktioniert der ***Garbage Collector***?
- Wie funktionieren ***Schleifen*** in mehreren ***Dimensionen*** von Listen?
- Was ist der **D*evide and Conquer-Ansatz***?
  - Was sagt er aus und wie läuft er ab?
- *Funktionen kommen in der Übung und nächste Vorlesung wieder*