

Einführung in die Informatik WS 2025/2026

Universität Kassel

basierend auf dem MIT-Kurs
“Introduction to Computer Science and Programming Using Python“
von John Guttag, Eric Grimson und Ana Bell

Prof. Dr. Stefan Göller
Fachgebiet Theoretische Informatik/Komplexe Systeme

stefan.goeller@uni-kassel.de
<http://www.uni-kassel.de/eecs/fachgebiete/tiks/>

Moodle-Seite

- Fachbereich 16
- Fachgebiet Theoretische Informatik / Komplexe Systeme
- WS 25/26
- Kurs „Einführung in die Informatik“

oder einfach

<https://tinyurl.com/Eidl2526>

Kein Kennwort erforderlich!

Vorlesungsbetrieb

Uhrzeiten, Räume und sonstige Daten finden Sie im eCampus/HisPos.

- Die Vorlesungen findet mittwochs von 12 bis 14 Uhr und donnerstags von 10 bis 12 im Hörsaal 1603 ab dem 16.10.25 statt.
- Bitte seien Sie pünktlich.
- Bitte keine digitalen Aufzeichnungen (Fotos, Videos, Audio-Aufnahme oder ähnliches).
- Bitte nicht essen während der Vorlesung

Übungsbetrieb

- Der Übungsbetrieb beginnt am 17.10.25.
- Jede Woche erscheint ein Übungsblatt (Übungsblatt 0 ist ungewertet).
- Die Hausaufgaben sind bis zur Abgabefrist in der nächsten Woche abzugeben:
erstmalige Abgabe ist am 03.11.25 (Übungsblatt 1)
- Es wird 12 Übungsblätter zur bewerteten Abgabe geben.
- Es wird **keine** regulären Musterlösungen für die Aufgaben geben.
- Wenn Sie Fragen zur Lösung von Hausaufgaben haben, stellen Sie diese bitte in der Übung.
Bei dringenden Fragen, können Sie auch das Fragenforum nutzen (das Fragenforum ersetzt nicht die Übung).
- Keine digitalen Aufzeichnungen (Fotos, Videos, Audio-Aufnahme oder ähnliches).

Übungen

- **Ziel:** Besprechung der Übungsblätter (nach Abgabe)
- Verantwortliche: Anna Maiworm
- Übungsleiter: Tim Hagen, Anna Maiworm, Christian Rauch
- Jedes Aufgabenblatt enthält Präsenzaufgaben und Hausaufgaben
- Präsenzaufgaben werden in der Übung besprochen
- Präsenzaufgaben bereiten die Hausaufgaben vor
- Hausaufgaben werden nicht in der Übung besprochen, sind wöchentlich zu bearbeiten und einzureichen
- „Übungswoche“: von Freitag bis Donnerstag
- Eintragung in die Übungsgruppe mittels Moodle (ab Donnerstag, 16.10.25, 12.00 möglich)
- Übungsblätter:
 - können in Gruppen bis **zu drei Studierenden** bearbeitet werden, die Namen und Matrikelnummern müssen in jeder Abgabe angegeben werden
 - **Ausgabe** der Aufgabenblätter immer **donnerstags**
 - **Abgabe** der Aufgabenblätter mindestens eine Woche nach Ausgabe über Moodle
 - **Korrektur** Ihrer Abgaben möglichst innerhalb einer Woche nach Abgabe

Abgabe der Übungsblätter

- Die Abgabe erfolgt digital über moodle
- Abgabe als separate .py-Datei für jede Aufgabe (beachten Sie die genauen Hinweise auf dem jeweiligen Übungsblatt)
- Die Abgabe ist mit einer Deadline versetzt, verspätete Abgaben können nicht akzeptiert werden
- Die Abgabe erfolgt in 2er-3er Gruppen (nicht mehr), wobei nur eine Person die Abgabe hochlädt
- Nutzen Sie gerne die **Partnerbörse** um Partner für die Abgabe zu finden
- In jeder Datei der Abgabe müssen die Namen aller Gruppenmitglieder stehen. Nur dann können auch die Punkte an alle Gruppenmitglieder vergeben werden
- **Schreiben Sie auf jede Abgabe immer die Namen und die Matrikelnummer aller Studierenden Ihrer Abgabegruppe!**

Abschreiben und ähnliches



- Bei erkennbarem Abschreiben zwischen zwei oder mehr Gruppen erhalten alle Mitglieder der beteiligten Gruppe 0 Punkte. Wir unterscheiden dabei nicht wer von wem abgeschrieben hat. Wir behalten uns vor die Studienleistung in einem solchen Fall nicht zu gewähren und weisen auf rechtliche Konsequenzen hin.
- Verwendung von generative KI (wie z.B. ChatGPT) oder das Abschreiben/Kopieren/Verwenden von Lösungen aus dem Internet (oder woher auch immer) wird genauso behandelt.

Studienleistung

- Mindestens 50% aller Punkte der ersten 6 Aufgabenblätter
- und mindestens 50% aller Punkte der letzten 6 Aufgabenblätter
- und mindestens 50% aller Punkte in der schriftlichen Prüfung

wird ersetzt durch:
50% der ersten 8 Aufgabenblätter
falls Sie eine 6-Credit-Prüfung ablegen

Zeitplan Vorlesungen

	Mi 05.11.25 Do 06.11.25	Mi 03.12.25 Do 04.12.25	Winterpause	Mi 04.02.26 Do 05.02.26
Do 16.10.25	Mi 12.11.25 Do 13.11.25	Mi 10.12.25 Do 11.12.25	Mi 14.01.26 Do 15.01.26	Mi 11.02.25 Do 12.02.25
Mi 22.10.25 Do 23.10.25	Mi 19.11.25 Do 20.11.25	Mi 17.12.25 Do 18.12.25	Mi 21.01.26 Do 22.01.26	
Mi 29.10.25 Do 30.10.25	Mi 26.11.25 Do 27.11.25	Winterpause	Mi 28.01.26 Do 29.01.26	

Zeitplan Übungen

	Fr 31.10.25 Übungsblatt 2 Do 06.11.25		
	Fr 07.11.25 Übungsblatt 3 Do 13.11.25	Fr 05.12.25 Übungsblatt 7 Do 11.12.25	Fr 30.01.26 Übungsblatt 12 Do 05.02.26
	Fr 14.11.25 Übungsblatt 4 Do 20.11.25	Fr 12.12.25 Übungsblatt 8 Do 18.12.25	Fr 06.02.26 Fragestunde Do 12.02.26
Fr 17.10.25 Übungsblatt 0 Do 23.10.25	Fr 21.11.25 Übungsblatt 5 Do 27.11.25	Winterpause	Fr 19.12.24 Übungsblatt 9 Do 15.01.26
Fr 24.10.25 Übungsblatt 1 Do 30.10.25	Fr 28.11.25 Übungsblatt 6 Do 04.12.25	Winterpause	Fr 16.01.26 Übungsblatt 10 Do 22.01.26
			Fr 22.01.26 Übungsblatt 11 Do 29.01.26

Erste Übungshälfte

jeweils 50% aller
Punkte notwendig*

Zweite Übungshälfte

6 ECTS-Variante
Übungsblatt 1-8
*

Übungstermine

- **Freitag: 10.15 - 11.45** (Tutor: Tim Hagen) **ab 17.10.25** WA 71 / HS 2104
- **Freitag: 12.15 - 13.45** (Tutor: Tim Hagen) **ab 17.10.25** Neubau / Raum -1607
- **Montag: 10.15 - 11.45** (Tutorin: Christian Rauch) **ab 20.10.25** WA 73 / Raum -1319
- **Montag: 14.15 - 15.45** (Tutorin: Anna Maiworm) **ab 20.10.25** Neubau / Raum -1607
- **Do: 14.00 - 16.00** (Tutor: Anna Maiworm) **ab 23.10.25** WA 73 / Raum -1418

Literatur

- ***Introduction to Computation and Programming Using Python: With Application to Understanding Data*** von John V. Guttag (The MIT Press), 2. Auflage.
(10 Exemplare sind in der Bibliothek verfügbar)
- *Introduction to Computer Science and Programming in Python* MIT-Kurs von Ana Bell, Eric Grimson und John Guttag.
- *Learning Python* von Mark Lutz & David Ascher (O'Reilly), 5. Auflage.
(5 Exemplare sind in der Bibliothek verfügbar)
- *Python documentation* von Python Software Foundation erhältlich unter
<https://docs.python.org/3/>
- *Tutorialspoint (für Python)*
www.tutorialspoint.com/python

Diese Veranstaltung ist stark an **dieser** Vorlesung am MIT orientiert!

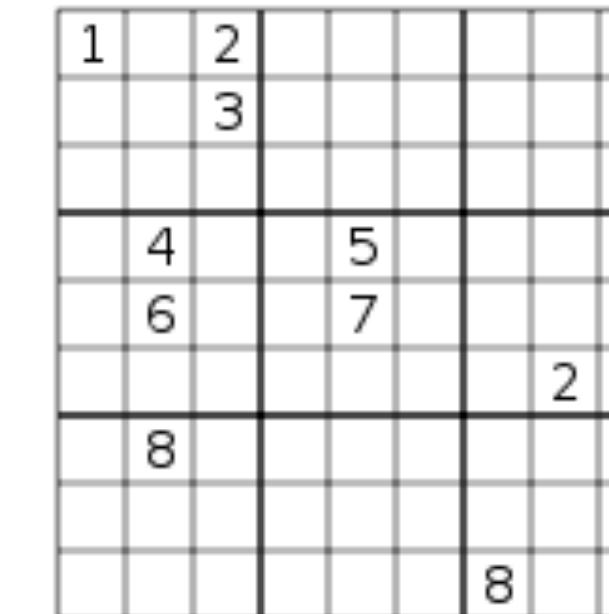
Ein paar Tipps/Bitten

- Bereiten Sie die Vorlesungen nach.
- Programmieren Sie immer alle Aufgaben selbst, selbst wenn Sie Ihr Aufgabenblatt später als Gruppe abgeben.
- Stellen Sie Fragen (in der Fragestunde, im Moodle-Forum, vor allem aber in der Übung).
- Fragen Sie nochmal, wenn Sie es immer noch nicht verstanden haben.
- Unterhalten/chatten/zoom/skypen... Sie (sich) mit Ihren Kommilitoninnen und Ihren Kommilitonen.
- Arbeiten Sie nicht alleine!
- Lassen Sie sich nicht einschüchtern von Leuten, die bereits mehr Programmiererfahrung haben als Sie.
- Haben Sie Spass an Informatik, an den Konzepten, den Ideen.
- Probieren Dinge aus, verwenden das Erlernte im Alltag!
- Lesen Sie die Aufgabenstellungen genau durch und beachten Sie das Abgabeformat!
- Seien Sie freundlich in den Chats/in den Foren/ und in den Übungsgruppen!



In dieser Veranstaltung lernen Sie unter anderem...

- ... Sudokus algorithmisch zu lösen.



- ... Gewinnstrategien für Spiele zu berechnen

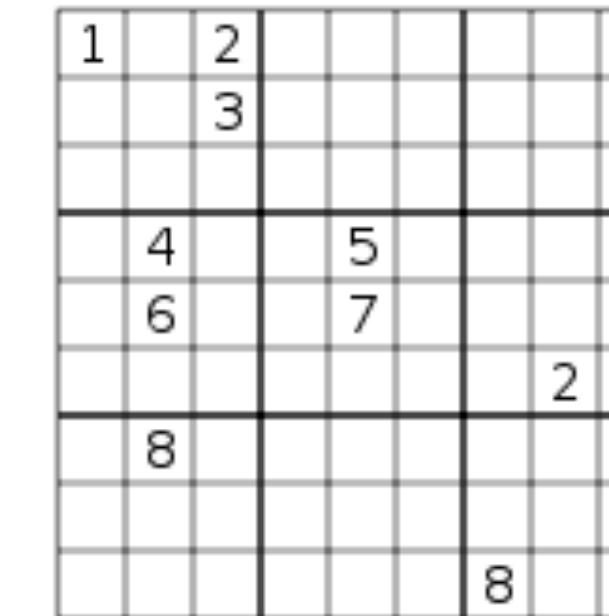


- ... ein Programm zu entwickeln, das Ihnen die schnellste Verbindung im Kasseler Verkehrsnetz berechnet.



In dieser Veranstaltung lernen Sie unter anderem...

- ... Sudokus algorithmisch zu lösen.



- ... **Gewinnstrategien für Spiele zu berechnen**

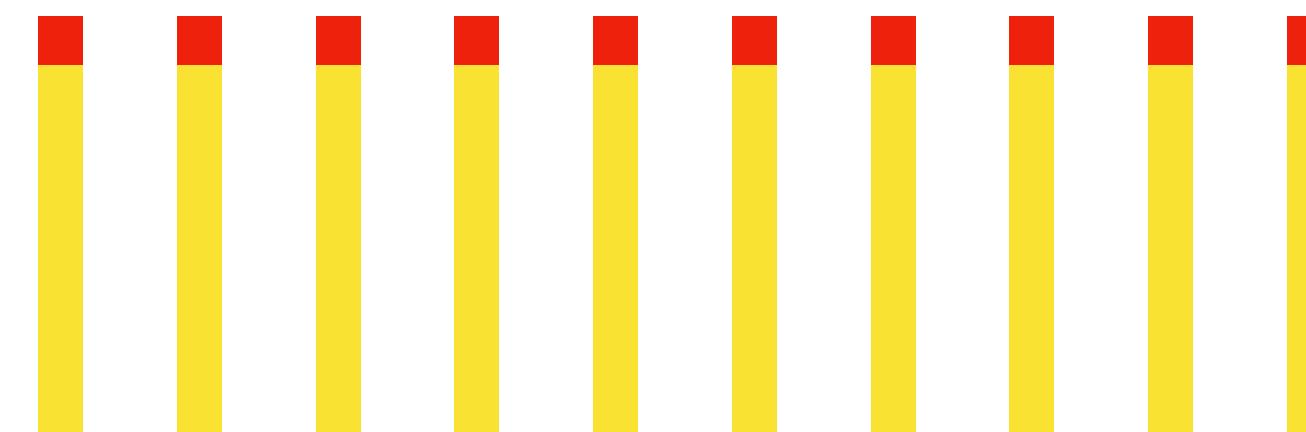


- ... ein Programm zu entwickeln, das Ihnen die schnellste Verbindung im Kasseler Verkehrsnetz berechnet.



Streichholzspiel

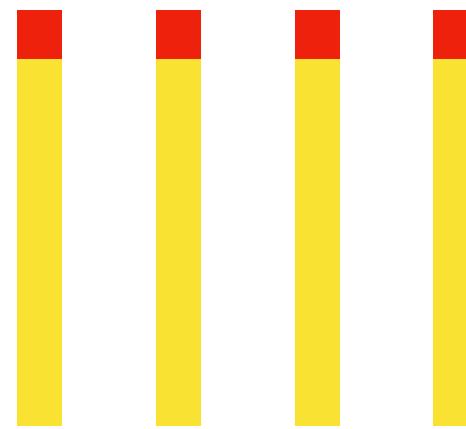
Gegeben sei ein Haufen Streichhölzer



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Streichholzspiel

Gegeben sei ein Haufen Streichhölzer



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Streichholzspiel

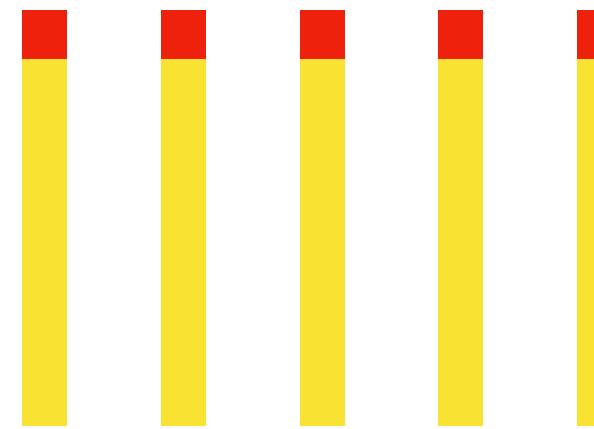
Gegeben sei ein Haufen Streichhölzer



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Streichholzspiel

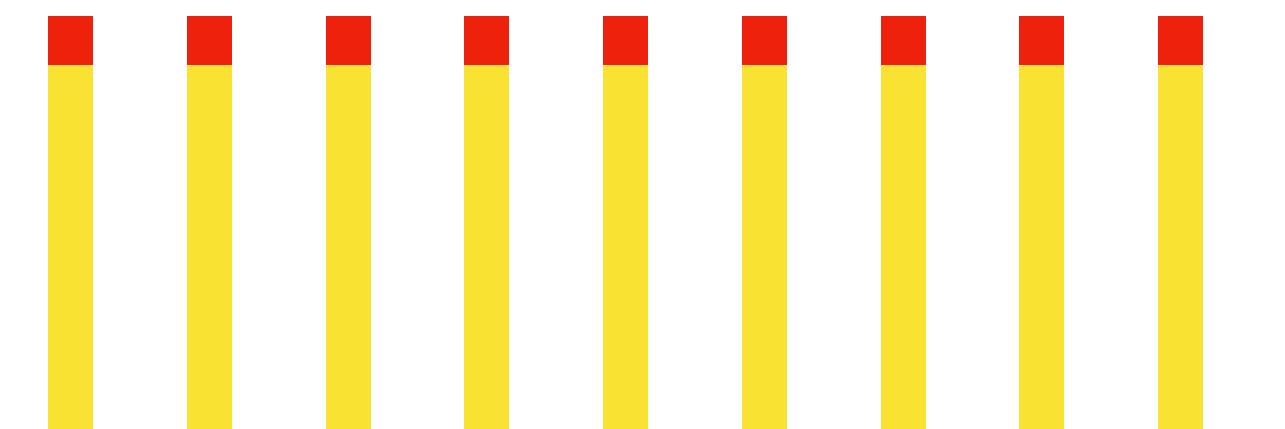
Gegeben sei ein Haufen Streichhölzer



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Streichholzspiel

Gegeben sei ein Haufen Streichhölzer



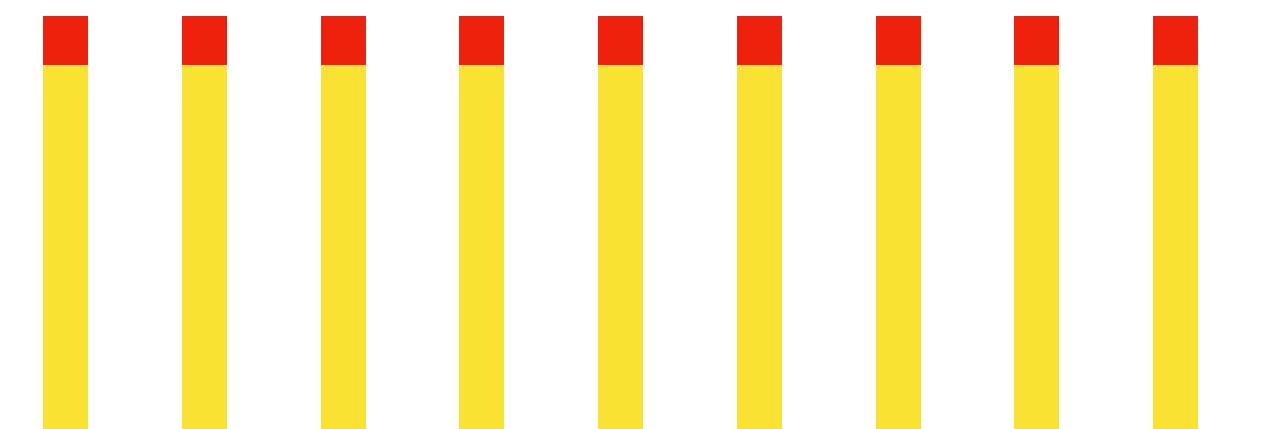
- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Grundsätzliche Fragen:

- Kann ich **immer** gewinnen? **Nein, nicht immer.**
- **Wann** kann ich gewinnen? **Wenn nicht $4n+1$ Streichhölzer übrig sind!**
- **Wie** kann ich gewinnen? **Entferne Streichhölzer so, dass $4n+1$ Streichhölzer übrig sind!**

Streichholzspiel

Gegeben sei ein Haufen Streichhölzer



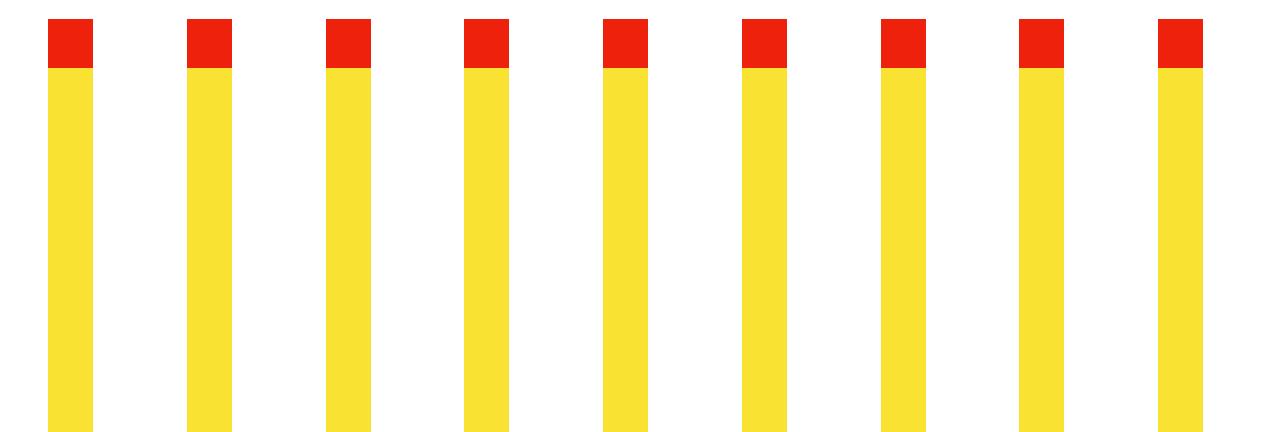
- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Grundsätzliche Fragen:

- Kann ich **immer** gewinnen? **Nein, nicht immer.**
- **Wann** kann ich gewinnen? **Wenn nicht $4n+1$ Streichhölzer übrig sind!**
- **Wie** kann ich gewinnen? **Entferne Streichhölzer so, dass $4n+1$ Streichhölzer übrig sind!**

Streichholzspiel

Gegeben sei ein Haufen Streichhölzer



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin, entweder **ein, zwei oder drei** Streichhölzer zu **entfernen**
- Der Spieler, der **nicht** das letzte Streichholz entfernt, **gewinnt**

Grundsätzliche Fragen:

- Kann ich **immer** gewinnen? **Nein, nicht immer.**
- **Wann** kann ich gewinnen? **Wenn nicht $4n+1$ Streichhölzer übrig sind!**
- **Wie** kann ich gewinnen? **Entferne Streichhölzer so, dass $4n+1$ Streichhölzer übrig sind!**

Schokoladenspiel

Gegeben sei eine Tafel Schokolade, Stück links oben ist vergiftet



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin ein Teil **abzuschneiden**: Fahre mit Messer entlang der Linien erst nach links und dann nach unten
- Der unvergiftete Spieler **gewinnt**

Schokoladenspiel

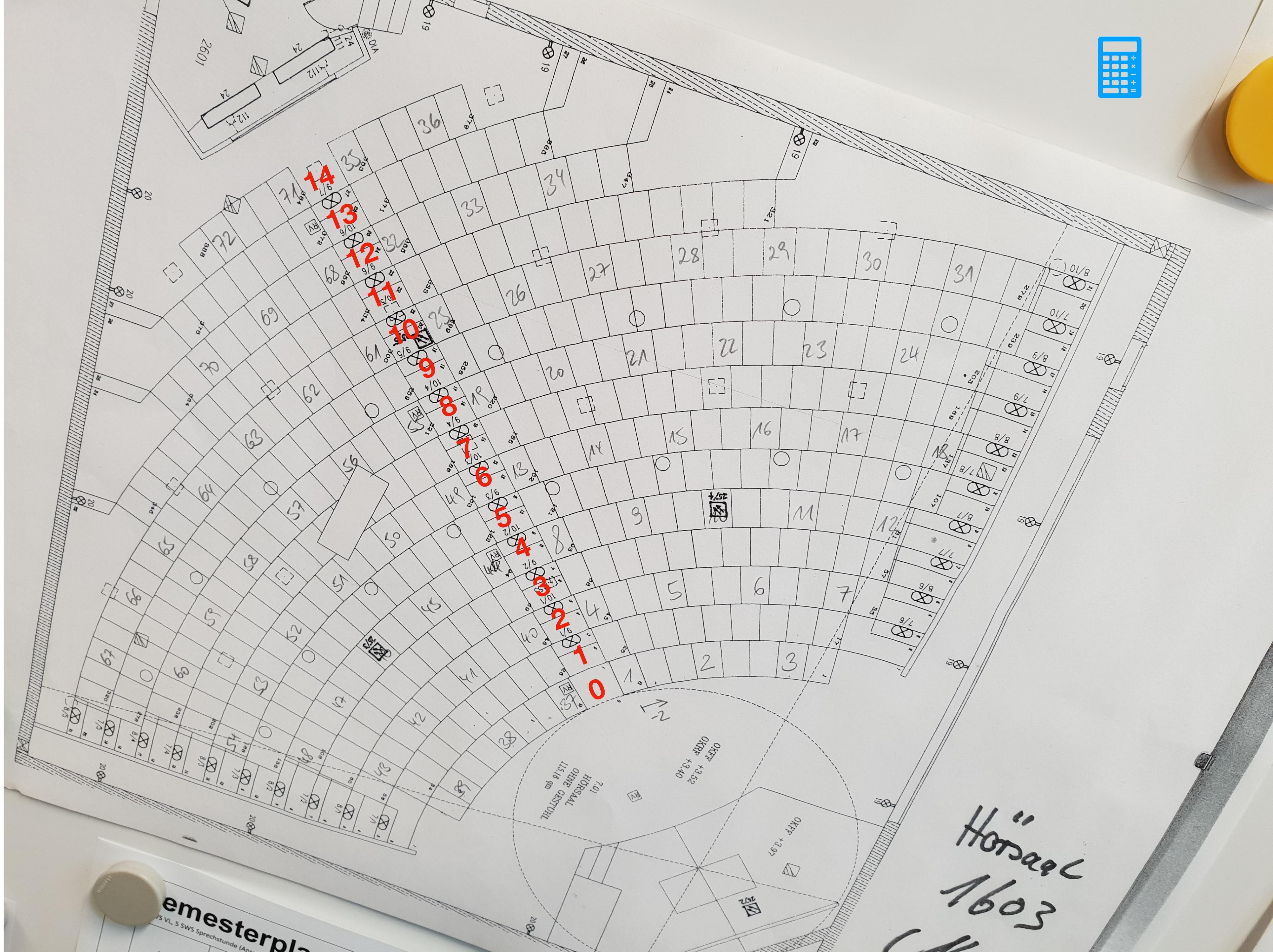
Gegeben sei eine Tafel Schokolade, Stück links oben ist vergiftet



- Das Spiel besteht aus **zwei Spielern**
- Die Spieler ziehen **abwechselnd**
- Ein **Zug** besteht darin ein Teil **abzuschneiden**: Fahre mit Messer entlang der Linien erst nach links und dann nach unten
- Der unvergiftete Spieler **gewinnt**

Grundsätzliche Fragen:

- Kann ich **immer** gewinnen? **Nein, nicht immer!**
- **Wann** kann ich gewinnen? **Wenn Tafel keine 1x1-Tafel ist**
- **Wie** kann ich gewinnen? **Bis heute keine simple Formel bekannt!**
Systematisches Durchprobieren=Backtracking



Teilgebiete der Informatik

- **Technische Informatik**

- Technische Realisierung von Rechnern, Rechnernetzen und eingebetteten Systemen

- **Praktische Informatik**

- Programmierung von Rechnern und Entwicklung der dafür benötigten Sprachen, Werkzeuge, Methoden...

- **Theoretische Informatik**

- Berechenbarkeit, Komplexität von Problemen, Korrektheit von Programmen

- **Angewandte Informatik**

- Anwendung der Kerninformatik auf spezielle Bereiche wie bspw. Wirtschaftsinformatik, Bioinformatik, Simulation, wissenschaftliches Rechnen

- **Informatik und Gesellschaft**

- Voraussetzungen, Wirkungen und Folgen der Informatik in der Gesellschaft, z.B. verantwortliche Technikgestaltung, Rechtsfragen

Tätigkeiten einer Informatikerin/ eines Informatikers (ein Auszug)

- Verstehen des Anwendungsgebiets und der Problemstellung
- Gespräche mit (fachfremden) Kunden bzw. Beteiligten
- Machbarkeitsstudien, Erarbeiten von Spezifikationen
- Erarbeiten von Lösungskonzepten, Strukturierung
- Softwareentwurf
- Formale Korrektheit des Entwurfs bzw. konzipierter Algorithmen
- Implementierung
- Testen, Experimentieren, Fehlersuche
- Wiederverwendung/Wartung existierender Software
- Installation, Administration

Vorläufige Themenübersicht

- Grundlagen der Programmiersprache Python
- Vollständige Induktion und Rekursion
- Backtracking
- Teile-und-Herrsche Verfahren
- Aufwandsanalyse
- Sortieralgorithmen
- ggf. Dynamische Programmierung
- Objektorientierung

Was macht ein Computer?

- Grundsätzlich...
 - ... führt ein Computer **Berechnungen** durch
(rund 1 Milliarde Berechnungen pro Sekunde)
 - ... speichert ein Computer **Ergebnisse**
(bis zu hunderte Gigabyte Speicher oder mehr)
- Was für **Berechnungen**?
 - **Voreingebaute** Berechnungen
(wie z.B. Integer-Multiplikation, XOR)
 - Berechnungen, die **Sie als Programmiererin/Programmierer** definieren

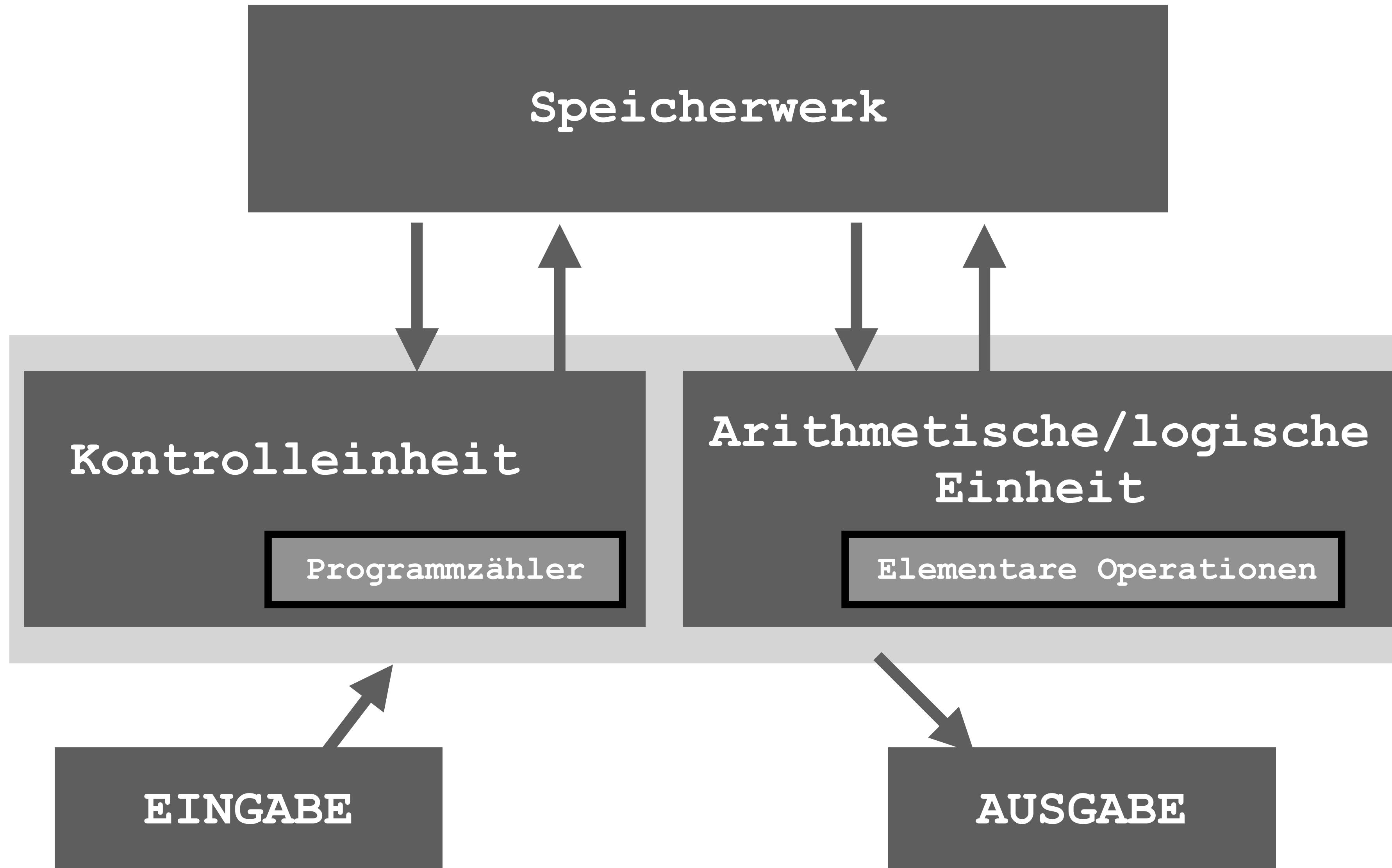
Ein numerisches Beispiel



- Die Quadratwurzel einer Zahl x ist eine Zahl y so, dass $y^2 = x$
- Mögliche Vorgehensweise, um die Quadratwurzel zu berechnen:
 0. Nehmen wir an Variable x speichert den Wert 16
 1. **Rate initialen Wert** einer Variablen g
 2. Falls $g \cdot g$ **nah genug** an x ist, terminiere mit Ergebnis g
 3. Andernfalls sei g **arithm. Mittel** aus g und x/g
 4. **Wiederhole** Schritt 2 mit neuem g (von Schritt 3)

x	g	$g \cdot g$	x/g	$(g+x/g) / 2$
16	3	9	5.33333	4.16667
16	4.16667	17.36114	3.84000	4.00334
16	4.00334	16.02731	3.99667	4.00001

Elementare Rechnerarchitektur



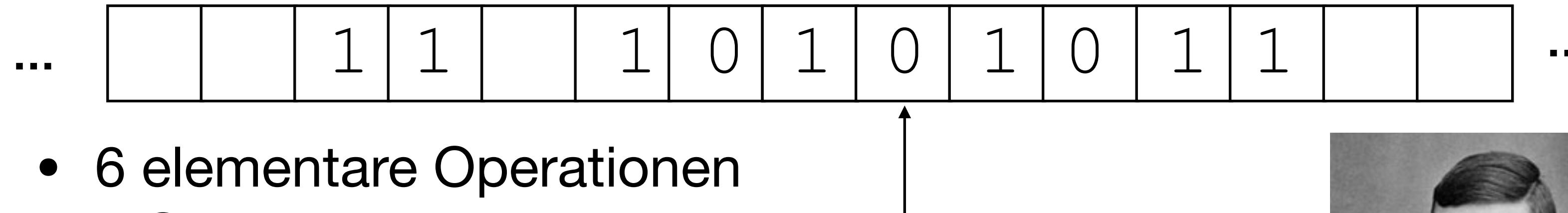
Elementare Rechnerarchitektur

- Eigentliche Sequenz von Instruktionen innerhalb eines Computers
 - ist zusammengesetzt aus einer Menge primitiver Befehle:
 1. Arithmetik und Logik
 2. Einfache Tests
 3. Lese- und Schreibeaktionen
- Ein spezielles Programm (Interpreter) führt jede Anweisung der Reihe nach durch.
 - Verwende Tests, um den Kontrollfluss der Sequenz zu ändern
 - Terminiere das Programm, wenn Berechnung zu Ende ist

Was ist Berechenbarkeit?



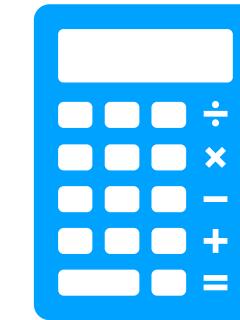
- Alan Turing (1912-1954) zeigte, dass 6 elementare Operationen ausreichen, um **alles Berechenbare** zu beschreiben
 - Speicher=imaginäres unendliches Band:



- 6 elementare Operationen
 - Gehe nach rechts
 - Gehe nach links
 - Drucke gewisses Symbol auf aktuelles Feld
 - Lese Symbol auf aktuellem Feld
 - Lösche Symbol auf aktuellem Feld
 - Halte an
- Moderne Programmiersprachen wie Python haben **mächtigere und praktischere Grundoperationen**
- Alles was in einer Programmiersprache berechenbar ist, ist auch in **jeder anderen** Programmiersprache berechenbar



Über Python



- Es gibt verschiedene Versionen von Python, diese Veranstaltung basiert auf **Python 3.14** (wichtig für Übungskorrektur)
- Installieren Sie am Besten Python 3.14 unter
<https://python.org/downloads>
- Zwei Möglichkeiten der Ausführung von Python-Programmen
 - klassische Variante (relevant für Übungsaufgaben):
 - **Erstellung** der Programme in einem Editor
(wie z.B. vi, emacs, Notepad, joe,...)
 - **Speichern** der Programme mit Datei-Endung *.py
(wie z.B. myprog.py)
 - **Ausführung** der Programme mittels Befehl
`python3.14 myprog.py`
 - Interactive Shell (u.a. relevant für Demonstration in der Vorlesung)

Hauptunterschied:

Ausdrücke werden in der Interactive Shell direkt ausgegeben, während man Sie bei der klassischen Ausführung mittels `print` ausgeben muss.

Operationen und Ausdrücke

- Jede Programmiersprache stellt eine Menge **primitiver Operationen** zur Verfügung, in Python unter anderem:

`== < > + / ** abs()`

- **Ausdrücke** können komplex sein, sind aber immer erlaubte Aneinanderreihungen, wie z.B. folgender Ausdruck

$$a^{**2} + b^{**2} \leq (\text{abs}(a) + \text{abs}(b))^{**2}$$

- Ausdrücke haben eine Semantik, also einen **Wert**.

Der letzte Ausdruck wertet bspw. zu **True** aus
(für alle Integer-Belegungen der Variablen `a` und `b`)

Natürliche Sprache vs. Programmiersprache

primitiver Konstrukte:

- in natürlicher Sprache: Wörter
 - in Programmiersprachen: Zahlen, Zeichenketten (Strings), einfache Operationen wie Vergleiche oder arithmet. Operationen



```
[ ]  
float  
bool ** <=  
return +=  
True if
```

Syntax

- **Syntax** stellt fest welche Zeichenketten überhaupt eine Interpretation zulassen.
 - im Deutschen
 - "Baby Kinder Hund Rahmen"
(syntaktisch ungültig, bspw. da kein Verb vorhanden ist)
 - "Der Bagger bewilligt ständig Häuser."
(syntaktisch gültig)
 - in Programmiersprachen (hier Python)
 - "pi"3.14
(syntaktisch ungültig, Operation fehlt bspw.)
 - 3.14*2+3
(syntaktisch gültig)

Semantik der natürlichen Sprache

(Großes) Problem: Natürliche Sprache kann mehrdeutig sein.

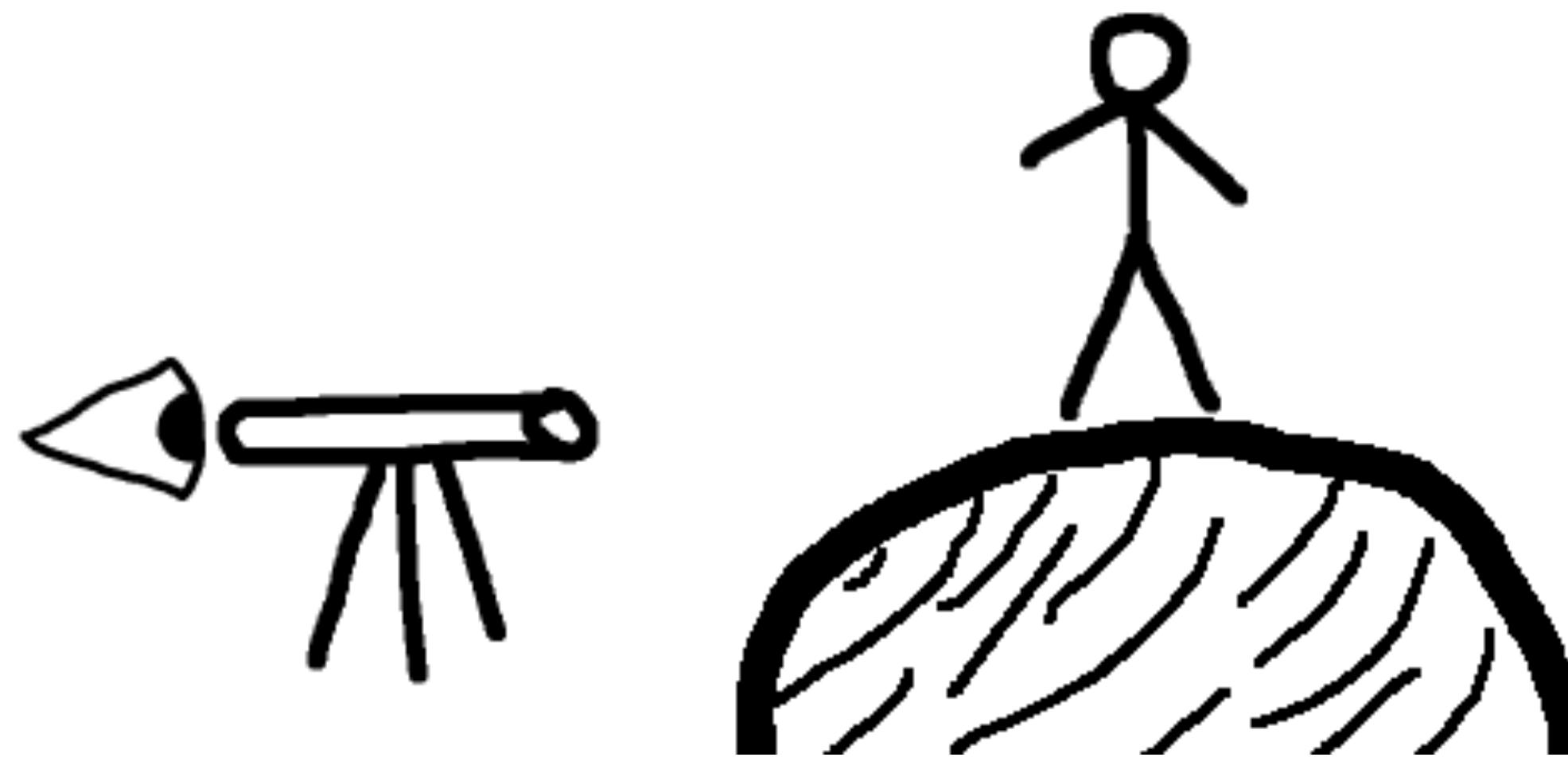
Beispiel: Ich sah den Mann auf dem Berg mit dem Fernrohr.

1. mögliche Interpretation



((Ich sah den Mann) auf dem Berg) mit dem Fernrohr)

2. mögliche Interpretation



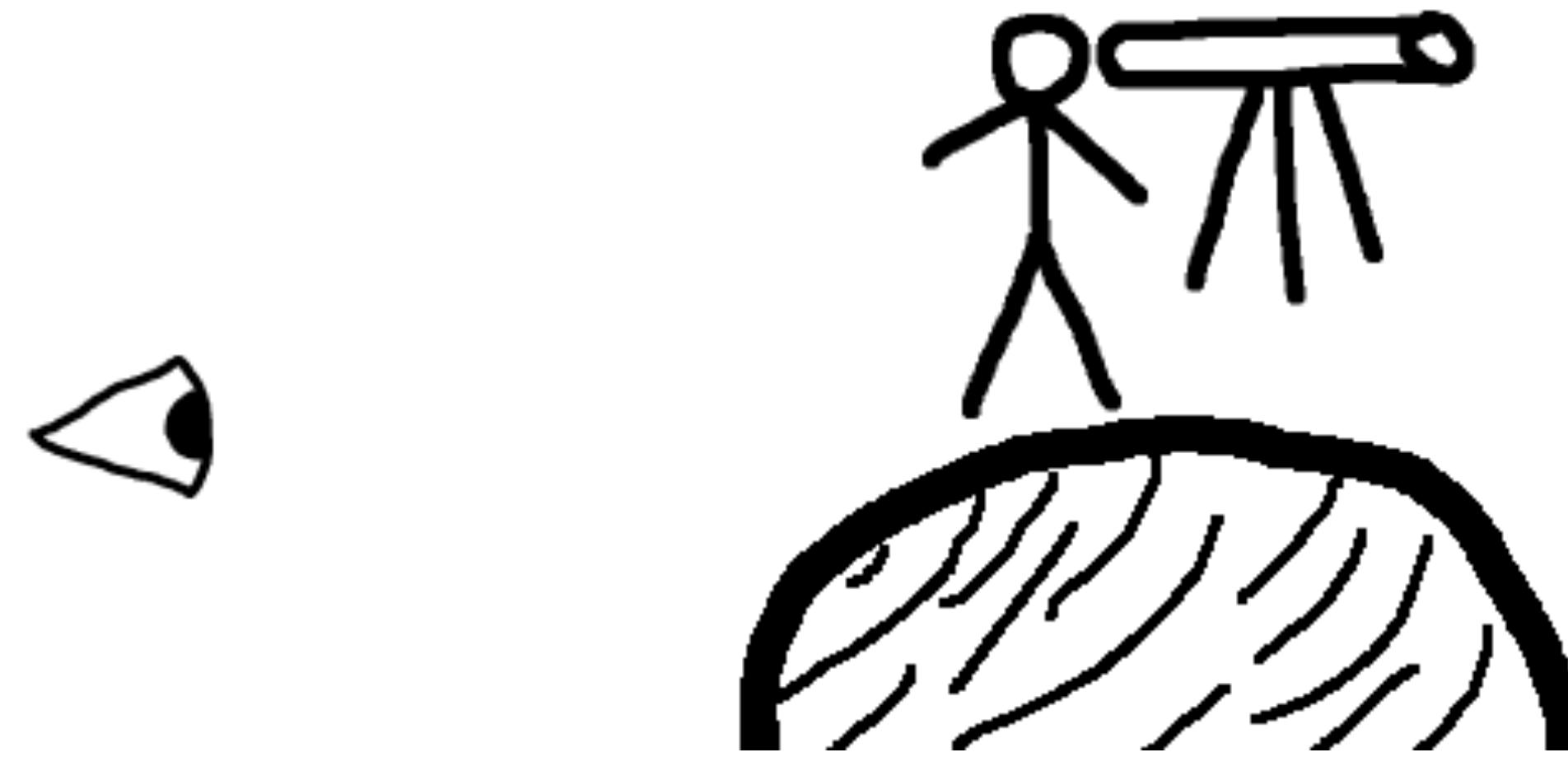
((Ich sah (den Mann auf dem Berg)) mit dem Fernrohr)

3. mögliche Interpretation



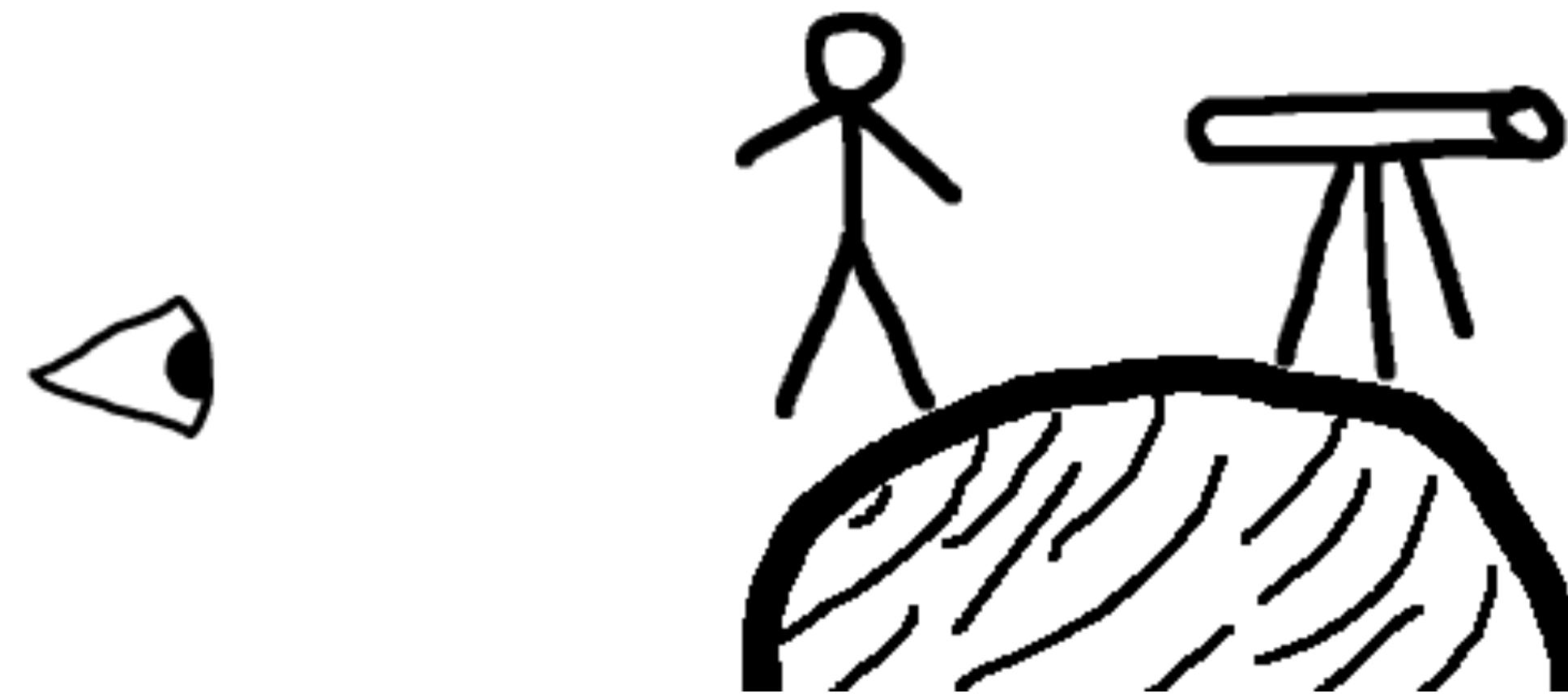
((Ich sah den Mann) (auf dem Berg mit dem Fernrohr))

4. mögliche Interpretation



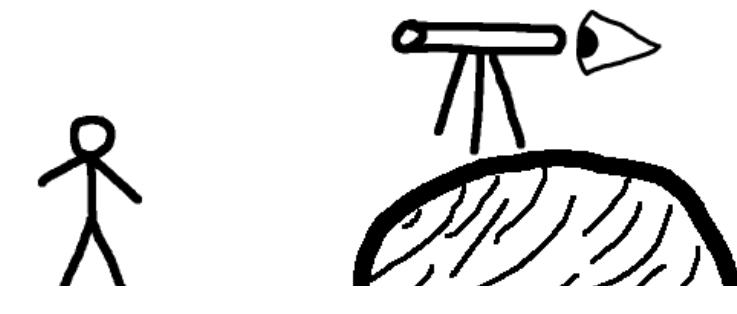
(Ich sah ((den Mann auf dem Berg) mit dem Fernrohr))

5. mögliche Interpretation

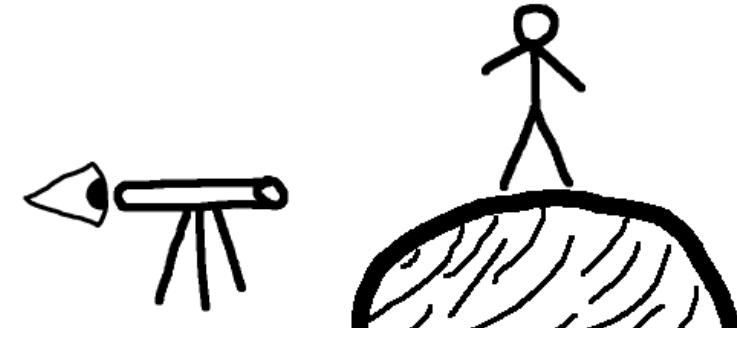


(Ich sah (den Mann (auf dem Berg mit dem Fernrohr)))

5 mögliche Interpretationen



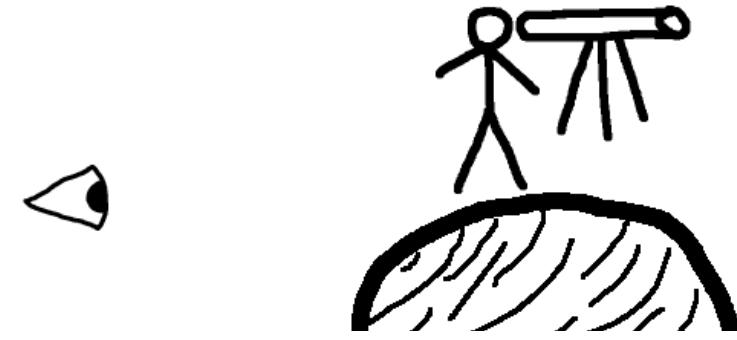
((((Ich sah den Mann) auf dem Berg) mit dem Fernrohr))



((Ich sah (den Mann auf dem Berg)) mit dem Fernrohr))



((Ich sah den Mann) (auf dem Berg mit dem Fernrohr))



(Ich sah ((den Mann auf dem Berg) mit dem Fernrohr))



(Ich sah (den Mann (auf dem Berg mit dem Fernrohr)))

Semantik von Programmiersprachen

Im Gegensatz zu natürlicher Sprache haben alle syntaktisch gültigen Programme (in jeder Programmiersprache) **genau eine Semantik**, z.B.

- $z = (x+y) / 2$
(weist der Variablen z das arithmetische Mittel der Variablen x und y zu)
- **for** y **in** **range** (4) :
 $x=x+x$
(versechzehnfacht den Wert der Variablen x)
- **while** **True** :
print ("Rechne noch")
(ist eine Endlosschleife, die unendlich oft "Rechne noch" ausgibt)
Str.+D um Interactive Shell bzw. Programm zu beenden

Was kann schiefgehen bei Programmen?

- **Syntaktische Fehler** sind sehr häufig und können von einem Compiler (wie in C) bzw. von einem Interpreter (wie in Python) recht einfach und effizient erkannt werden. (Beispiel Shell)
- **Semantische Fehler** sind Verhaltensweisen, die durch die Programmiererin/den Programmier nicht beabsichtigt waren, bspw.:
 - Programmabstürze
 - Endlosschleifen, Endlosrekursionen
 - Programm terminiert mit falscher Ausgabe

Konsequenz von semantischen Fehlern!

Nach Entwicklungskosten von 7 Milliarden USD und einem Wert von rund 500 000 USD wurde die unbemannte Rakete Ariane 5 der European Space Agency am 4.6.1996 von Kourou (Franz. Guiana) gestartet.



Sie explodierte nur 37 Sekunden nach dem Start in einer Höhe von rund 3700m.



Grund des Unglücks:

Eine 64-Bit Floating-Variable, die die horizontale Geschwindigkeit beschrieb wurde in ein 16-Bit Integer umgewandelt. Da die eigentliche Zahl größer als $32767 = 2^{16} - 1$ war, scheiterte die Konvertierung.

Python-Programme

- Ein **Programm** ist eine Sequenz von Definitionen und Befehlen, wobei
 - Definitionen ausgewertet werden und
 - Befehle ausgeführt werden (vom Python-Interpreter)
- **Befehle** befehlen dem Interpretierer etwas zu tun
- Entweder werden diese in
 - **einer Datei eingegeben** und in einer Kommandozeilenumgebung bzw. Shell eingegeben oder
 - in der **Python Interactive Shell** ausgeführt

Python-Objekte

- Programme manipulieren (**Daten**)objekte
- Objekte haben einen **Typ**, der festlegt welche Operationen das Programm auf ihm ausführen darf.
 - Prof. Göller ist ein Mensch, kann lachen, Basketball spielen, Laute von sich geben, reden, ...
 - Sein Mobiltelefon kann sich ausschalten, Laute von sich geben, aufgeladen werden,...
- Objekte sind sind entweder
 - **skalar** (können nicht unterteilt werden)
 - **nichtskalar** (besitzen eine interne Struktur, die es erlaubt sie zu unterteilen)

Skalare Objekte in Python

- `int` - repräsentieren **ganze Zahlen \mathbb{Z}** (Integers), z.B. -17
Exakte Genauigkeit, Bitlänge unbeschränkt
(bei vielen Programmiersprachen nur beschränkt)!
- `float` - repräsentieren **rationale Zahlen \mathbb{Q}** , z.B. 1.33
Bitlänge beschränkt!
- `bool` - repräsentieren die **Booleschen Werte True und False**
- `NoneType` - ist ein spezieller Wert und hat den Wert `None`
- Die Operation `type()` gibt den Typ zurück

```
>>> type(5)  
<class 'int'>
```

```
>>> type(3.0)  
<class 'float'>
```

Das geben Sie in die Python Interactive Shell ein
Das wird angezeigt sobald sie <Enter> drücken.

Typkonvertierung

- Objekte eines Typs können **in anderen Typ konvertiert** werden
- **float(3)** konvertiert den Integer 3 in den Float 3.0
- **int(3.9)** schneidet den Nachkommateil von 3.9 ab und konvertiert in den Integer 3
- **bool(17.8)** wandelt den Float 17.8 in den Booleschen Wert **True** (ganzzahlige Werte ungleich 0 werden nach **True** konvertiert, nur Zahlen gleich 0 wird nach **False** konvertiert)

Ausdrücke

- Objekte können durch Operatoren verknüpft werden
- Jeder Ausdruck hat einen Wert, eine eindeutige Semantik
- Syntax einfacher Operationen:

$\langle \text{Ausdruck} \rangle ::= \langle \text{Objekt} \rangle \text{ } \langle \text{Operator} \rangle \text{ } \langle \text{Objekt} \rangle$

- Syntax allgemeiner Operationen:

$\langle \text{Ausdruck} \rangle ::= \langle \text{Objekt} \rangle$
 | $(\langle \text{Ausdruck} \rangle)$
 | $\langle \text{Ausdruck} \rangle \text{ } \langle \text{Operator} \rangle \text{ } \langle \text{Ausdruck} \rangle$

Operatoren auf int und float

- $i + j$ die **Summe**
 - $i - j$ die **Differenz**
 - $i * j$ das **Produkt**
 - i / j **Division**
 - $i // j$ **Integer-Division**
 - $i \% j$ die **Rest** beim Teilen durch j
 - $i ** j$ **Exponentiation**, d.h. i^j
-
- The diagram consists of five red arrows pointing from the right side of the operator names to their corresponding definitions. The first three arrows point to the same definition: 'Typ int, falls beide Argumente Typ int'. The fourth arrow points to 'Immer vom Float float'. The fifth arrow points to 'Immer vom Float int'.

Operatorenvorrang

- $()$ **Klammern haben immer Vorrang**
bspw. wertet sich $(3+2) * 5$ zu 25 statt 13 aus.
- Der **Operatorenvorrang** ist folgendermaßen (absteigend)
 - $**$
 - $\%$
 - $*$
 - $/$
 - $+$ und $-$

Operatorenvorrang

- Der **Operatorenvorrang** ist folgendermaßen definiert (absteigend)
 - ** (ist **rechtsassoziativ**, d.h. wird von rechts nach links ausgewertet)
 - $*$
 - $/$ (**linksassoziativ**)
 - $+$ und $-$ haben den selben Vorrang (auch **linksassoziativ**)
- Beispiele:
 - $3^{**}3-19/2$ wertet sich zu 17.5 aus
 - $2/2/2$ wertet sich zu 0.5 aus
 - $2^{**}3^{**}4$ ist eine Zahl mit 25 Ziffern
 - $2-2-2-2-2$ wertet sich wie $((((2-2)-2)-2)-2)$ zu -6 aus
 - $(2-(2-(2-(2-2))))$ wertet sich zu 2 aus.

Variablenbelegung und Wertzuweisung

- Das **Gleichheitszeichen belegt** eine **Variable** mit dem **Wert eines Ausdrucks** und ist kein Vergleichsoperator!

The diagram illustrates two assignment statements with annotations:

- `picirca = 3.14159`:
 - A red arrow labeled "Variable" points to the variable name `picirca`.
 - A red arrow labeled "Wert" points to the value `3.14159`.
- `pirund = 22/7`:
 - A green arrow labeled "Ausdruck erlaubt" points to the expression `22/7`.

- Der zugewiesene Wert wird im Hauptspeicher des Computers hinterlegt.
- Die Zuweisung vergibt in Python der Variablen automatisch einen Namen, d.h. der Typ einer Variable muss/kann vor der Zuweisung nicht festgelegt werden
- Wert der Variablen, der gerade ein Wert zugewiesen wurde, darf in späteren Ausdrücken wieder vorkommen!

Ausdrücke abstrahieren

- **Frage:** **Wozu** sollte man Ausdrücken **Namen geben?**
- **Antwort:** Um Sie später **wiederverwenden** zu können!
→ Der Programmcode wird dadurch **wartbarer!**
- **Beispiel.**

```
picirca = 3.14159  
radius = 2.2  
flaeche = picirca * (radius**2)
```

Mathematische Gleichung vs. Programmieren

- Im Gegensatz zu mathematischen Gleichungssystemen, wird bei Programmen nicht einer Belegungen der Variablen gesucht, die gewisse Gleichungen erfüllen.

```
picirca = 3.14  
radius = 2.2  
flaeche = picirca * (radius**2)  
radius = radius+1
```

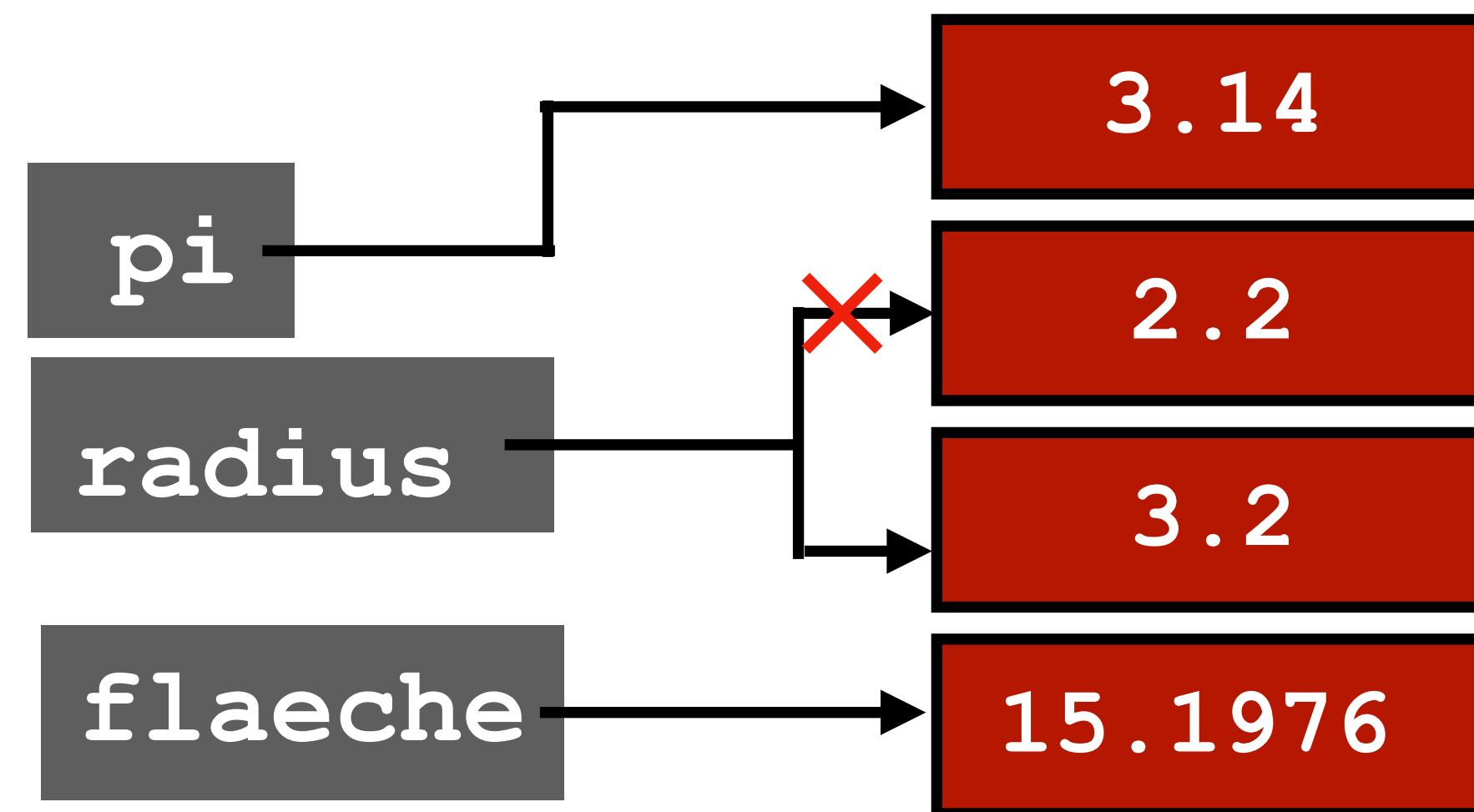
Zuweisung
- Der Wert des Ausdrucks rechts von = wird der Variablen links von = zugewiesen
- Variablename steht links von =
- ist übrigens äquivalent zur Zuweisung radius+=1



Bindungen umändern

- Variablennamen können im Laufe des Programms **neue Werte annehmen**
- Vorheriger Wert wird möglicherweise noch im Hauptspeicher verwendet, jedoch ist der **Zugriff** darauf im Programm **u.U. nicht mehr vorhanden**.
- **Beispiel:** Der Wert der Variablen `flaeche` ändert sich nicht bis Sie Python explizit auffordern die Berechnung neu durchzuführen.

```
pi = 3.14159
radius = 2.2
flaeche = pi * (radius**2)
radius = radius+1
```



Zeichenketten (Strings)

- ... sind Sequenzen von Buchstaben, Sonderzeichen und Ziffern

- können innerhalb zweier Anführungszeichen festgelegt werden

```
moin="Guten Tag"
```

- Können aneinandergehängt werden

```
name="Max"
```

```
gruss=moin + " " + name
```

... und einige andere...

- Objekte vom Typ **bool**, **int** oder **float** können durch die Funktion **str()** in eine Zeichenkette konvertiert werden

```
x="1+1=" + str(1+1)
```

- Die **Länge** einer Zeichenkette kann durch die Funktion **len()** ermittelt werden.

```
len("2**10") + len(str(2**10))
```

1024

Der Ausgabebefehl `print`

- wird verwendet um Dinge in die Konsole **auszugeben**
- Python hat dafür ein **reserviertes Schlüsselwort**, nämlich **print**
- **print** kann **mehrere Argumente unterschiedlichen Typs** entgegennehmen

```
x=1
print(x)
xs=str(x)
print("Tolle Zahl, ",x, " x=",x)
print("Tolle Zahl" + xs + " " + "x=" + xs)
```

Der Eingabebefehl `input (" ")`

- Gibt all das aus, was innerhalb der beiden Anführungszeichen steht
- **liest dann Eingabe des Benutzers ein** und gibt gelesenen String zurück nachdem der Benutzer die Eingabe mit <Enter> beendet hat
- die eingelesene Zeichenkette wird zurückgeben und kann in einer Variablen gespeichert werden.

```
text=input("Geben Sie irgendetwas ein")
print("Danke für " + text)
```

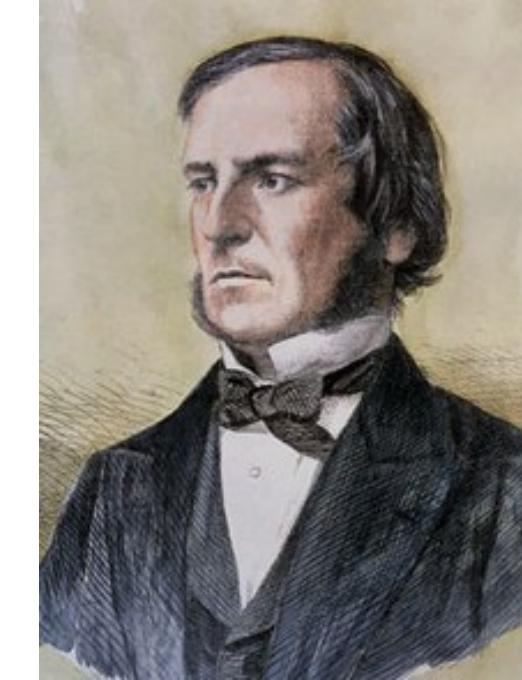
- **input liefert eine Zeichenkette** zurück, d.h. wenn Eingabe als Integer verwertet werden soll, muss sie nach `int` umgewandelt werden.

```
zahl = input("Geben Sie eine Zahl ein")
print("Zwei mal Ihre Zahl ist ", 2*int(zahl))
```

Vergleichsoperatoren auf int, float, string

- Seien i und j im Folgenden Variablenamen
- Vergleiche werten sich immer zu Objekten vom Typ **bool** aus, sind also insbesondere Ausdrücke!
- Typische Vergleiche:
 - $i > j$ (im Gegensatz zur Zuweisung $i = j$, welche kein Vergleich ist)
 - $i >= j$
 - $i < j$
 - $i <= j$
 - $i == j$ **Gleichheit**, d.h. genau dann True, wenn i gleich j ist.
 - $i != j$ **Ungleichheit**, d.h. genau dann False, wenn i gleich j

Boolesche Operatoren



- Seien `a` und `b` im Folgenden Variablennamen vom Typ `bool`
- `not a` → **True** falls Wert von `a` gleich **False**
False falls Wert von `a` gleich **True**
- `a and b` → **True** falls Wert von `a` und von `b` gleich **True**
False andernfalls
- `a or b` → **True** falls Wert von `a` oder von `b` gleich **True**
False andernfalls

Boolesche Wertetabelle				
<code>a</code>	<code>b</code>	<code>a and b</code>	<code>a or b</code>	<code>not a</code>
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Vorrang Boolescher Operatoren

- Der Vorrang der Booleschen Operatoren ist folgendermaßen (absteigend geordnet)
 - **not**
 - **and**
 - **or**
- Insbesondere gilt folgendes:
 - **False and False or True** wertet sich zu **True** aus
 - **not False or True** wertet sich zu **True** aus

Nachbar: „Ich habe gehört Sie sind Mutter geworden, herzlichen Glückwunsch!“
Handelt es um ein Mädchen oder einen Jungen?“

Logikerin: „Ja!“

Boolesche Beispiele

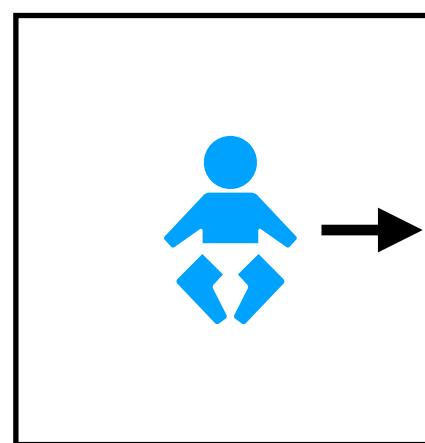
```
a=14<-15 False
b=(39%13)==0 True
c=(not a) or not b True True
print("a oder b ist ",a or b)
print("3**2<3**2 ist ",3**2<3**2)
print(not c or not a) False
                                True
```

True
a=**bool**("0.00")

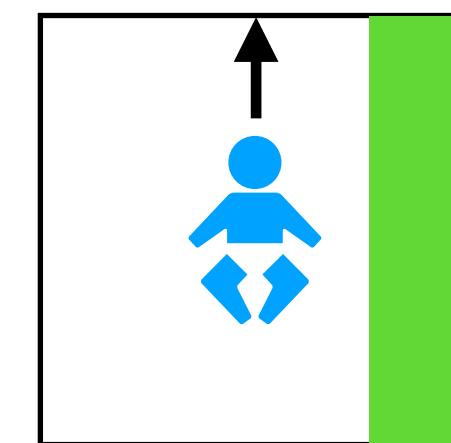
Falls x vom Typ **str** ist, so ist **bool**(x)
genau dann **True**, falls **len**(x)>0

a=**bool**(**int**(**float**("0.00"))) False Falls x vom Typ **int** oder **float** ist, so ist
bool(x) genau dann **True**, falls x ungleich 0
ist.

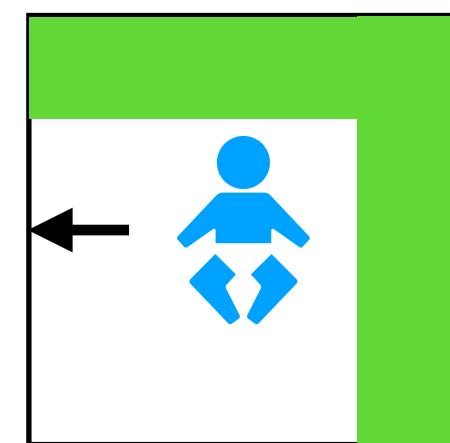
Durch das Labyrinth



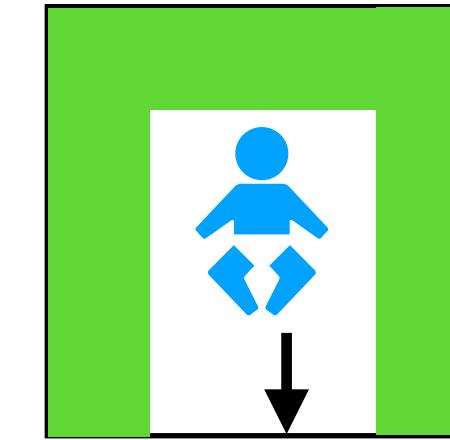
falls rechts frei,
gehe rechts



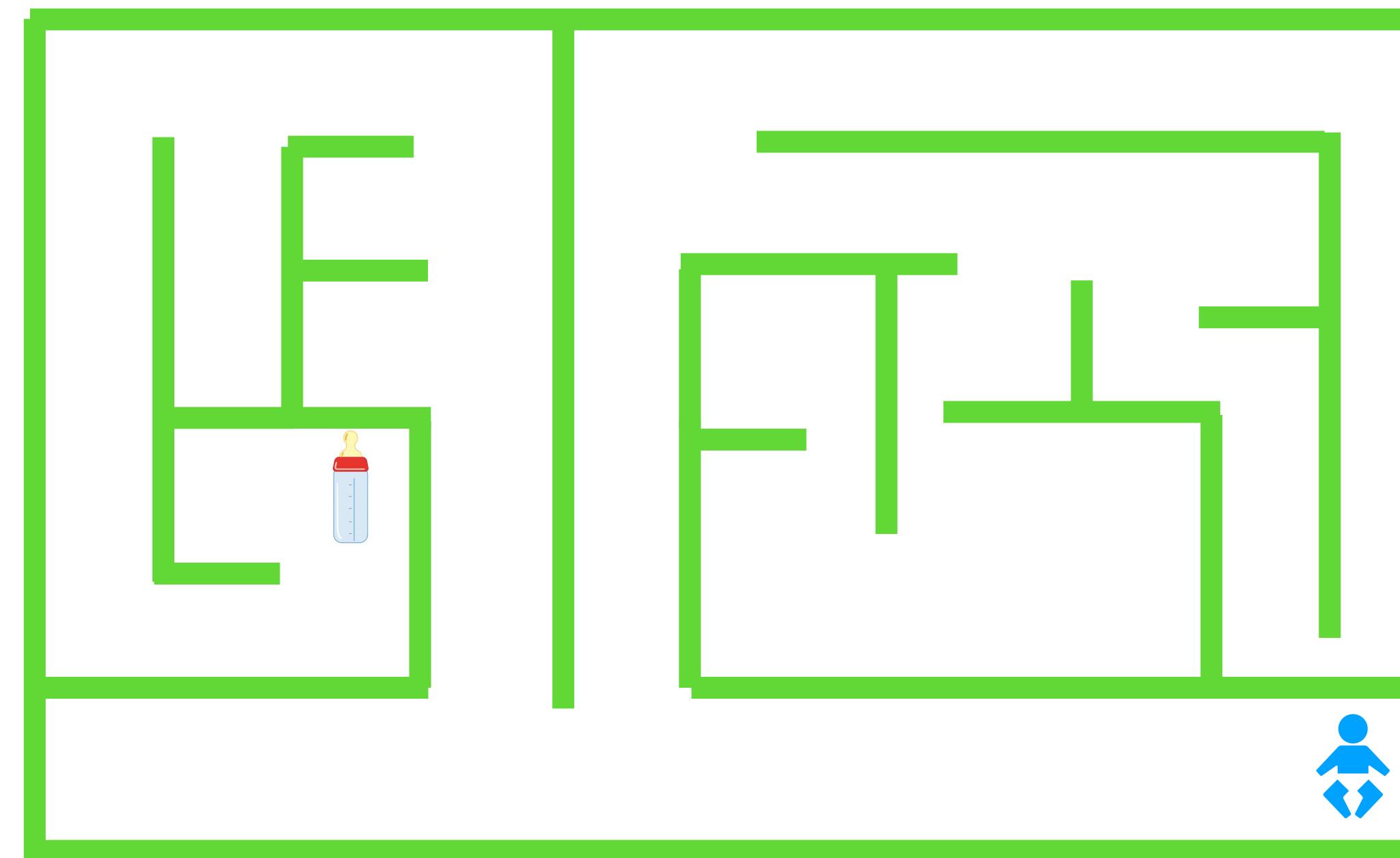
falls rechts blockiert,
gehe geradeaus



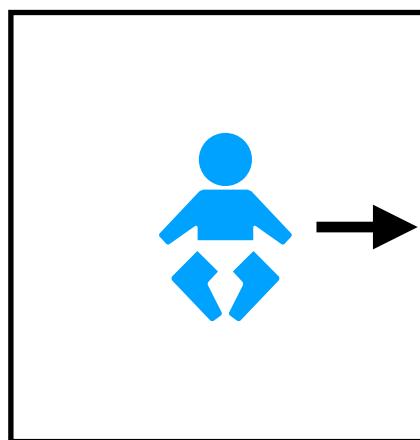
falls rechts und
geradeaus
blockiert, gehe links



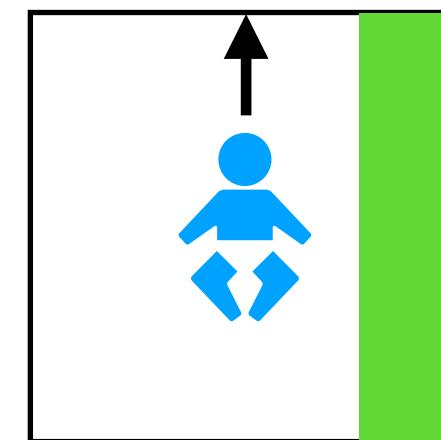
falls rechts,
geradeaus und links
blockiert, gehe
zurück



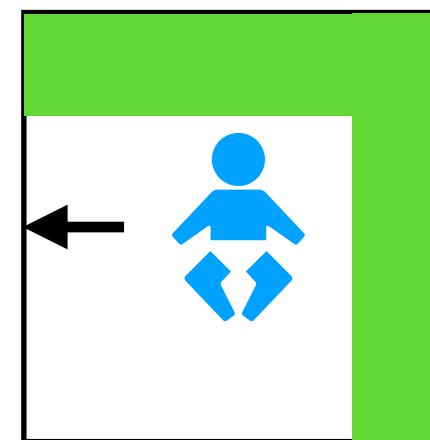
Durch das Labyrinth



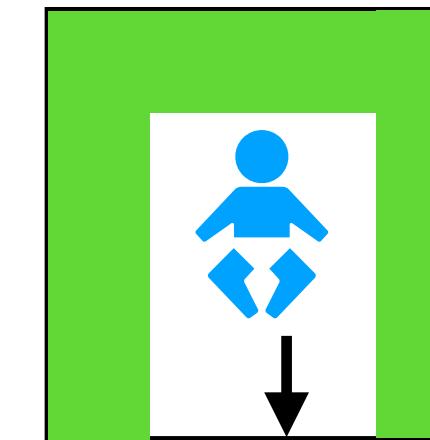
falls rechts frei,
gehe rechts



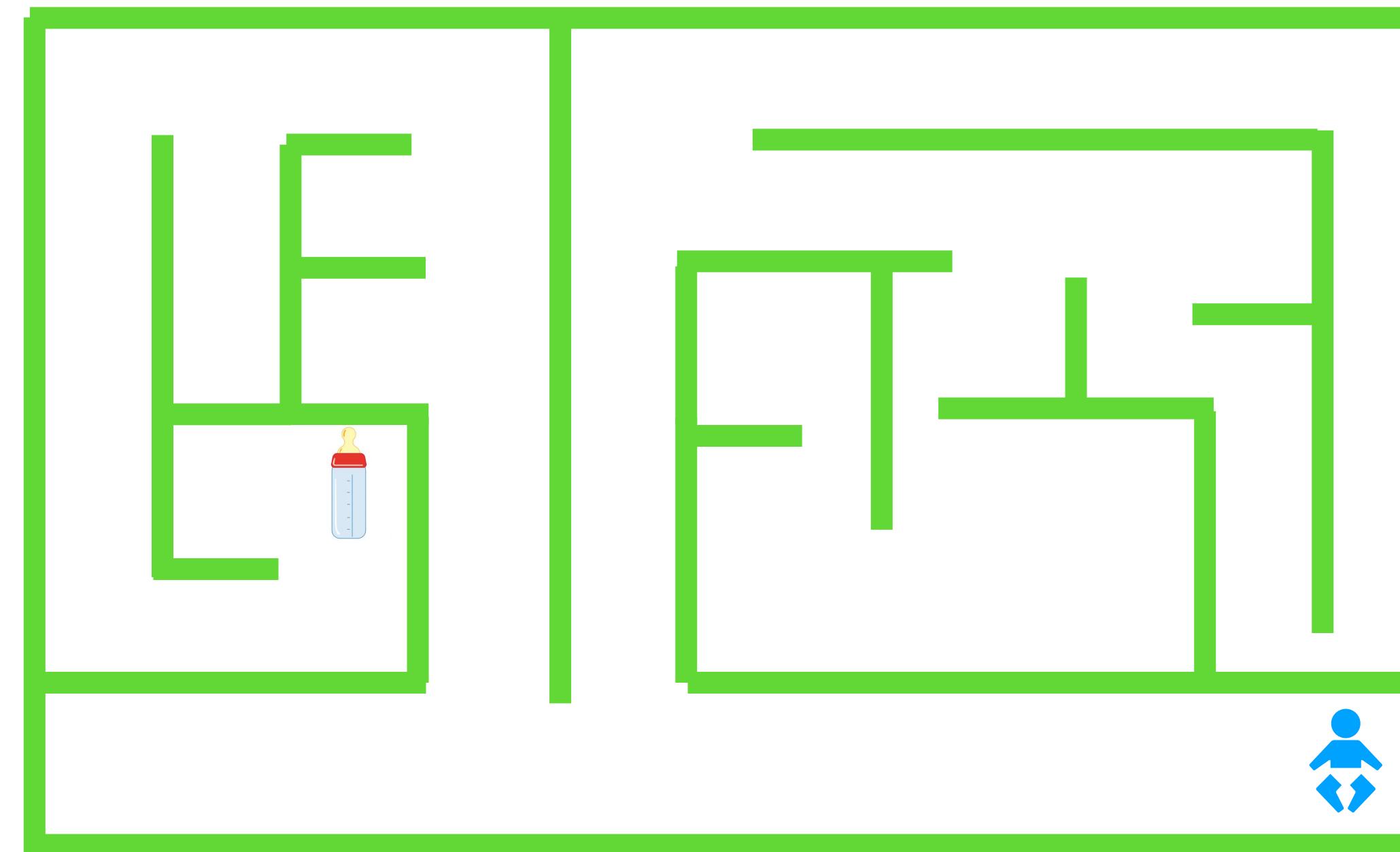
falls rechts blockiert,
gehe geradeaus



falls rechts oder
geradeaus
blockiert, gehe links



falls rechts,
geradeaus und links
blockiert, gehe
zurück



Kontrollfluss Syntax

```
if <Bedingung>:  
    <Programm>
```

```
if <Bedingung>:  
    <Programm1>  
else:  
    <Programm2>
```

```
if <Bedingung1>:  
    <Programm1>  
elif <Bedingung2>:  
    <Programm2>  
else:  
    <Programm3>
```

- <Bedingung> wertet sich entweder zu **True** oder **False** aus
- Führe entsprechendes Programm aus, dessen Bedingung erfüllt wird
- **else** -Programm wird ausgeführt, wenn alle anderen Bedingungen scheitern.
- Beliebig viele **elif**-Zweig möglich, oberste zu **True** ausgewerteter wird zuerst ausgeführt (und nur dieser dann)

Einrückung

- ist in Python notwendig (im Gegensatz zu einigen anderen Programmiersprachen)
- entspricht der Logik des Programmflusses
- Beispiel:

```
x=float(input("Geben Sie eine Zahl x ein: " ))  
y=float(input("Geben Sie eine Zahl y ein: " ))  
if x==y:  
    print("x und y sind gleich")  
    if y!=0:  
        print("daher ist x/y gleich",x/y)  
elif x<y:  
    print("x ist kleiner als y")  
else:  
    print("y ist kleiner als x")
```

= VS. ==

```
x=float(input("Geben Sie eine Zahl x ein: "))
y=float(input("Geben Sie eine Zahl y ein: "))
if x==y:
    print("x und y sind gleich")
    if y!=0:
        print("daher ist x/y gleich",x/y)
elif x<y:
    print("x ist kleiner als y")
else:
    print("y ist kleiner als x")
```

Was würde passieren, wenn wir hier `x=y` schreiben würden?

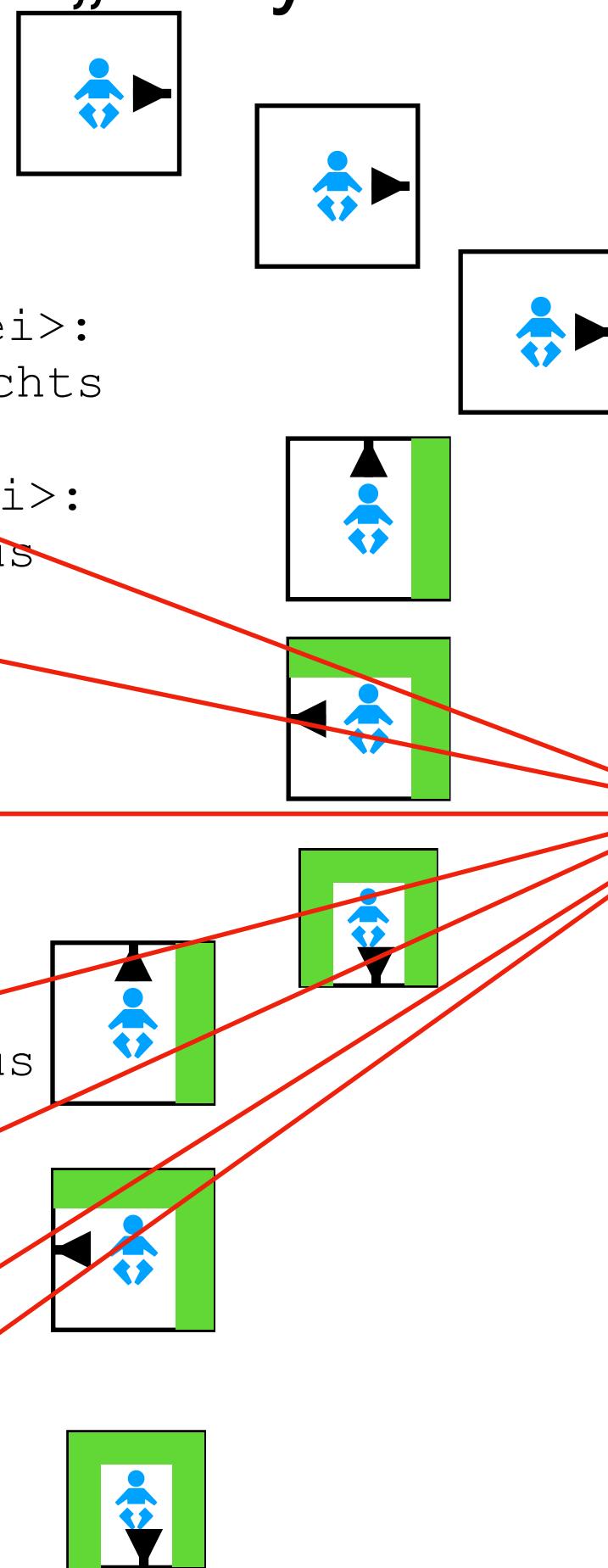
Wir bekämen einen Syntax-Fehler!

Tiefer Kontrollfluss



- Wie könnten wir den obigen „Labyrinth-Algorithmus“ implementieren?

```
if <rechtsfrei>:  
    gehe_rechts  
    if <rechtsfrei>:  
        gehe_rechts  
        if <rechtsfrei>:  
            gehe_rechts  
            ...  
        elif <geradeausfrei>:  
            gehe_geradeaus  
            ...  
        elif <linksfrei>:  
            gehe_links  
            ...  
    else:  
        gehe_zurueck  
        ...  
    elif <geradeausfrei>:  
        gehe_geradeaus  
        ...  
    elif <linksfrei>:  
        gehe_links  
        ...  
    else:  
        gehe_zurueck  
        ...
```



Problem:

- Beliebige Schachtelungstiefe!
- Programmcode Länge hängt von Labyrinthgröße ab!

Lösung:

- Schleifenkonstrukte!

Kontrollfluss: **while**-Schleifen

```
while <Bedingung>:  
    <Programm>
```

- Wie bei den **if**-Bedingungen wertet sich **<Bedingung>** zu einem Booleschen Wert aus
- Solange sich **<Bedingung>** zu **True** auswertet, führe **<Programm>** (u.U. mehrmals) aus.
- Wiederhole bis sich **<Bedingung>** zu **False** auswertet

Beispiel **while**-Schleife

- Folgendes Programm bricht erst dann ab, wenn die Benutzerin bzw. die Benutzerin bzw. der Benutzer "Danke" eingibt

```
text=""  
while text!="Danke":  
    text=input("Wie lautet das Zauberwort ? ")  
print("Endlich haben Sie das Zauberwort eingegeben")
```

Kontrollfluss: **for**-Schleifen

- **for-Schleifen** sind geeignet, um durch eine geordnete Struktur bzw. durch eine Sequenz zu iterieren
- Folgende **while**-Schleife

```
n=0
while n<5:
    print(n)
    n=n+1
```

verhält sich äquivalent zu folgender **for**-Schleife

```
for n in range(5):
    print(n)
```

range(start, ende, schritt)

- Die Standardzuweisungen sind
 - start=0 und schritt=1
 - durchlaufe Schleife bis Schleifenvariable Wert ende-1 annimmt
- **Beispiele:**

```
meinesumme=0  
for i in range(7,10):  
    meinesumme+=i  
print(meinesumme)
```

Probieren Sie es aus!

```
meinesumme=0  
for i in range(5,11,2):  
    meinesumme+=i  
print(meinesumme)
```

Die **break**-Anweisung

- verlässt den aktuellen Schleifendurchlauf
- führt verbleibendes Programm (nach der **break**-Anweisung) im Block **nicht** mehr aus
- verlässt jedoch nur innerste Schleife

```
while <Bedingung1>:  
    while <Bedingung2>:  
        <Programm1>  
        break  
        <Programm2>
```

springt wieder zum Rumpf der
äußeren Schleife zurück

<Programm2> wird nicht mehr ausgeführt!

Die **break**-Anweisung

```
meinesumme=0
for i in range(5,11,2):
    meinesumme+=i
    if meinesumme==5:
        break
    meinesumme+=1
print(meinesumme)
```

Was macht dieses Programm?

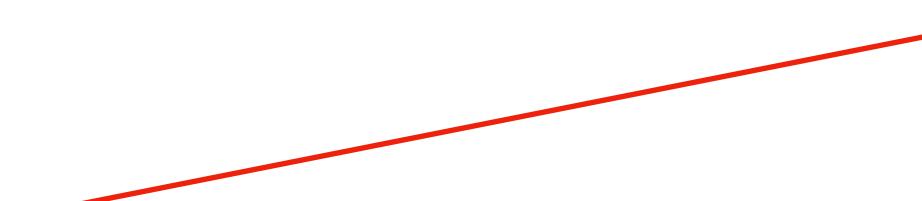
for- vs. while-Schleifen

for-Schleifen

- Anzahl der Schleifendurchläufe **bekannt**
- können **vorzeitig** mittels **break beendet** werden
- benutzen einen **Zähler/Iterator**
- können durch **while**-Schleifen simuliert werden

while-Schleifen

- Anzahl der Schleifendurchläufe im Allgemeinen **unbekannt**
- können **vorzeitig** mittels **break beendet** werden
- benutzen u.U. einen **Zähler bzw. Iterator oder etwas anderes**
- sind im Allgemeinen **nicht** durch **for**-Schleifen **simulierbar**



Mehr hierzu in der Vorlesung Berechenbarkeit und Komplexität
im Rahmen Ihrer Ausbildung in Theoretischer Informatik

Die ASCII Tabelle

- Kurzform für American Standard Code for Information Interchange
- ist eine Kodierung gängiger (aber nicht aller) Buchstaben/Zeichen
- 128 verschiedene Zeichen jeweils durch eine Sequenz von sieben Nullen und Einsen kodiert.
- Beispielsweise ist
 - % an Position 37, also 0100101 in Binärdarstellung
 - B an Position 66, also 1000010 in Binärdarstellung
 - b an Position 98, 1100010 in Binärdarstellung
 - [SPACE] an Position 32, also 0100000 in Binärdarstellung

Die ASCII Tabelle

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Zeichenketten

- sind Sequenzen von Buchstaben, Ziffern und Sonderzeichen bei denen es auf die Groß- und Kleinschreibung ankommt
- können mittels `==`, `<` und `>` verglichen werden:
 - Für zwei Zeichenketten u und v gilt $u < v$, falls Buchstaben x und y und ein Wort w (möglicherweise leer) existiert so, dass
 - u fängt mit wx an
 - v fängt mit wy an
 - x ist **vor** y in der ASCII-Tabelle

Wie vergleichen sich diese Zeichenketten?

`w="aaa"`
`x="aa"`
`y="b"`
`z=x+"a"`

Zugriff auf Zeichenketten

- durch die Verwendung eckiger Klammern kann auf einzelne Positionen in Zeichenketten zugegriffen werden
- Beispiel:

`s = "abc"`

Indizes	0	1	2	← Indizierung beginnt stets bei 0
	-3	-2	-1	← Letztes Element bei -1

<code>s [0]</code>	→	wertet sich zu "a" aus
<code>s [1]</code>	→	wertet sich zu "b" aus
<code>s [2]</code>	→	wertet sich zu "c" aus
<code>s [3]</code>	→	liefert einen „Index-Error“ da Index zu groß
<code>s [-1]</code>	→	wertet sich zu "c" aus
<code>s [-2]</code>	→	wertet sich zu "b" aus
<code>s [-3]</code>	→	wertet sich zu "a" aus

Herausschneiden (Slicing) auf Zeichenketten

- Spezielle Unterwörter können mittels [start:stop:schritt] herausgeschnitten werden
- es reicht auch nur die ersten beiden anzugeben (getrennt durch :)
- Beliebige Parameter können auch weggelassen werden:
 - start=0 ist die Standardbelegung
 - stop ist standardmäßig Länge des Worts
 - schritt=1 ist die Standardbelegung

s="abcdefg"

s[3:6]

wertet sich zu "def" aus, wie s[3:6:1]

s[3:6:2]

wertet sich zu "df" aus

s[:]

wertet sich zu "abcdefg" aus

s[::-1]

wertet sich zu "hgfedcba"

Probieren Sie es im
Zweifelsfall selbst aus

Zeichenketten

- sind **unveränderlich** (immutable), d.h. sie können nicht modifiziert werden

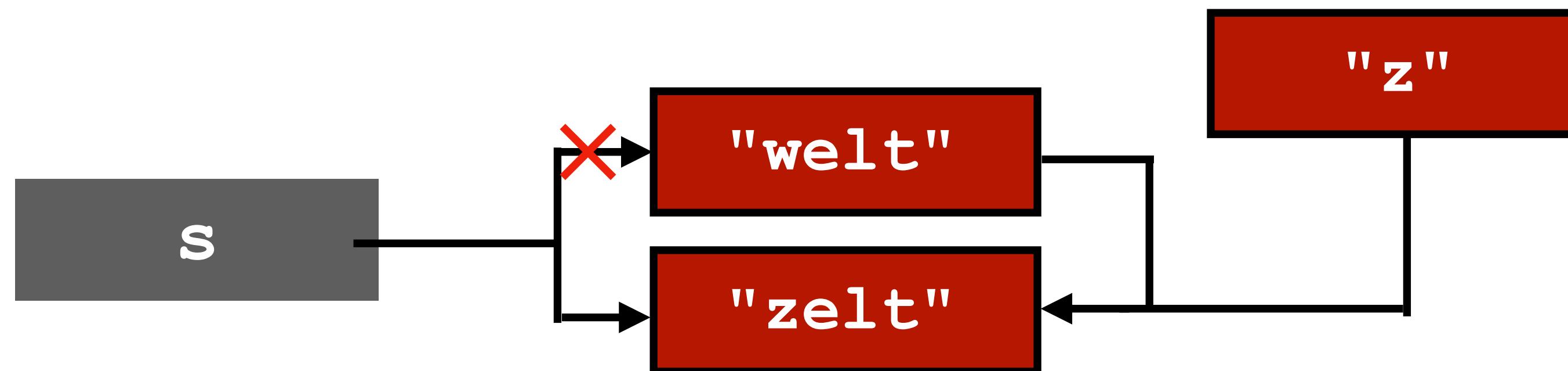
`s="welt"`

`s[0]="z"`

→ liefert einen Type-Error

`s="z"+s[1:]`

→ ist erlaubt, `s` wird einem neuen Objekt zugewiesen



Wiederholung **for**-Schleifen

- Jede **for**-Schleifen hat eine Laufvariable, die über eine Sequenz (von Werten) iteriert

```
for var in range(4): →var iteriert über die Werte 0, 1, 2, 3  
<Programm>
```

```
for var in range(4, 6): →var iteriert über die Werte 4, 5  
<Programm>
```

- **range** liefert eine Möglichkeit über Zahlen zu iterieren, aber eine Laufvariable einer **for**-Schleife kann über jede Sequenz von Werten iterieren, nicht nur Sequenzen von Zahlen!

Zeichenketten und for-Schleifen

- Die folgenden beiden Programme sind im Wesentlichen äquivalent
- Das zweite ist jedoch eleganterer Python-Code

```
s="abcdefghijklmnopqrstuvwxyz"  
for index in range(len(s)):  
    if s[index]== "i" or s[index]== "u":  
        print("Es gibt ein i oder ein u")  
  
for buchstabe in s:  
    if buchstabe=="i" or buchstabe=="u":  
        print("Es gibt ein i oder ein u")
```

Beispiel eines Cheerleader-Roboters

auf englisch

```
an_letters="aefhilmnorsxAEFHILMNORSX"  
  
word=input("I will cheer for you! Enter a word: ")  
times=int(input("Enthusiasm level (1-10) : "))  
  
i=0  
while i<len(word):  
    char=word[i]  
  
    if char in an_letters:  
        print("Give me an " + char + "! " + char)  
else:  
        print("Give me a " + char + "! " + char)  
  
    i+=1  
  
print("What does that spell?")  
for i in range(times):  
    print(word, "!!!!")
```

Übung

- Was macht folgendes Programm?

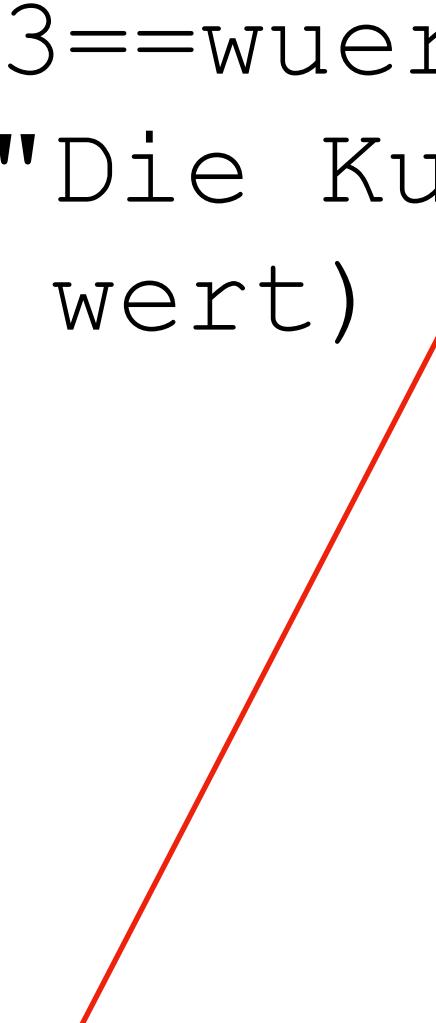
```
s1="abracadabra"  
s2="rhabarberbarbara"  
for char1 in s1:  
    for char2 in s2:  
        if char1==char2:  
            print("gemeinsamer Buchstabe")  
            break
```

Rate und verifiziere

- Der folgende Ansatz wird oft auch „Exhaustive Enumeration“, erschöpfende Aufzählung genannt.
- Gegeben sei irgendein algorithmisches Problem
- Sie raten zu Beginn eine mögliche Lösung
- Sie können überprüfen ob ein gegebener Wert tatsächlich eine Lösung des Problems beschreibt
- Sie raten weiter bis Sie eine Lösung gefunden haben oder Sie alle Möglichkeiten durchprobiert haben

Berechnung der Kubikwurzel

```
wuerfel=8
for wert in range(wuerfel+1):
    if wert**3==wuerfel:
        print("Die Kubikwurzel von ",wuerfel,"ist ",\
              wert)
```



Zur Erinnerung: **range**(wuerfel+1)=[0, 1, ..., wuerfel]

Berechnung der Kubikwurzel

```
wuerfel=8
for wert in range(abs(wuerfel)+1):
    if wert**3>=abs(wuerfel):
        break
    if wert**3!=abs(wuerfel):
        print(wert, "ist keine Kubikzahl mit "+
              "ganzer Wurzel")
else:
    if wuerfel<0:
        wert=-wert
    print("Kubikwurzel von " + str(wuerfel) + \
          " ist " + str(wert))
```

Absolutwert

Approximative Lösungen

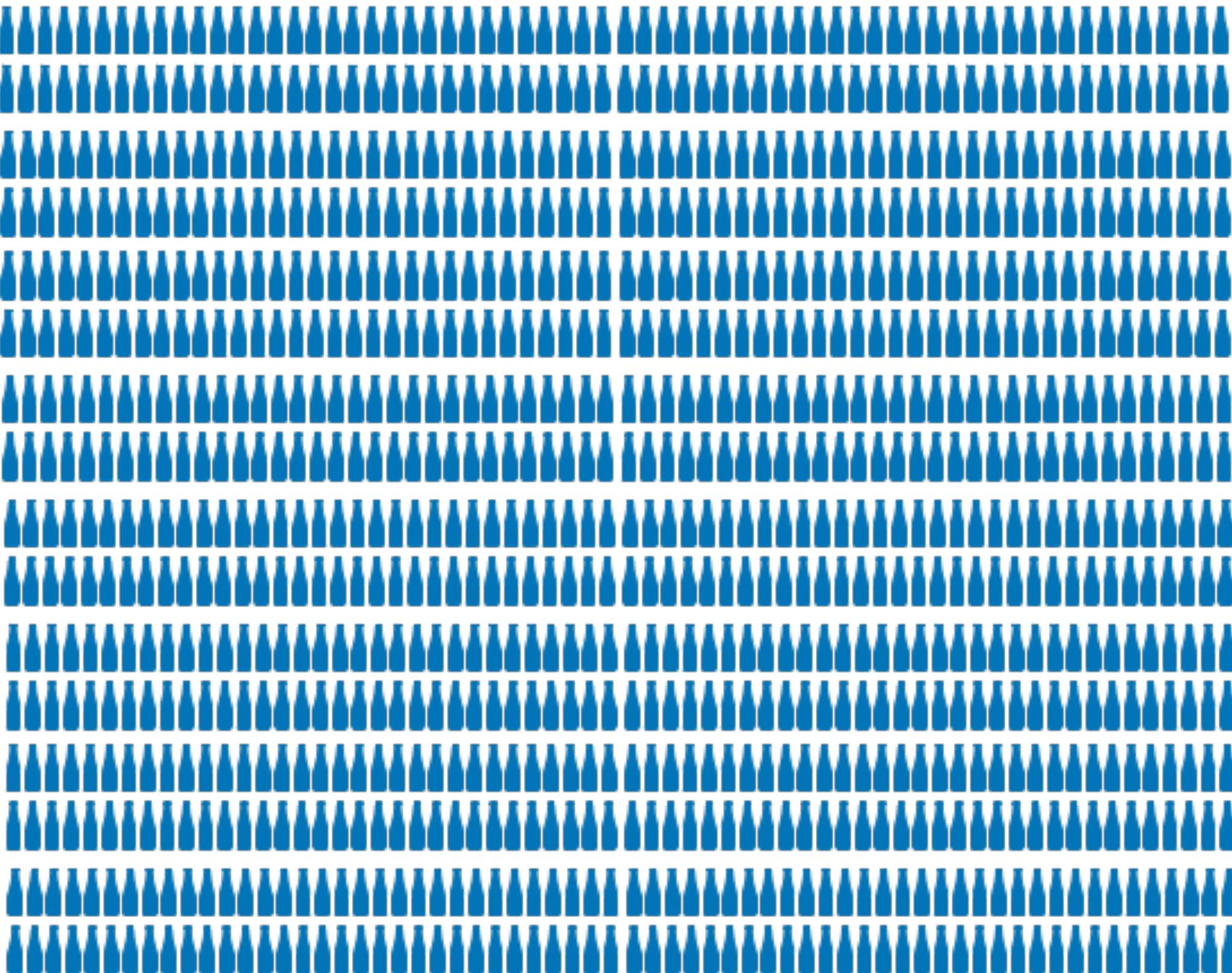
- können als Lösungen aufgefasst werden, die zwar nicht unbedingt exakt aber „gut genug“ sind
- Rate initialen Wert und inkrementiere mit einem kleinen Wert
- Rate weiter falls $|wert^3 - wuerfel| \geq \epsilon$ für einen kleinen Wert ϵ
- **Abwägung notwendig:**
 - je kleiner ϵ gewählt wird, desto langsamer das Programm
 - je größer ϵ gewählt wird, desto ungenauer das Ergebnis

Approximativer Berechnen der Kubikwurzel

```
wuerfel=27
epsilon=0.01
wert=0
inkrement=0.0001
rate_anzahl=0
while abs(wert**3-wuerfel)>=epsilon and wert<abs(wuerfel):
    wert+=inkrement
    rate_anzahl+=1
print("Wir haben ",rate_anzahl," mal geraten")
if abs(wert**3-wuerfel)>= epsilon:
    print("Leider konnte die Kubikwurzel von " +\
          wuerfel, " nicht berechnet werden ")
else:
    print(wert , " ist nahe an Kubikwurzel von ", \
          wuerfel)
```

Ein Flaschenrätsel

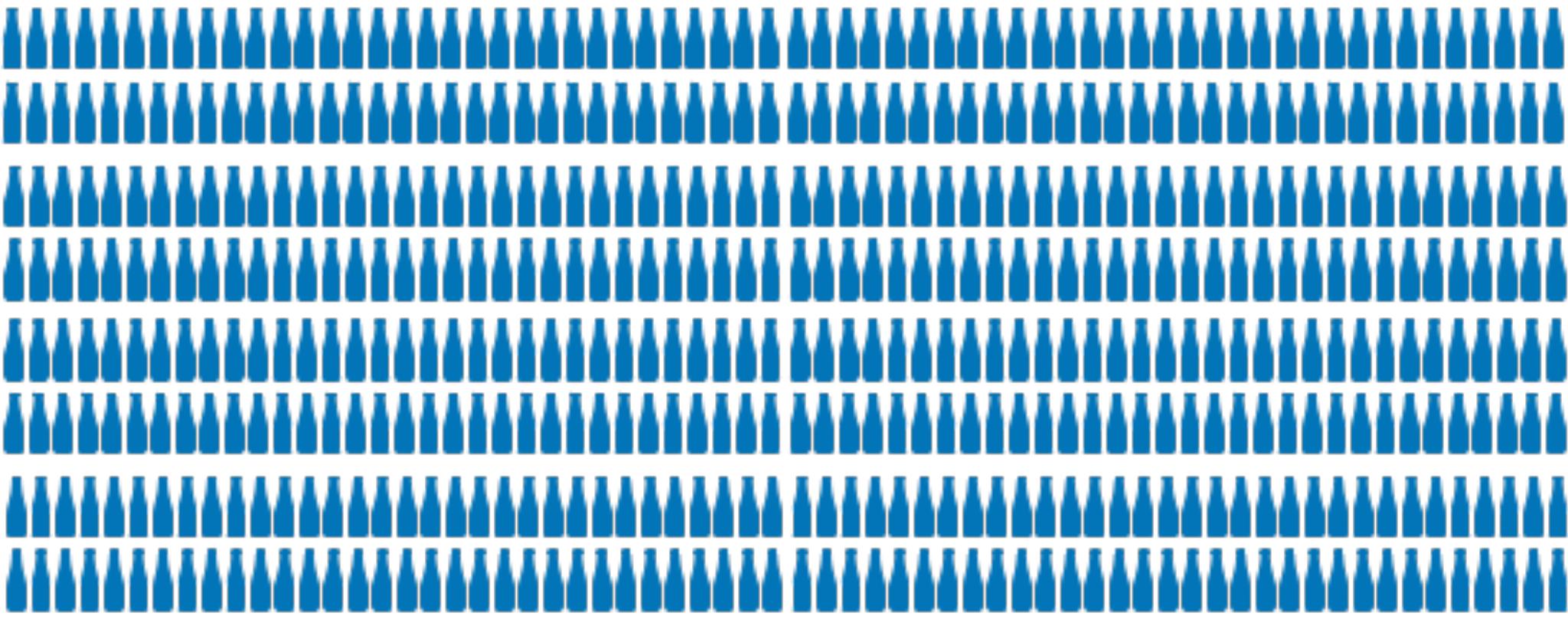
- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



schluckzähler 0

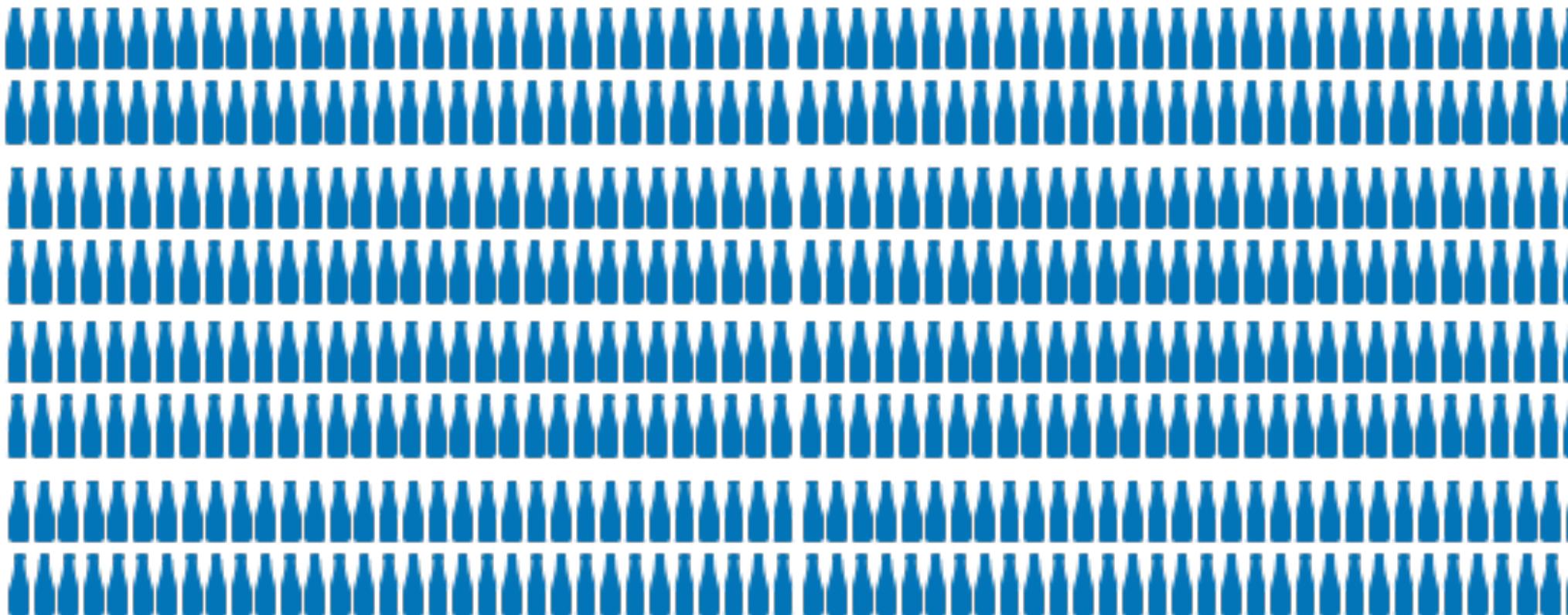
Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



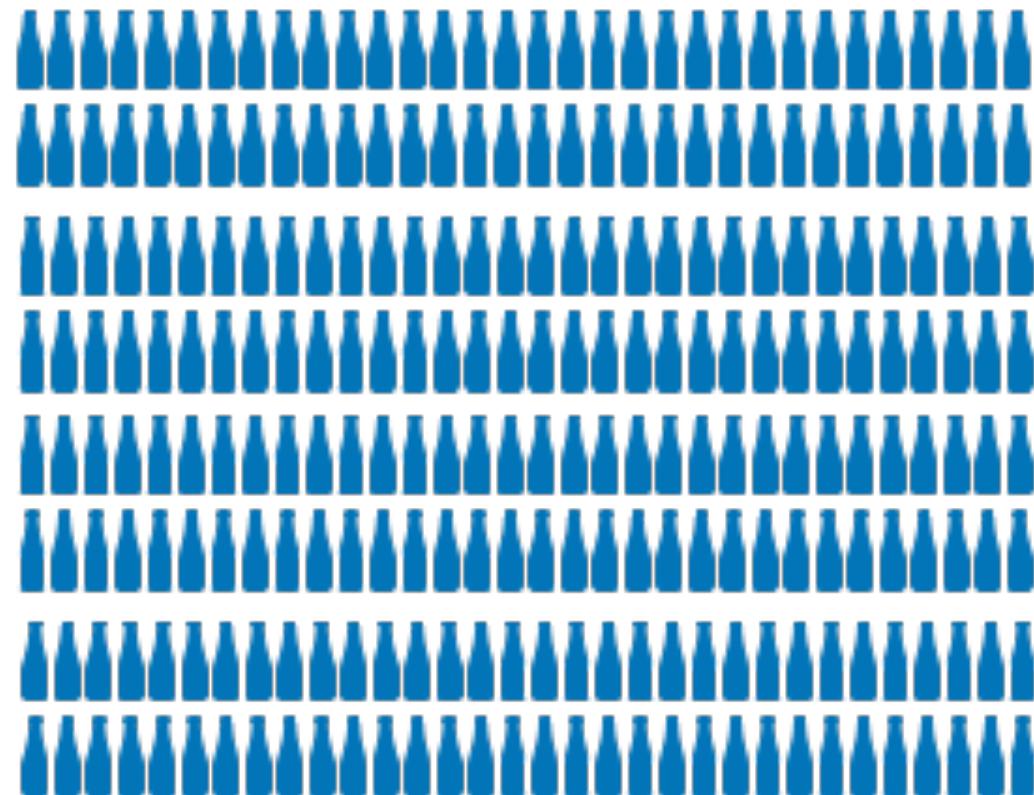
Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



Ein Flaschenrätsel

- Sie haben 1000 Flaschen von denen **genau eine vergiftet** ist
- Die eine vergiftete Flasche schmeckt bitter
- Nehmen wir an Sie hätten das entsprechende Gegengift griffparat
- Wieviele Schlücke sind notwendig, um die vergiftete Flasche zu finden?



schluckzähler 10

10 Schlücke reichen aus (sogar bei 1024 Flaschen)

$$\log_2(1000) = 10$$

Binäre Suche

- Gegeben sei eine Sequenz geordneter Werte (z.B. eine aufsteigend sortierte Sequenz natürlicher Zahlen)
- Berechnungsproblem:
 - **Gegeben:** Geordnete Sequenz s und Wert x
 - **Ausgabe:** Bestimme ob x in s enthalten und falls ja bestimme Position von x in s
- Idee:
 - Fasse s als Intervall auf
 - Zu vergleichender Wert: Mittleres Element m von s
 - Vergleiche m mit x , falls gleich sind wir fertig
 - Falls $m < x$, fahre mit halbem Intervall links von m fort
 - Falls $m > x$, fahre mit halbem Intervall links von m fort

Kubikwurzelberechnung beschleunigt

```
wuerfel=27
epsilon=0.01
links=0
rechts=wuerfel
rate_anzahl=0
wert=(links+rechts)/2
while abs(wert**3-wuerfel)>=epsilon:
    if wert**3<wuerfel:
        links=wert
    else:
        rechts=wert
    wert=(links+rechts)/2.0
    rate_anzahl+=1
print("Wir haben ",rate_anzahl," mal geraten")
print(wert, " ist nah an der Kubikwurzel" +\
        " von ", wuerfel)
```

Aufwand Binäre Suche

- Suchraum, wenn Sequenz zu Beginn Länge n hat
 - Nach erstem Vergleich: $n/2$
 - Nach zweitem Vergleich: $n/4$
 - Nach k -tem Vergleich: $n/2^k$
- Das Verfahren terminiert nach $\log_2 n$ Schritten
- Binäre Suche funktioniert nur auf geordneten Sequenzen
- Der vorherige Code funktioniert nicht auf allen Eingaben.
Warum?

Wie schreiben wir Code?

- Bisher...
 - haben wir kleine Berechnungen durchgeführt
 - haben wir für jede Berechnung ein Programm in eine Datei geschrieben
 - bestand jede Datei aus einem Programm, das irgendetwas berechnet bzw. erledigt
 - war der Code eine Sequenz von Instruktionen
- Probleme mit diesem Ansatz
 - schwieriger zu realisieren für umfangreiche Probleme
 - bei sich wiederholenden Berechnungen können sich viele Fehler einschleichen (wenn man Code bspw. dupliziert)

Gute Programmierpraktiken

- **Weniger kann durchaus mehr sein:** Ein längeres Programm ist nicht unbedingt ein besseres Programm
 - schwieriger zu verstehen bzw. zu warten
 - es können mehr Fehler enthalten sein
 - Laufzeit erhöht sich unter Umständen
- **Qualität** eines Programms ist eher seine **Funktionalität** (was wird eigentlich berechnet?)
- Wir führen daher **Funktionen** ein
- Funktionen realisiert das Prinzip der **Modularisierung** und **Abstraktion**

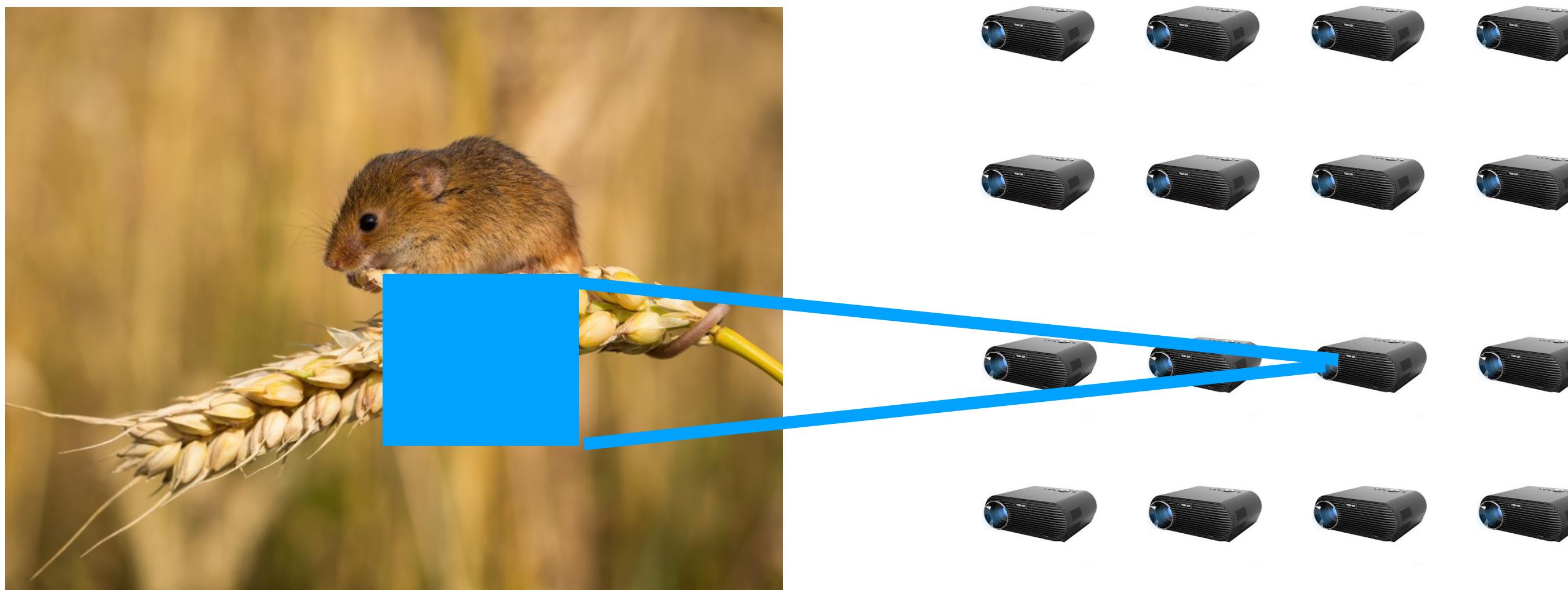
Beispiel: Ein Projektor

- Ein Projektor ist eine Black-Box, d.h.
 - wir wissen nicht wie ein Projektor funktioniert
 - wir kennen nur seine Schnittstelle: also welche Ausgabe bei welcher Eingabe zu erwarten ist
- Nehmen wir an wir könnten diverse elektronischen Geräte anschließen an einen Projektor anschließen, die mit ihm kommunizieren können
- Ein Projektor ist eine Black-Box, die ein gegebenes Bild von der Eingabekquelle an die Wand vergrößernd darstellt
- **Abstraktionsidee:** Wir müssen nicht wissen wie ein Projektor funktioniert, um ihn zu nutzen (u.U. für andere/größere Zwecke)



Beispiel: Ein Projektor

- Nehmen wir an wir wollen ein großes Bild darstellen, das aus vielen kleinen Bildern besteht
- jeder einzelne Projektor projiziert nur einen Teil des Eingabe-Bilds
- Alle Projektoren arbeiten zusammen, um ein großes Bild darzustellen



- **Idee der Dekomposition:** Zerlege die eigentliche Aufgabe bzw. das eigentliche Problem in verschiedene Teile

Struktur durch Dekomposition

- Im obigen Beispiel bestand die Dekomposition im Zusammenspiel verschiedener Projektoren
- In der Programmierung entspricht die Dekomposition darin den Code in einzelne **Module** zu zerlegen...
 - die in sich **geschlossen** sind
 - benutzt werden, um Code **zusammenzuführen**
 - mit der Absicht **wiederverwendbar** zu sein
 - um den Code zu **strukturieren**
 - den Code zu **vereinheitlichen**
- In dieser Vorlesung wird dies zunächst durch **Funktionen** realisiert
- Später wird dies durch **Klassen** im Rahmen der Objektorientierung realisiert



Detailunterdrückung durch Abstraktion

- Im obigen Projektoren-Beispiel reichte es völlig aus zu wissen wie man Projektoren benutzt, unwichtig war es zu wissen wie sie genau funktionieren
- Beim Programmieren ist das ähnlich, wir stellen uns Codestücke auch als eine Black-Box vor, d.h.
 - wir haben kein Wissen darüber wie sie funktioniert
 - wir benötigen dieses Wissen auch nicht
 - wir wollen diese Details auch nicht sehen, um uns Mühe zu ersparen
- Wir erreichen diese Abstraktion durch **Funktionsspezifizierungen** oder **Docstrings**

Funktionen

- Wir schreiben Codestücke, die sich **Funktionen** nennen
- Funktionen werden nicht ausgeführt ausser sie werden explizit aufgerufen
- Komponenten von Funktionen:
 - **Name der Funktion**
 - **Parameter**
 - **Docstring** (optional, wird jedoch empfohlen)
 - **Rumpf/Body**
 - **Rückgabewert**

Funktionen, genauer

```
def ist_gerade( i ):  
    """  
        Eingabe: i, ein positiver Integer  
        Gibt True zurueck, falls i gerade ist,  
        ansonsten False  
    """  
    print("Gerade sind wir in ist_gerade")  
    return i%2==0
```

Schlüsselwort
Name
Parameter bzw.
Argumente, hier ein Parameter
Rumpf
Spezifikation,
docstring

```
ist_gerade(3)
```

Später können Sie diese Funktion
durch ihren Namen und Werte für
die Parameter übergeben

Im Rumpf der Funktion

```
def ist_gerade( i ):  
    """  
        if Eingabe: i, ein positiver Integer  
        Gibt True zurueck, falls i gerade ist,  
        ansonsten False  
    """  
print("Gerade sind wir in ist_gerade")  
return i%2==0
```

Schlüsselwort

Auswertung dieses Ausdrucks
ist die Rückgabe dieser Funktion,
hier ein Objekt vom Typ **bool**

Ausführung des
print-Befehls

Im Rumpf der Funktion

- **Deklарierter Parameter** (formal parameter) wird dem Wert des **konkreten Parameters** (actual parameter) beim Funktionsausfruf zugewiesen
- Der **Geltungsbereich** ist die Objektzuweisung der Namen
- Neuer Geltungsbereich wird **beim Betreten der Funktion** kreiert

```
def f( x ):  
    x=x+1  
    print("innerhalb von f(x) : x=", x)  
    return x  
  
x=3  
z=f( x )
```

Dekлierter Parameter

Funktionsdefinition

Konkreter Parameter

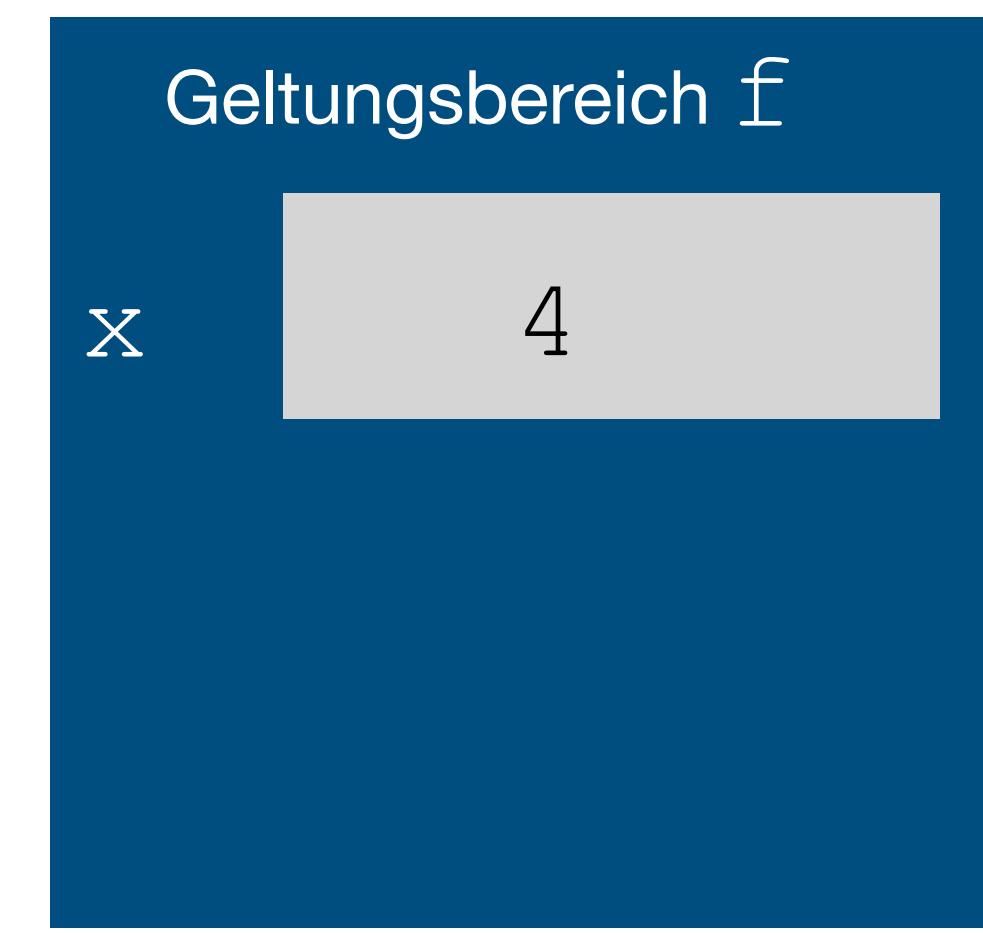
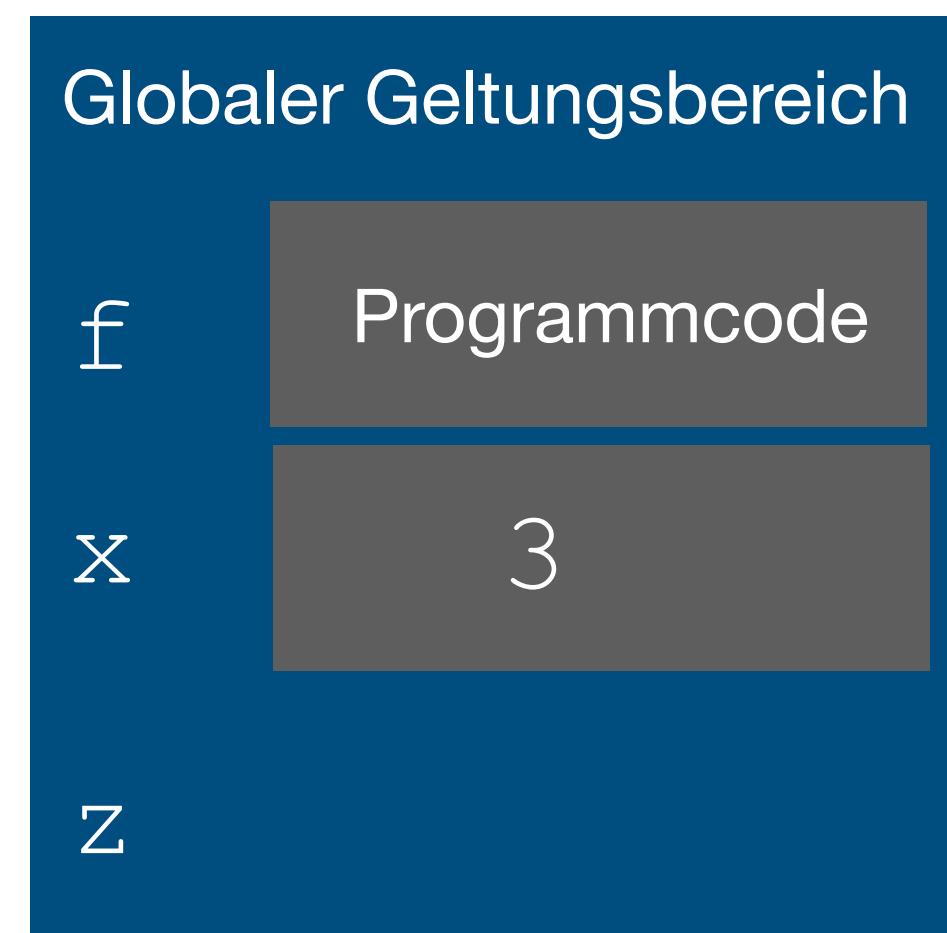
Code des Hauptprogramms

- Initialisiert Variable x
- macht einen Funktionsaufruf $f(x)$
- weist Variablen z den Rückgabewert des Funktionsaufrufs zu

Geltungsbereich

```
def f( x ):  
    x=x+1  
    print("innerhalb von f(x) : x=",x)  
    return x
```

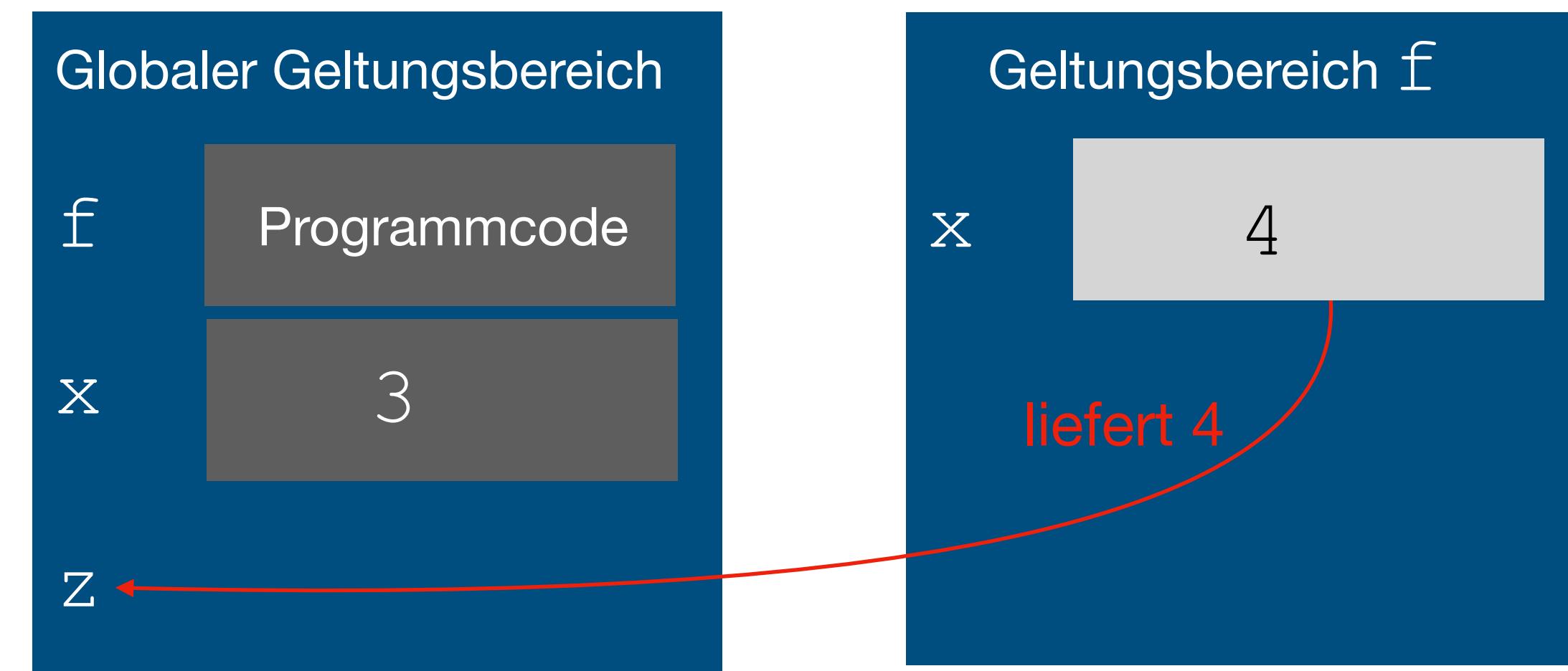
```
x=3  
z=f( x )
```



Geltungsbereich

```
def f( x ):  
    x=x+1  
print("innerhalb von f(x) : x=", x)  
return x
```

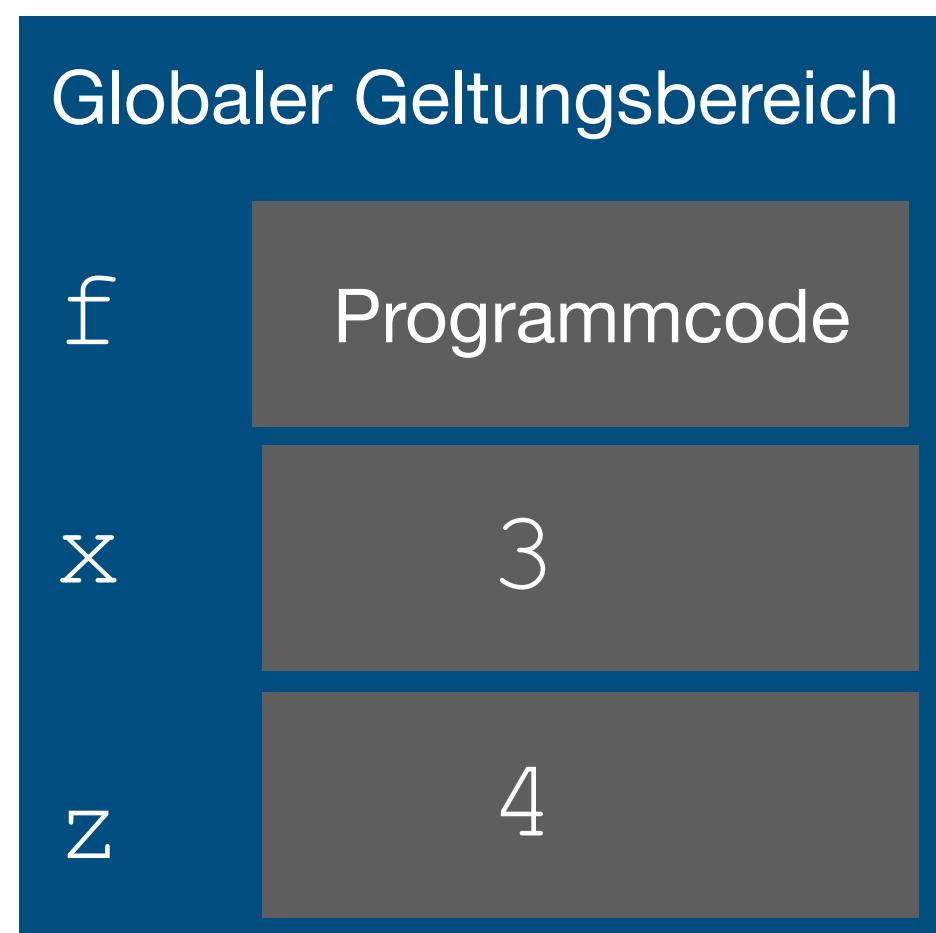
```
x=3  
z=f( x )
```



Geltungsbereich

```
def f( x ):  
    x=x+1  
print("innerhalb von f(x) : x=", x)  
return x
```

```
x=3  
z=f( x )
```



Geltungsbereich

```
def ist_gerade( i ):  
    """  
        Eingabe: i, ein positiver Integer  
        Gibt True zurueck, falls i gerade ist,  
        ansonsten False  
    """  
print("Gerade sind wir in ist_gerade")  
    i%2==0
```

Ohne Rückgabewert

- Python gibt den Rückgabewert **None** wenn keine **return**-Anweisung ausgeführt wird bzw. wenn gar keine solche vorhanden ist
- Repräsentiert die Abwesenheit eines Rückgabewerts

return

vs.

print

- `return` hat nur eine Bedeutung innerhalb einer Funktion
- **maximal ein `return`** wird innerhalb einer Funktion ausgeführt
- Code nach `return`-Anweisung innerhalb der Funktion wird nicht ausgeführt
- hat einen assoziierten Wert, der dem **Aufrufer** (z.B. Ausdruck) **zurückgeliefert** wird
- `print` kann **außerhalb** Funktionen ausgeführt werden
- mehrere `print`-Anweisungen sind in einer Funktion ausführbar
- Code innerhalb einer Funktion kann nach einer `print`-Anweisung ausgeführt werden
- Assoziierte Wert wird **in der Konsole** ausgegeben

Funktionen in Argumenten

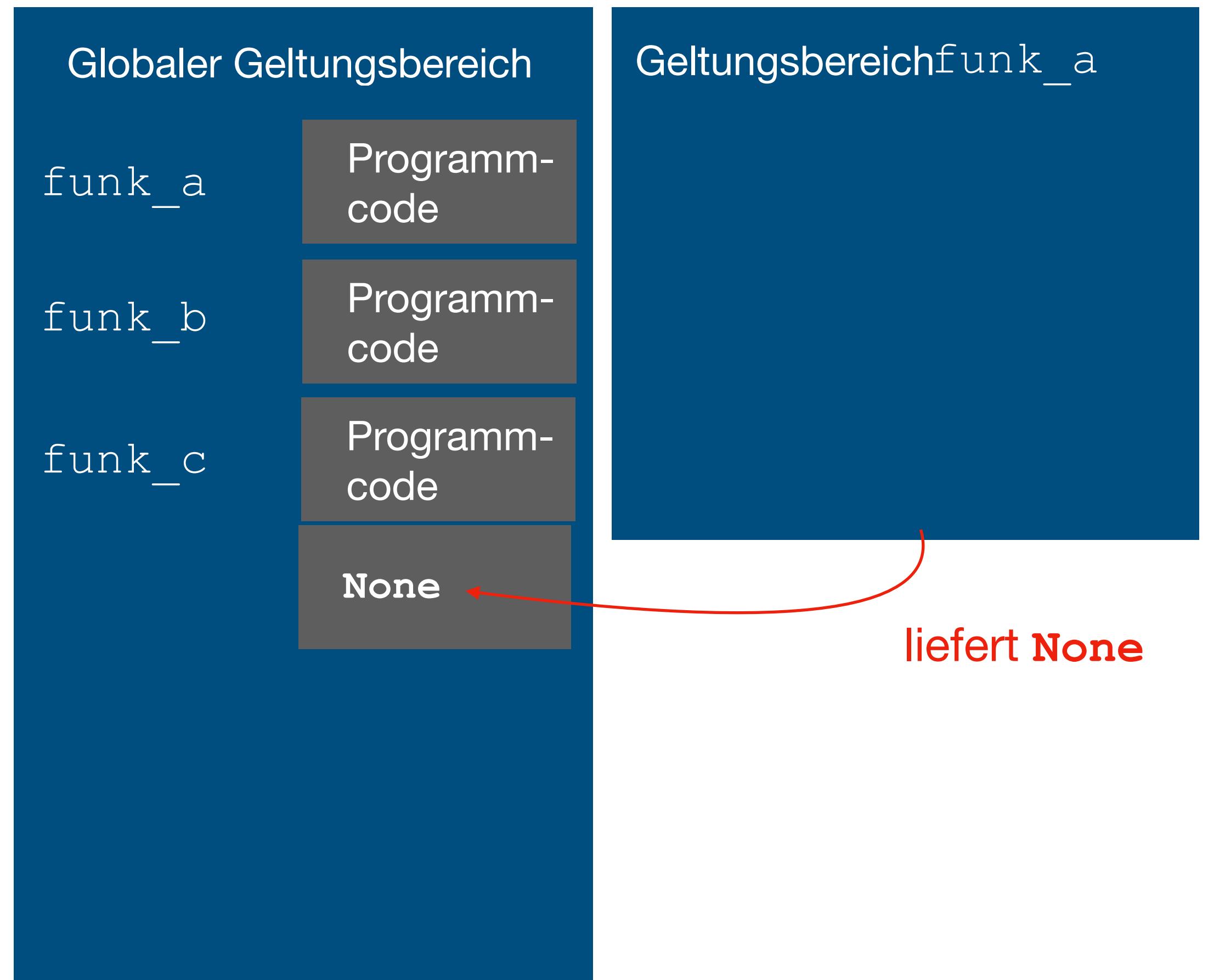
- Funktionen selbst können jeden möglichen Typ entgegennehmen, sogar Funktionen selbst!

```
def funk_a():
    print("innerhalb von funk_a ")
def funk_b(y):
    print("innerhalb von funk_b ")
    return y
def funk_c(z):
    print("innerhalb von funk_c ")
    return z()
print(funk_a())
print(5+funk_b(2))
print(funk_c(funk_a()))
```

Rufe funk_a auf, ohne Parameter
Rufe funk_b auf, mit einem Parameter
Rufe funk_c auf, mit einem Parameter, nämlich einer
Funktion

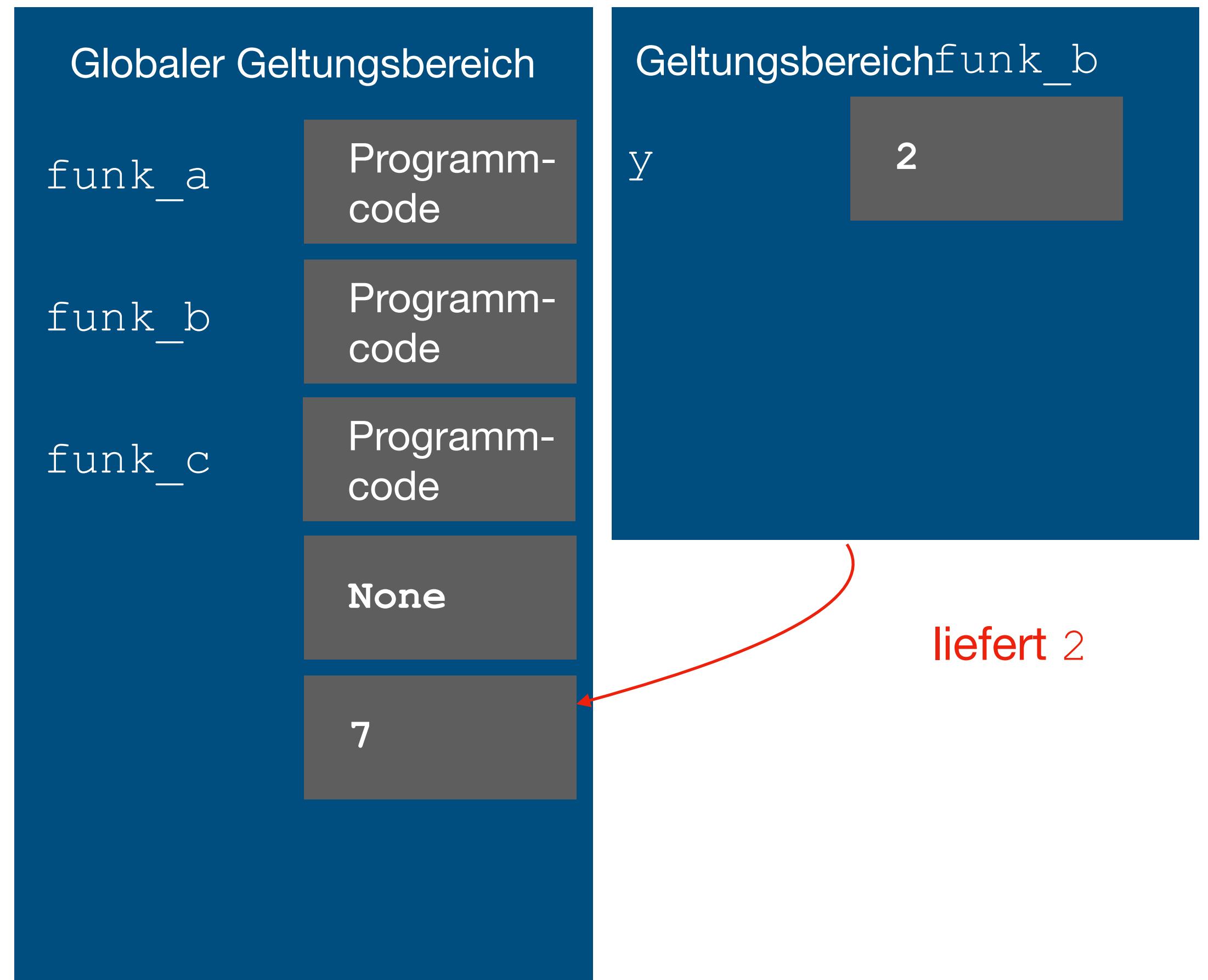
Funktionen in Argumenten

```
def funk_a():
    print("innerhalb von funk_a ")
def funk_b(y):
    print("innerhalb von funk_b ")
    return y
def funk_c(z):
    print("innerhalb von funk_c ")
    return z()
print(funk_a())
print(5+funk_b(2))
print(funk_c(funk_a))
```



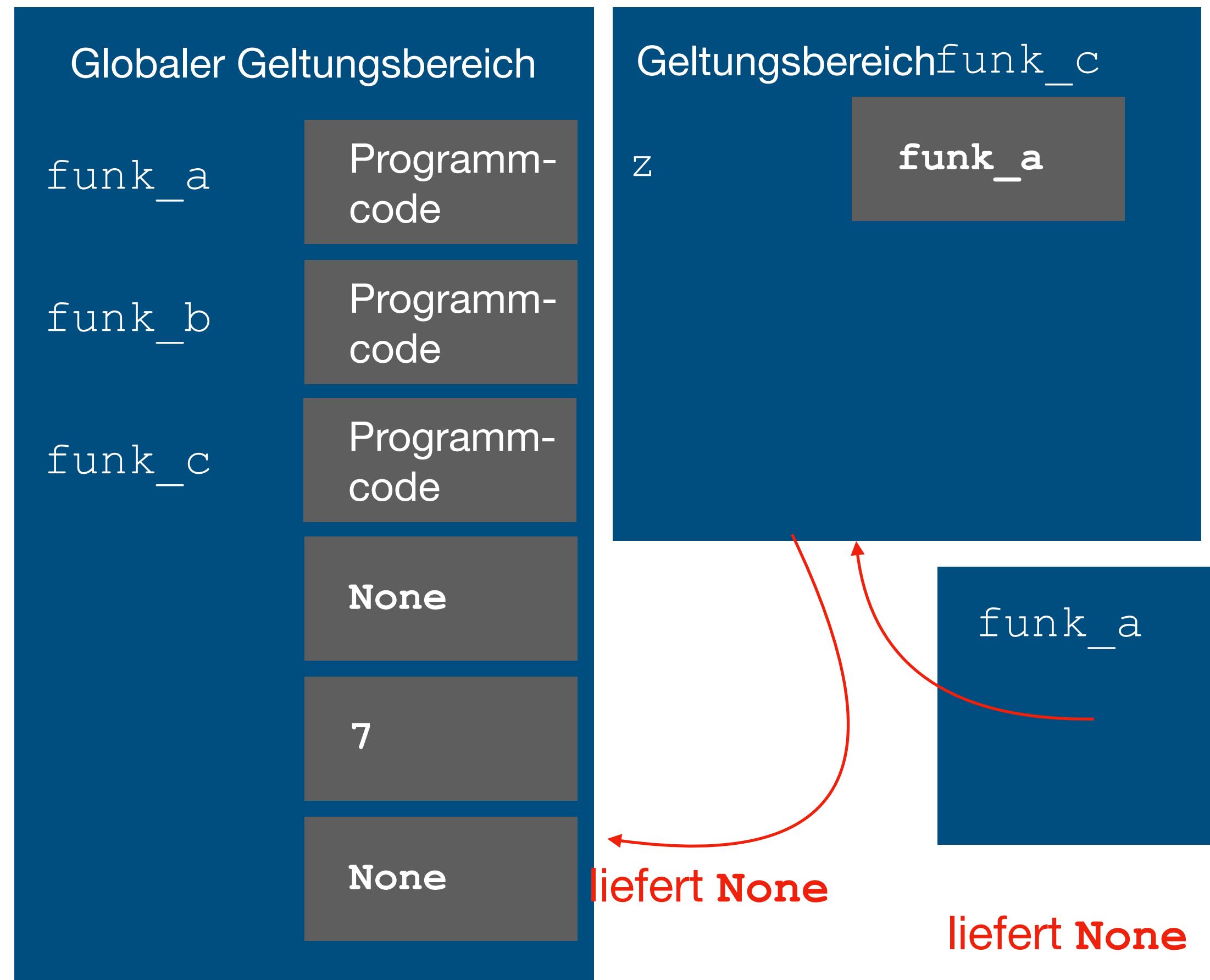
Funktionen in Argumenten

```
def funk_a():
    print("innerhalb von funk_a ")
def funk_b(y):
    print("innerhalb von funk_b ")
    return y
def funk_c(z):
    print("innerhalb von funk_c ")
    return z()
print(funk_a())
print(5+funk_b(2))
print(funk_c(func_a))
```



Funktionen in Argumenten

```
def funk_a():
    print("innerhalb von funk_a ")
def funk_b(y):
    print("innerhalb von funk_b ")
    return y
def funk_c(z):
    print("innerhalb von funk_c ")
    return z()
print(funk_a())
print(5+funk_b(2))
print(funk_c(funk_a))
```



Geltungsbereich, Beispiele

- Innerhalb einer Funktion kann auf Variablen zugegriffen werden, die außerhalb der Funktion definiert werden

Variable `x` wird innerhalb des Geltungsbereichs der Funktion `f` neu definiert

```
def f(y):  
    x=1  
    x+=1  
    print(x)  
    x=5  
    f(x)  
    print(x)
```

Zwei verschiedene Variablen vom Namen `x`, in unterschiedlichem Geltungsbereich

Es wird auf Variable `x` zugegriffen, die außerhalb des Geltungsbereichs der Funktion `g` ist

```
def g(y):  
    print(x)  
    print(x+1)
```

x=5
g(x)
print(x)

Auf Variable `x` wird zugegriffen, die sich im Geltungsbereich des Aufrufers befindet

```
def h(y):  
    x+=1
```

x=5
h(x)
print(x)

UnboundLocalError:
local variable 'x'
referenced before
assignment

Geltungsbereich, Beispiele

- Innerhalb einer Funktion kann auf Variablen zugegriffen werden, die außerhalb der Funktion definiert werden

```
def f(y):  
    x=1  
    x+=1  
    print(x)  
x=5  
f(x)  
print(x)
```

```
def g(y):  
    print(x)  
    print(x+1)  
  
x=5  
g(x)  
print(x)
```

```
def h(y):  
    x+=1  
  
x=5  
h(x)  
print(x)
```

Variable x ist im Geltungsbereich des Hauptprogramms

Funktionen in Argumenten

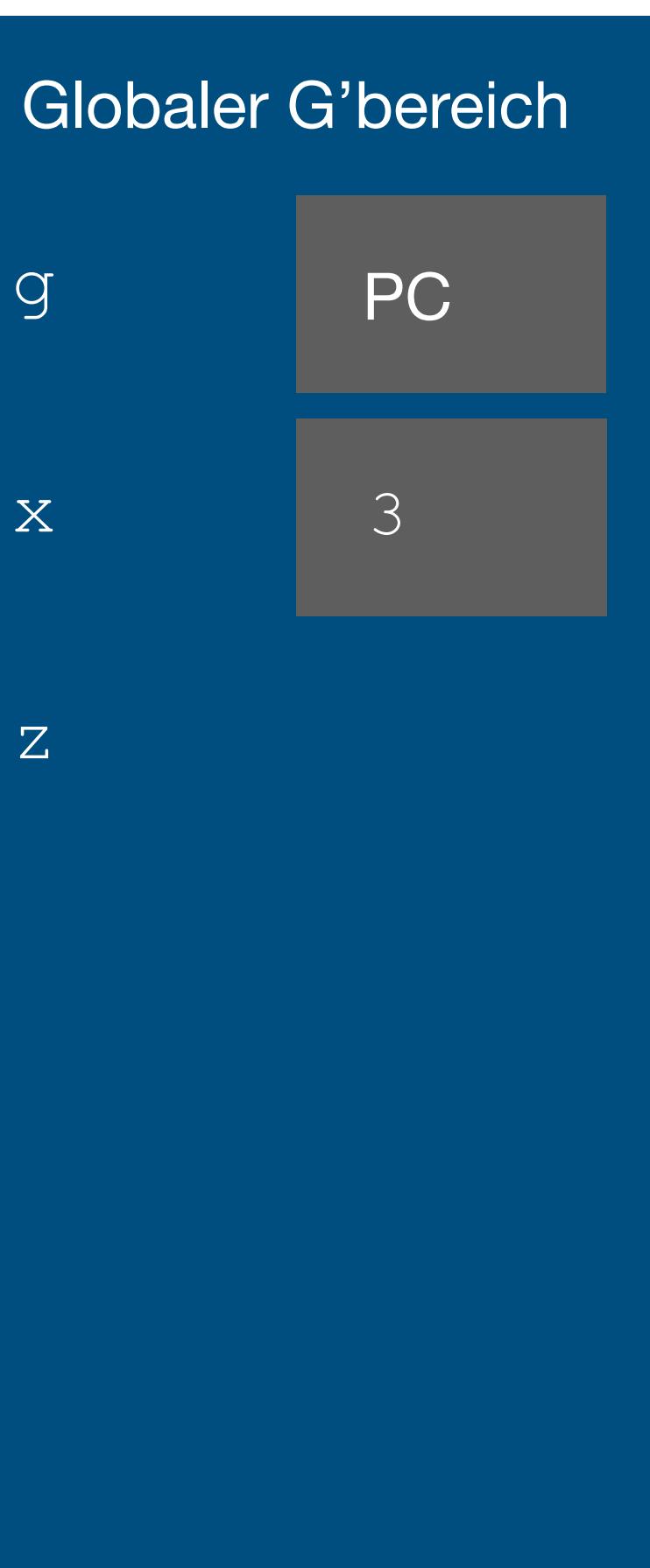
Im Folgenden steht **PC** für irgendein Programmcode

```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ",x)
        h(x)
    return x
x=3
z=g(x)
```

Funktionen in Funktionen

Im Folgenden steht **PC** für irgendeinen Programmcode

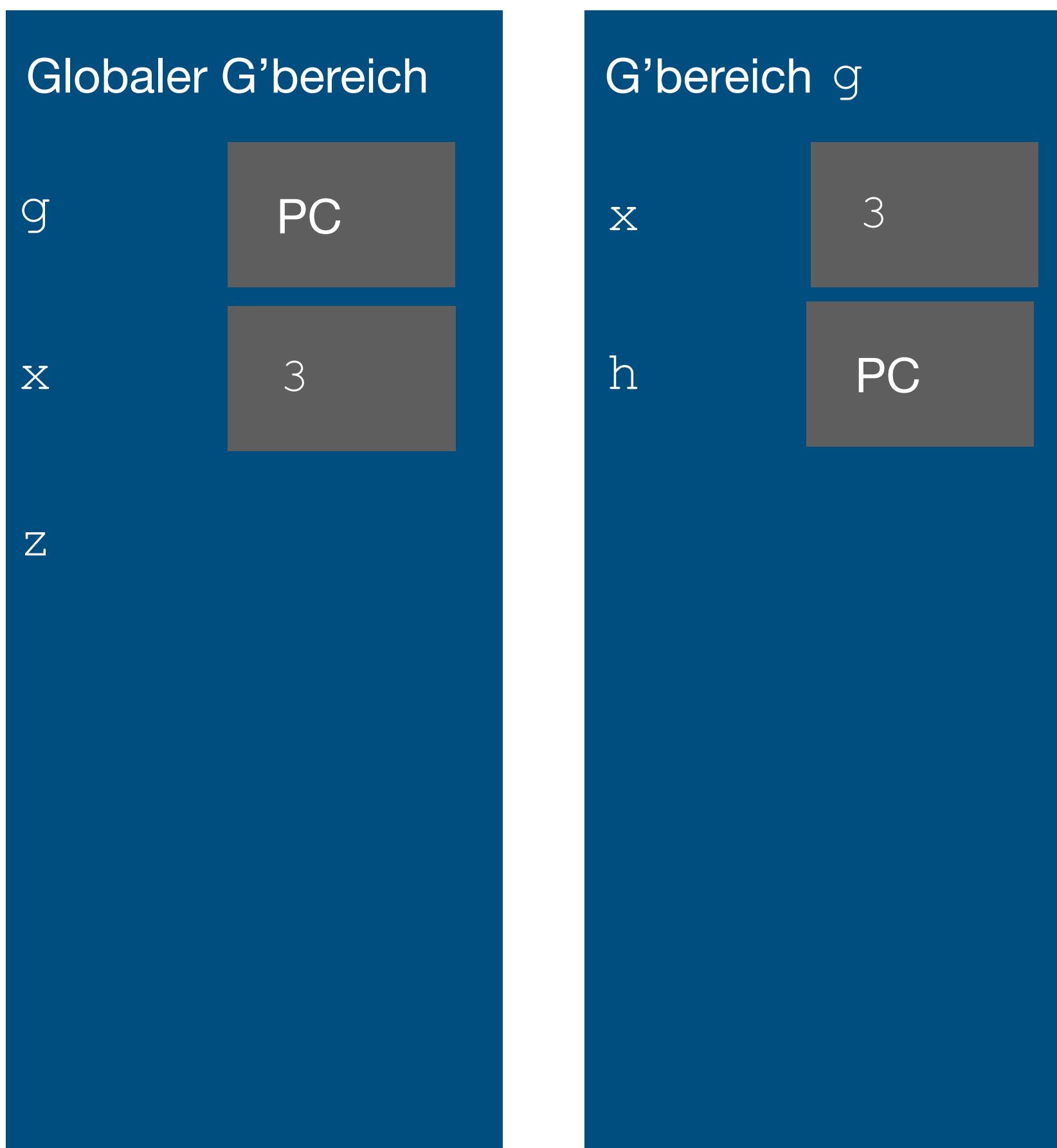
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

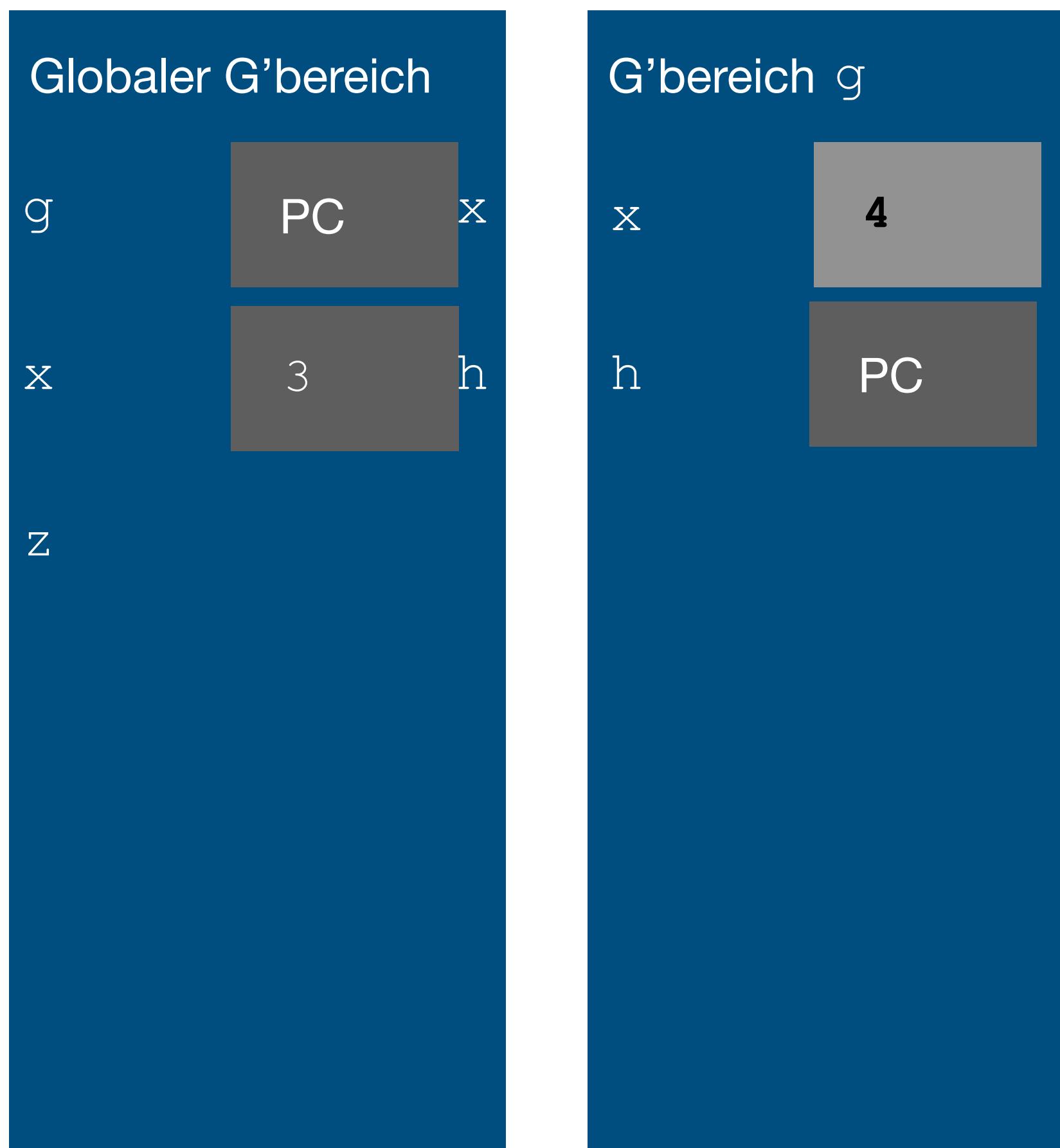
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

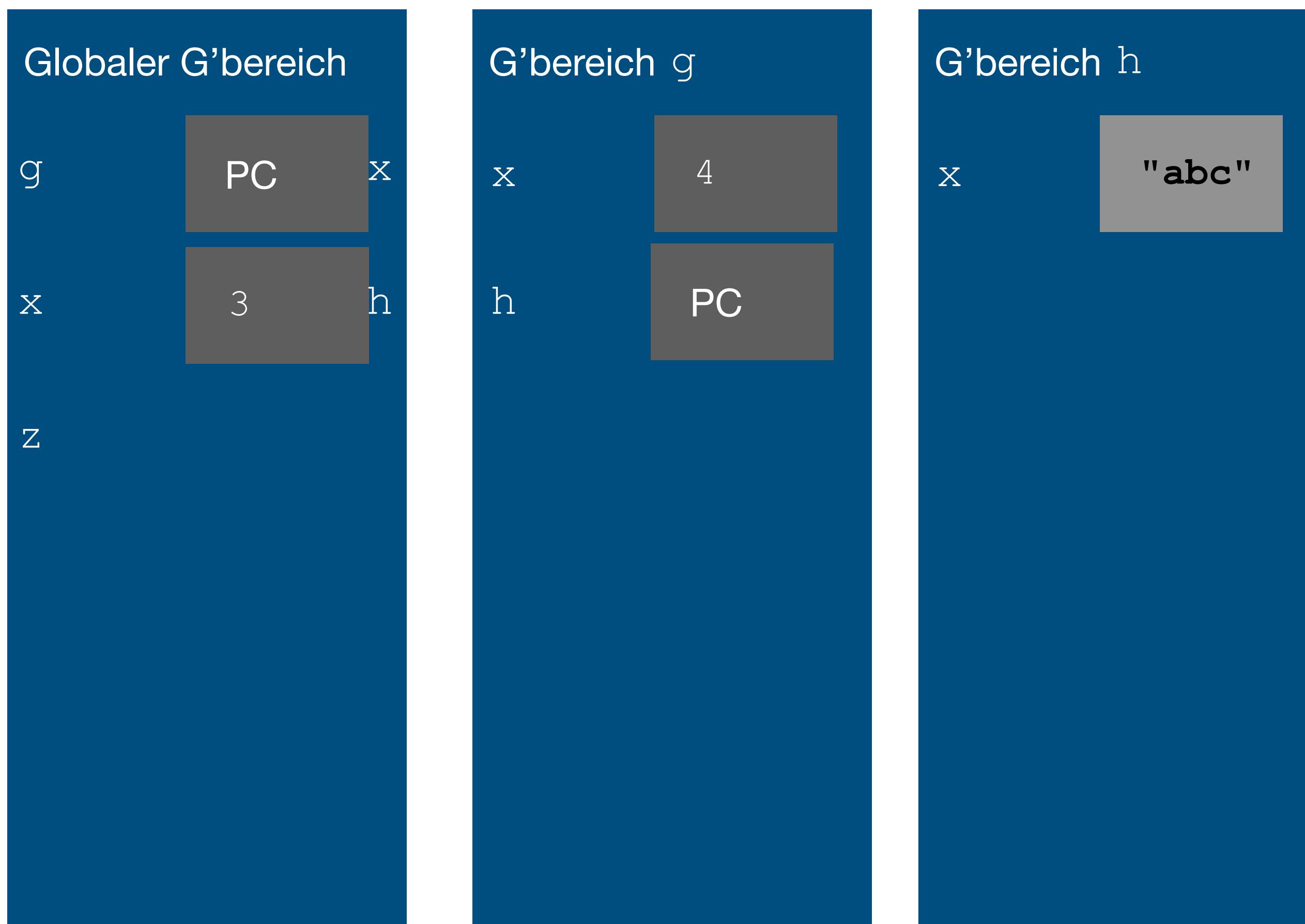
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

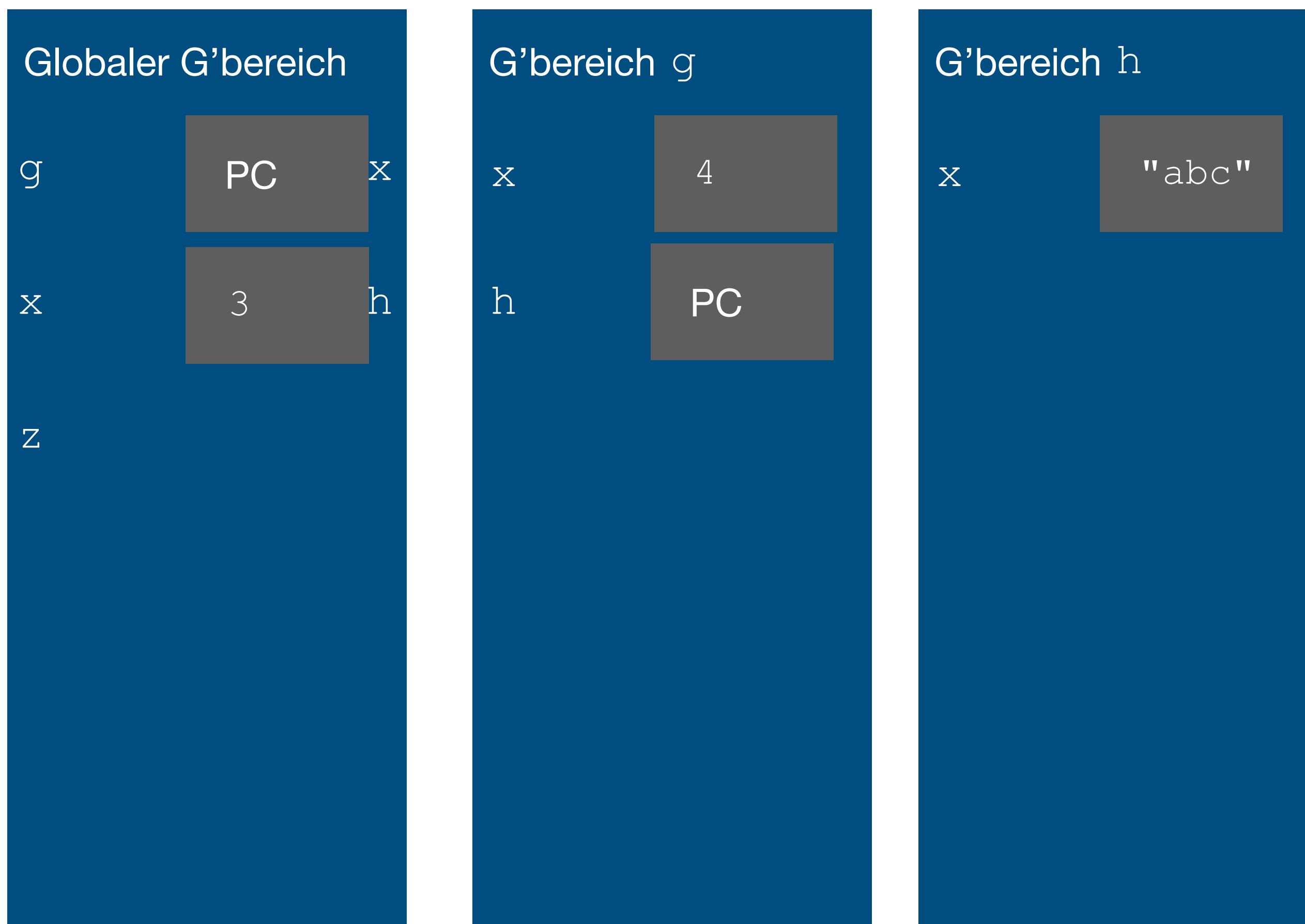
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

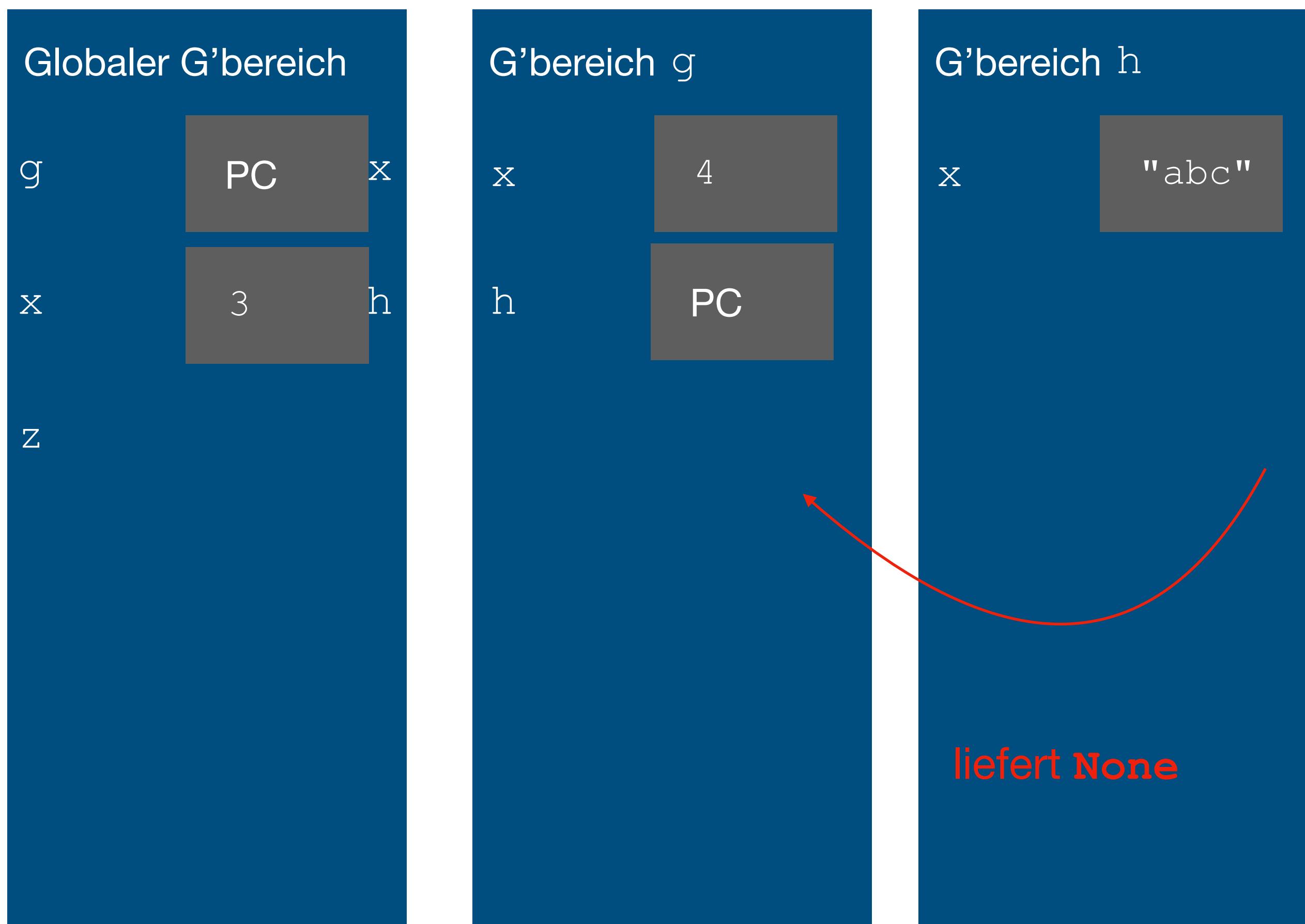
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

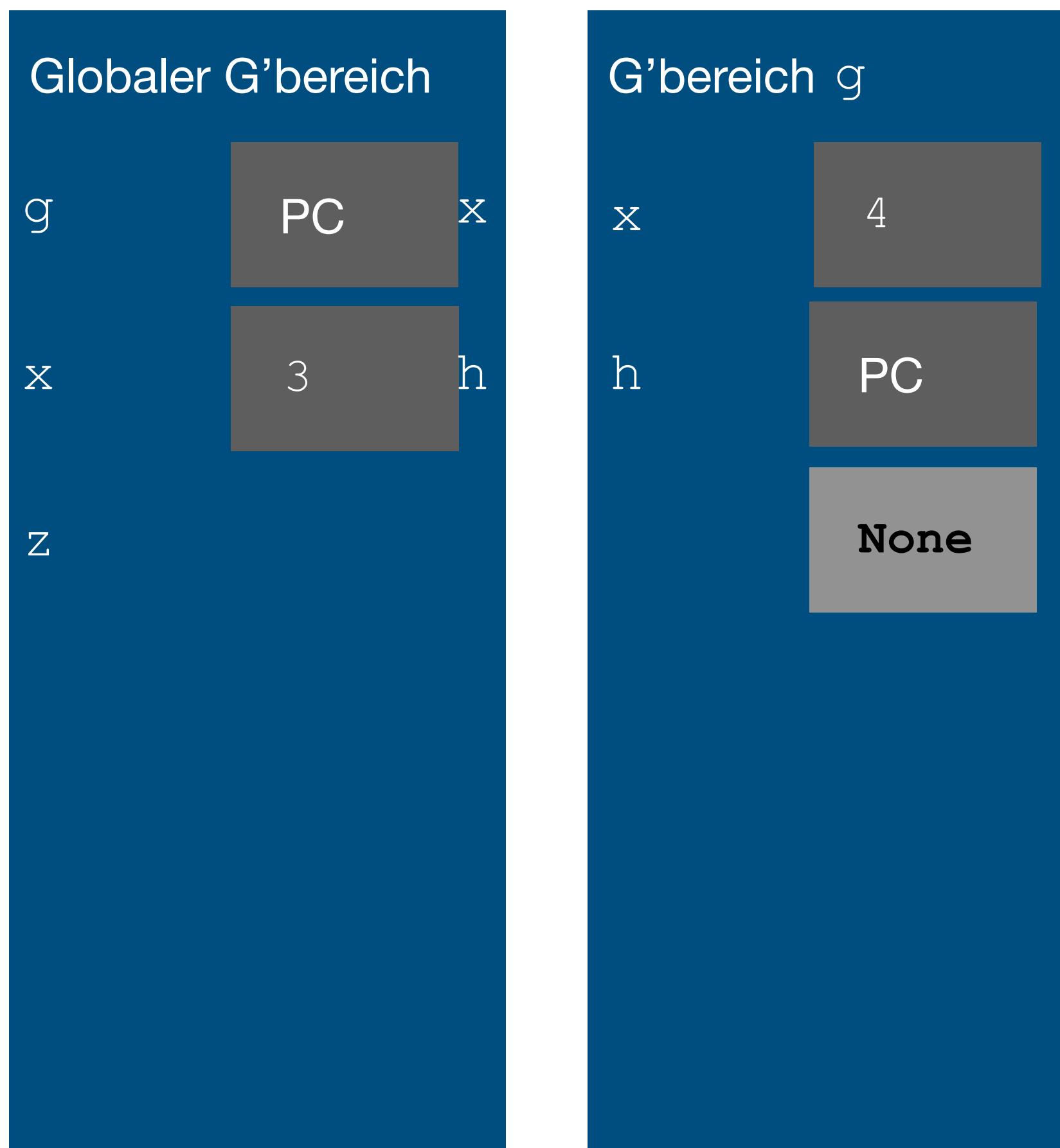
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

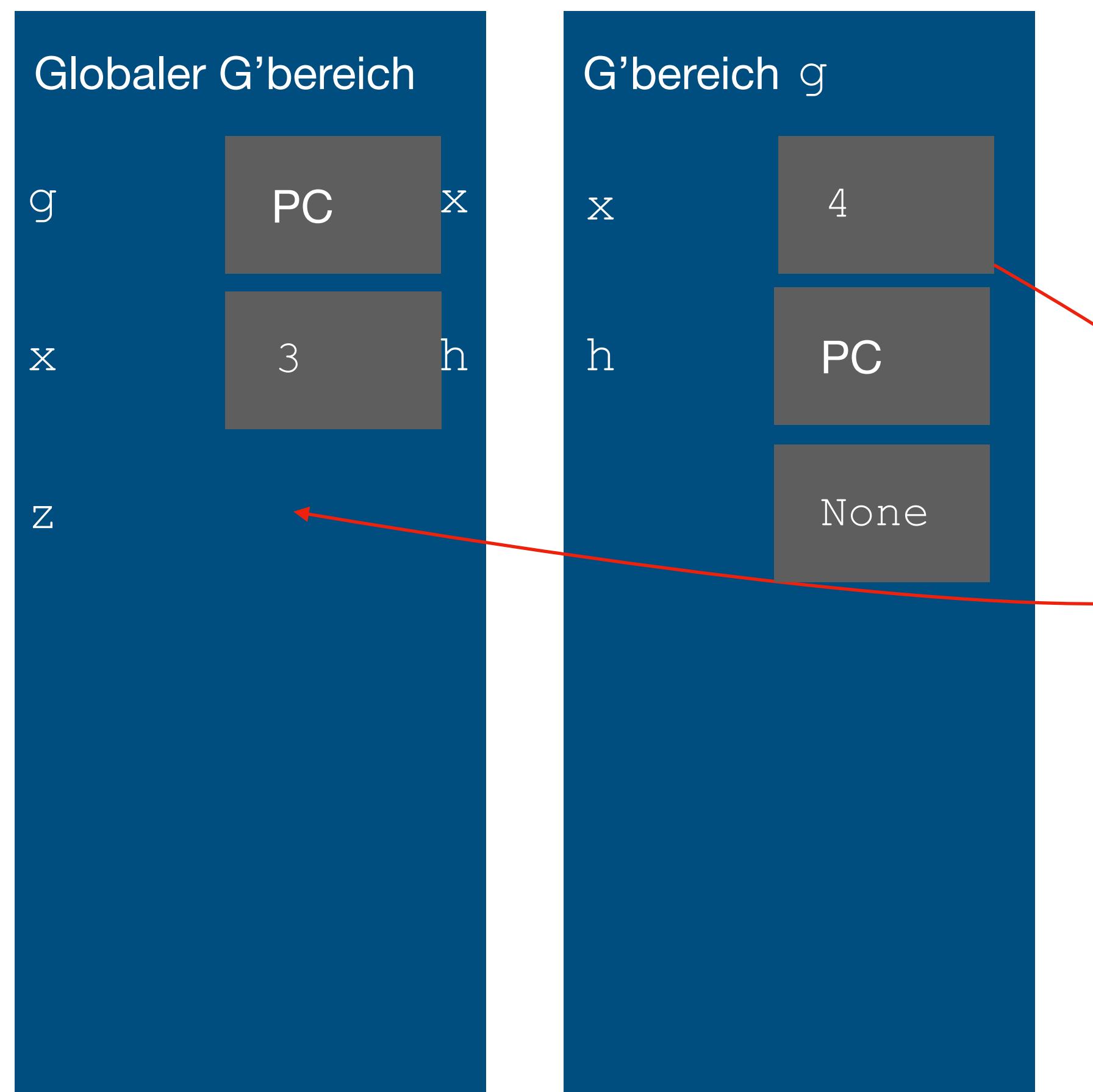
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

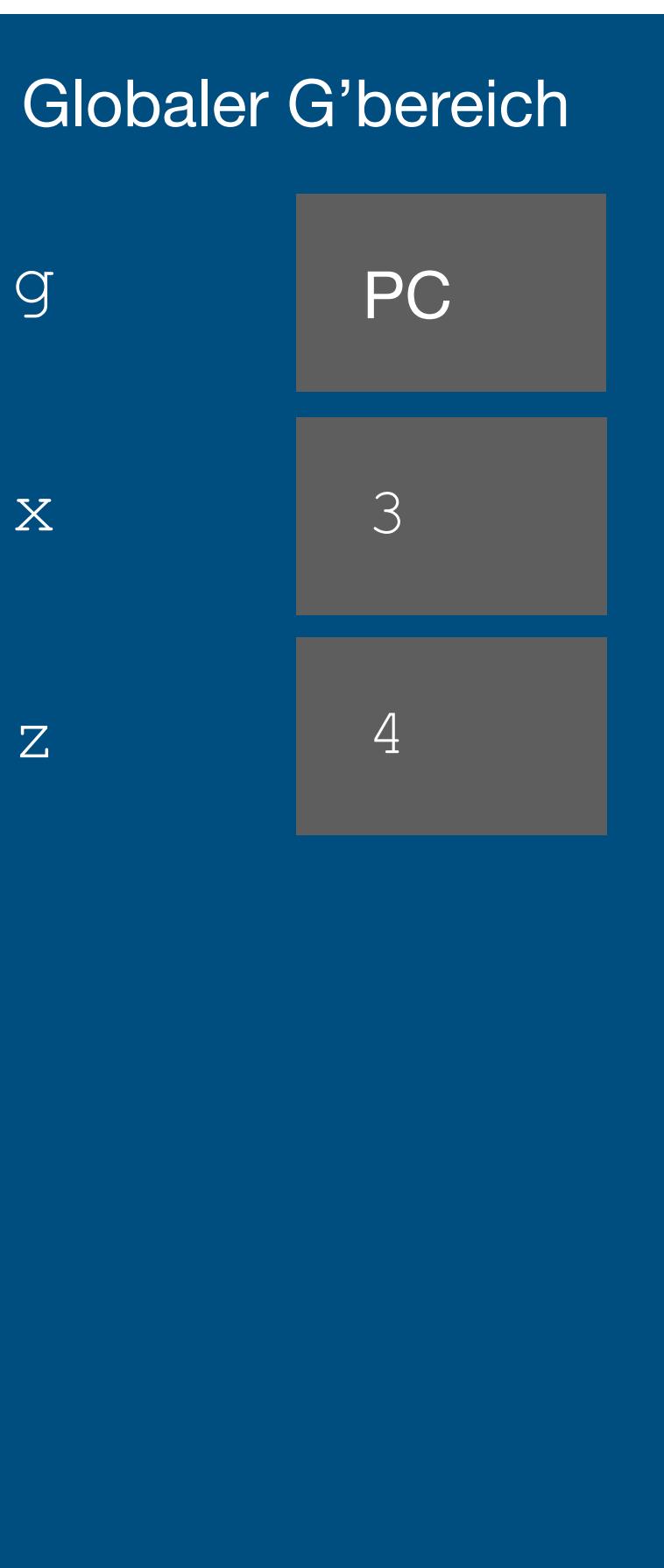
```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Funktionen in Argumenten

Im Folgenden steht **PC** für irgendeinen Programmcode

```
def g(x):
    def h(x):
        x="abc"
        x=x+1
        print("g: x= ", x)
        h(x)
    return x
x=3
z=g(x)
```



Erinnerung

Dekomposition und Abstraktion

- Dekomposition: Herstellung von Struktur
- Abstraktion: Unterdrücke Details
- Dekomposition und Abstraktion sind stark im Zusammenspiel
- Code kann dadurch mehrfach verwendet werden, muss aber nur einmal verbessert/modifiziert werden
- Wir haben Funktionen kennengelernt, diese werden wir ab jetzt vermehrt einsetzen

Tupel

- sind geordnete Sequenzen von Elementen (möglicherweise verschiedenem Typs)
- sind wie Strings **unveränderbar**
- werden durch Klammern repräsentiert

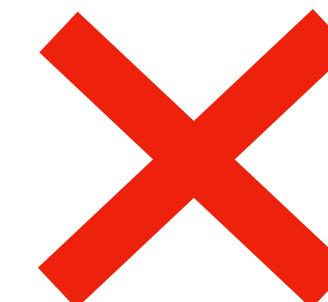
Extra Komma hier notwendig
um einelementiges Tupel
zu beschreiben

t1=()	
t=(2,"Kassel",3)	
t[0]	→ wertet sich zu 2 aus
(2,"Kassel",3)+(5,6)	→ wertet sich zu (2,"Kassel",3,5,6) aus
t[1:2]	→ Tupelausschnitt (Slice), wertet sich zu ("Kassel",) aus
t[1:3]	→ wertet sich zu ("Kassel",3) aus
len(t)	→ wertet sich zu 3 aus
t[1]=4	→ liefert einen Fehler, da Tupel unveränderbar sind

Tupel

- werden oft verwendet um Elemente einfach zu vertauschen

x=y
y=x



temp=x
x=y
y=temp



(x, y) = (y, x)

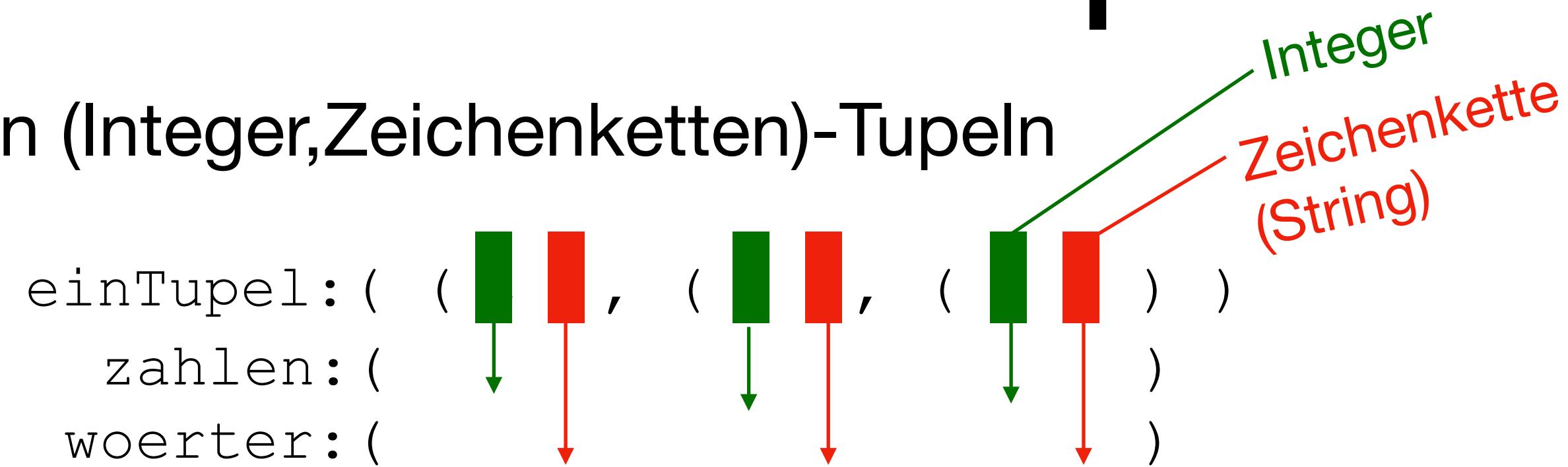


- werden verwendet um mehr als einen Parameter von einer Funktion zurückzugeben

```
def teiler_und_rest(x, y)           Integer Division
    t=x//y
    r=x%y
    return (t, r)
(teiler, rest)=teiler_und_rest(4, 5)
```

Manipulation von Tupeln

- Iteration über Tupel von (Integer,Zeichenketten)-Tupeln



```
def hole_daten(einTupel):  
    zahlen=() Leeres Tupel  
    woerter=()  
    for t in einTupel:  
        zahlen=zahlen + (t[0],) Einelementiges Tupel  
        if t[1] not in woerter:  
            woerter=woerter + (t[1],)  
durchschnitt=0.0  
summe=0  
for z in zahlen:  
    summe=summe+z  
if len(zahlen)>0:  
    durchschnitt=summe/len(zahlen)  
return (durchschnitt,woerter)
```

Listen

- sind **geordnete Sequenzen** von Informationen, auf die mittels Indizes zugegriffen werden können
- werden durch **eckige Klammern** [] beschrieben
- eine Liste enthält **Elemente**
 - die typischerweise homogen sind (bspw. nur Integers)
 - die auch von unterschiedlichem Typ sein können
- sind **veränderbar** (im Gegensatz zu Zeichenketten und Tupeln)

Indizierung und Ordnung

eine_liste=[] *Leere Liste*

L=[2, "a", 4, [1, 2]]

len(L) → wertet sich zu 4 aus

L[0] → wertet sich zu 2 aus

L[2]+1 → wertet sich zu 5 aus

L[3] → wertet sich zu [1, 2] aus, also einer Liste!

L[4] → liefert einen Fehler

i=2

L[i-1] → wertet sich zu "a" aus, da L[1] Wert "a" zugewiesen wurde

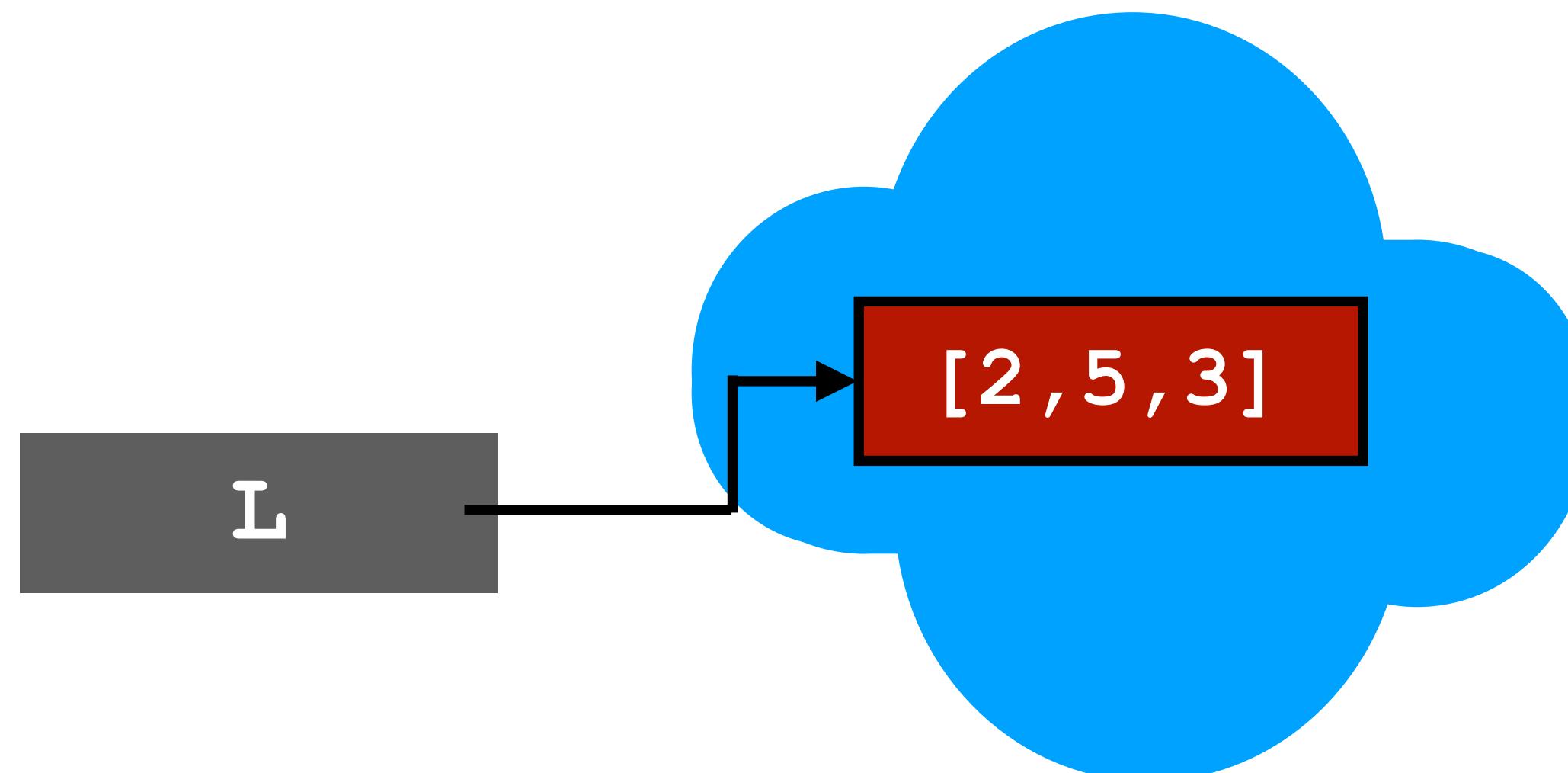
Ändern von Elementen

- sind **veränderbar**
- **Elemente** einer Liste können **Überschrieben** werden

```
L = [2, 1, 3]
```

```
L[1]=5
```

- L ist nun [2, 5, 3] jedoch immer noch **dasselbe Objekt** L



Über Listen iterieren

- angenommen wir wollen die Summe aller Listenelemente berechnen
- so etwas wird häufig benötigt

```
total=0  
for i in range(len(L)):  
    total+=L[i]  
print(total)
```

```
total=0  
for element in L:  
    total+=element  
print(total)
```

wie bei Zeichenketten
kann über die Elemente
einer Liste direkt iteriert
werden

- Man beachte
 - Listenelemente werden von 0 bis **len**(L) - 1 indiziert
 - **range**(n) liefert das Intervall ganzer Zahlen von 0 bis n-1

Operationen auf Listen

- Mittels `L.append(element)` wird der Liste `L` das Element `element` hinzugefügt

```
L=[2,1,3]           → L sieht nun so aus: [2,1,3,0]  
L.append(0)
```

- Was bedeutet der Punkt?
 - **Listen sind Objekte** in Python, alles ist in Python ein Objekt selbst Funktionen
 - Objekte **besitzen Daten**
 - Objekte besitzen **Funktionen** und **Methoden**, die die Daten dieser Objekte u.U. **verändern**
 - auf diese Funktionen/Methoden können mittels `objektname.methodename()` zugegriffen werden
 - über Objekte lernen wir später mehr im Rahmen der Objektorientierung

Operationen auf Listen: +

- Listen können aneinandergehängt werden mittels **Konkatenationsoperator +** (nicht kommutativ, da Listen geordnet sind)
- Listen können erweitert werden mittels `L.extend(antereListe)`

`L1=[2, 1, 3]`

`L2=[4, 5, 6]`

`L3=L1+L2`



`L3` ist nun `[2, 1, 3, 4, 5, 6]`
`L1` und `L2` bleiben unverändert

`L1.extend([0, 6])`



`L1` wurde nun zu `[2, 1, 3, 0, 6]`
erweitert

Operationen auf Listen: Entfernen von Elementen

- Elemente an einem konkreten Index können mittels **del (L[index])** entfernt werden
- Letztes Element der Liste kann mittels **L.pop ()** entfernt werden, Methode **pop ()** liefert dieses Element zusätzlich zurück
- Entfernen eines spezifischen Elements mittels **L.remove (element)**
 - entfernt das erste Auftreten von **element** in **L**
 - falls **element** mehrfach in **L** auftritt, wird nur das erste Auftreten entfernt
 - falls **element** in **L** gar nicht auftritt, gibt es einen Fehler

```
L=[2,1,3,6,3,7,0]
L.remove(2)      → L ist nun [1,3,6,3,7,0]
L.remove(3)      → L ist nun [1,6,3,7,0]
del(L[1])       → L ist nun [1,3,7,0]
L.pop()          → gibt 0 zurück, L ist nun [1,3,7]
```

Listen ↔ Zeichenketten

- Für jede Zeichenkette `s` liefert die Funktion `list(s)` die Liste aller Buchstaben zurück, die `s` entspricht
- Methode `s.split(wort)` liefert die Liste aller Zeichenketten, die zwischen allen Vorkommnissen der Zeichenkette `wort` in Zeichenkette `s` liegen (ohne Parameter wird " " als Standard genommen)
- a
- Verwenden Sie `" ".join(L)` um eine Liste von Buchstaben in eine Zeichenkette umzuwandeln; setzen sie Zwischen Anführungszeichen beliebige Zeichenketten

<code>s="I<3 cs"</code>	→ L ist eine Zeichenkette
<code>list(s)</code>	→ liefert ['I','<','3','c' , 's ']
<code>s.split("<")</code>	→ liefert ['I','3 cs']
<code>L=["a","b","c"]</code>	→ L ist eine Liste
<code>" ".join(L)</code>	→ liefert "abc"
<code>"haha".join(L)</code>	→ liefert "ahahabhbahac"

Weitere Operationen auf Listen

- **sorted**(L) liefert sortierte Variante von L zurück

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']
print(sorted(meineListe))
['a', 'e', 'i', 'o', 'u']
```

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um

```
meineListe = ['e', 'a', 'u', 'o', 'i']  
meineListe.reverse()  
print(meineListe)  
['i', 'o', 'u', 'a', , e']
```

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']
print(sorted(meineListe))
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um
 - meineListe = ['e', 'a', 'u', 'o', 'i']
meineListe.reverse()
print(L)
['i', 'o', 'u', 'a', , e']
- viele weitere Operationen unter
<https://docs.python.org/3/tutorial/datastructures.html>

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um
 - meineListe = ['e', 'a', 'u', 'o', 'i']
meineListe.reverse()
print(L)
['i', 'o', 'u', 'a', , e']
- viele weitere Operationen unter
<https://docs.python.org/3/tutorial/datastructures.html>

L=[9, 6, 0, 3]

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um

```
meineListe = ['e', 'a', 'u', 'o', 'i']  
meineListe.reverse()  
print(L)  
['i', 'o', 'u', 'a', , e']
```

- viele weitere Operationen unter
<https://docs.python.org/3/tutorial/datastructures.html>

L=[9, 6, 0, 3]

sorted(L)  liefert [0, 3, 6, 9], lässt L aber unverändert

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um

```
meineListe = ['e', 'a', 'u', 'o', 'i']  
meineListe.reverse()  
print(L)  
['i', 'o', 'u', 'a', , e']
```

- viele weitere Operationen unter
<https://docs.python.org/3/tutorial/datastructures.html>

L=[9, 6, 0, 3]

sorted(L)  liefert [0, 3, 6, 9], lässt L aber unverändert

L.sort()  L ist nun [0, 3, 6, 9]

Weitere Operationen auf Listen

- **sorted(L)** liefert sortierte Variante von L zurück

```
meineListe = ['e', 'a', 'u', 'o', , i']  
print(sorted(meineListe))  
['a', 'e', 'i', 'o', 'u']
```

- L.sort() sortiert L
- L.reverse() dreht die (Reihenfolge der) Liste L um

```
meineListe = ['e', 'a', 'u', 'o', 'i']  
meineListe.reverse()  
print(L)  
['i', 'o', 'u', 'a', , e']
```

- viele weitere Operationen unter
<https://docs.python.org/3/tutorial/datastructures.html>

L=[9, 6, 0, 3]

sorted(L) → liefert [0, 3, 6, 9], lässt L aber unverändert

L.sort() → L ist nun [0, 3, 6, 9]

L.reverse() → L ist nun [9, 6, 3, 0]

Veränderung, Aliasing, Klonen



wichtiges aber auch
schwieriges Thema!

**Wie so oft, schlagen Sie bei Unklarheiten unter
www.pythontutor.com
nach!**

Speicherung von Listen

- Erinnerung: Listen sind **veränderlich**
- verhalten sich also anders als unveränderliche Objekte wie Zeichenketten und Tupel
- wird als Objekt im Hauptspeicher abgelegt
- Variablenname zeigt auf dieses Objekt
- **jeder** Variablenname, der auf dieses Objekt zeigt ist betroffen
- Wichtiges Stichwort beim Arbeiten mit Listen: **Nebeneffekte**

Eine Analogie

- Attribute einer Person
 - Sänger, reich,...
- sie könnte unter vielen verschiedenen Namen bekannt sein
- Alle Kosenamen meinen ein und dieselbe Person
 - wir könnten der Person gewisse Attribute geben

Justin Bieber	Sänger	reich	Unruhestifter
---------------	--------	-------	---------------

- ...alle seine Kosenamen beziehen sich auf seine alten Attribute und vielleicht auch auf neue

The Bieb	Sänger	reich	Unruhestifter
JBeebs	Sänger	reich	Unruhestifter

Aliase

- **heiss** ist ein Alias für **warm** - wenn wir unter **warm** etwas anderes verstehen, verstehen wir unter **heiss** auch etwas anderes
- Die Methode `append()` hat Nebeneffekte!

```
a=1  
b=a  
b=2  
print(a)  
print(b)
```

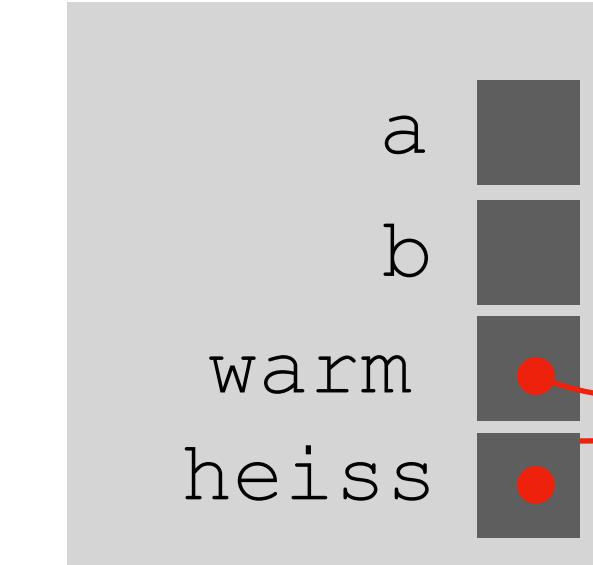
Ausgabe

```
1  
2  
['rot', 'gelb', 'orange', 'pink']  
['rot', 'gelb', 'orange', 'pink']
```

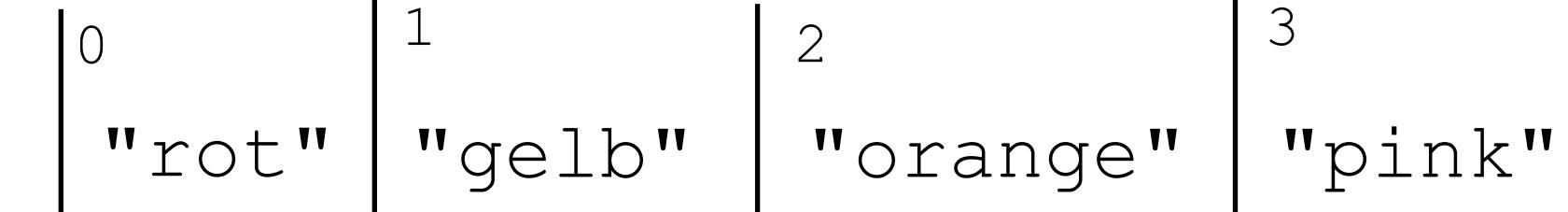
```
warm= ["rot", "gelb", "orange"]  
heiss=warm  
heiss.append("pink")  
print(heiss)  
print(warm)
```

Objekte

Frames



Liste



Klonen von Listen

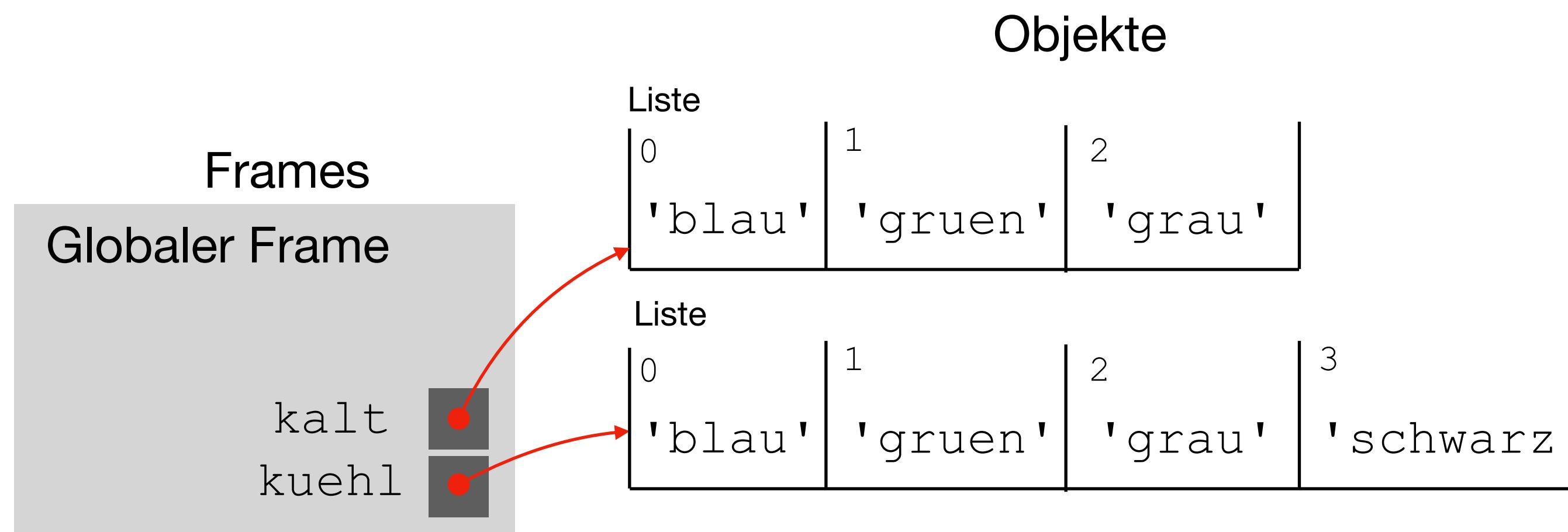
- Kreieren (als neues Objekt) einer Liste L, indem jedes Element kopiert wird

```
kopie=L[:]
```

```
kalt=['blau', 'gruen', 'grau']
kuehl=kalt[:]
kuehl.append('schwarz')
print(kuehl)
print(kalt)
```

Ausgabe

```
['blau', 'gruen', 'grau', 'schwarz']
['blau', 'gruen', 'grau']
```



Sortieren von Listen

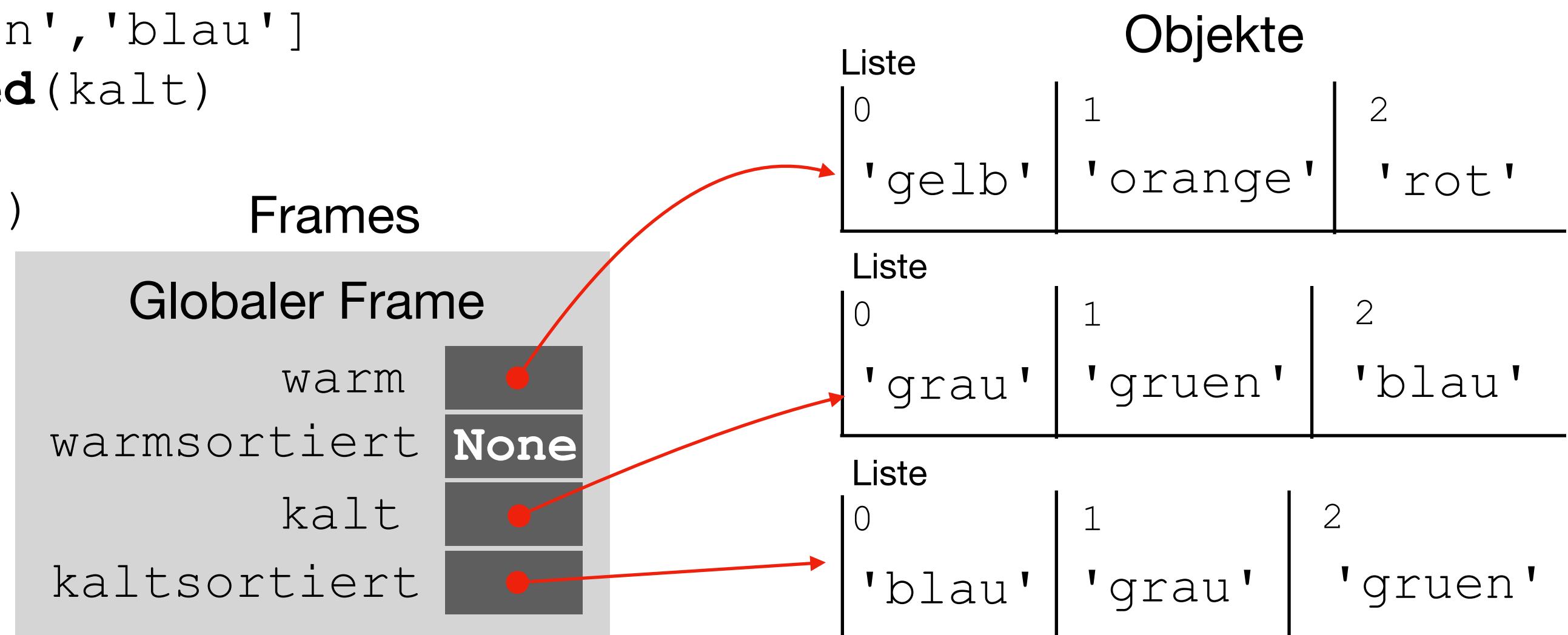
- Die Funktion **sort()** verändert die Liste, liefert aber nichts zurück
- Der Aufruf von **sorted()** verändert die Liste selbst nicht, das Ergebnis muss jedoch einer Variablen zugeordnet werden um verwendet werden zu können

```
warm=['rot', 'gelb', 'orange']
warmsortiert=warm.sort()
print(warm)
print(warmsortiert)
```

```
kalt=['grau', 'gruen', 'blau']
kaltsortiert=sorted(kalt)
print(kalt)
print(kaltsortiert)
```

Ausgabe

```
['gelb', 'orange', 'rot']
None
['grau', 'gruen', 'blau']
['blau', 'grau', 'gruen']
```



Listen von Listen von...

- Listen können **verschachtelt** sein
- Nebeneffekte bei verschachtelten Listen sind immer noch möglich nach Kopie!

```
warm= ['gelb', 'orange']
heiss=['rot']
hellefarben=[warm]
hellefarben.append(heiss)
print(hellefarben)
heiss.append('pink')
print(heiss)
print(hellefarben)
```

Ausgabe

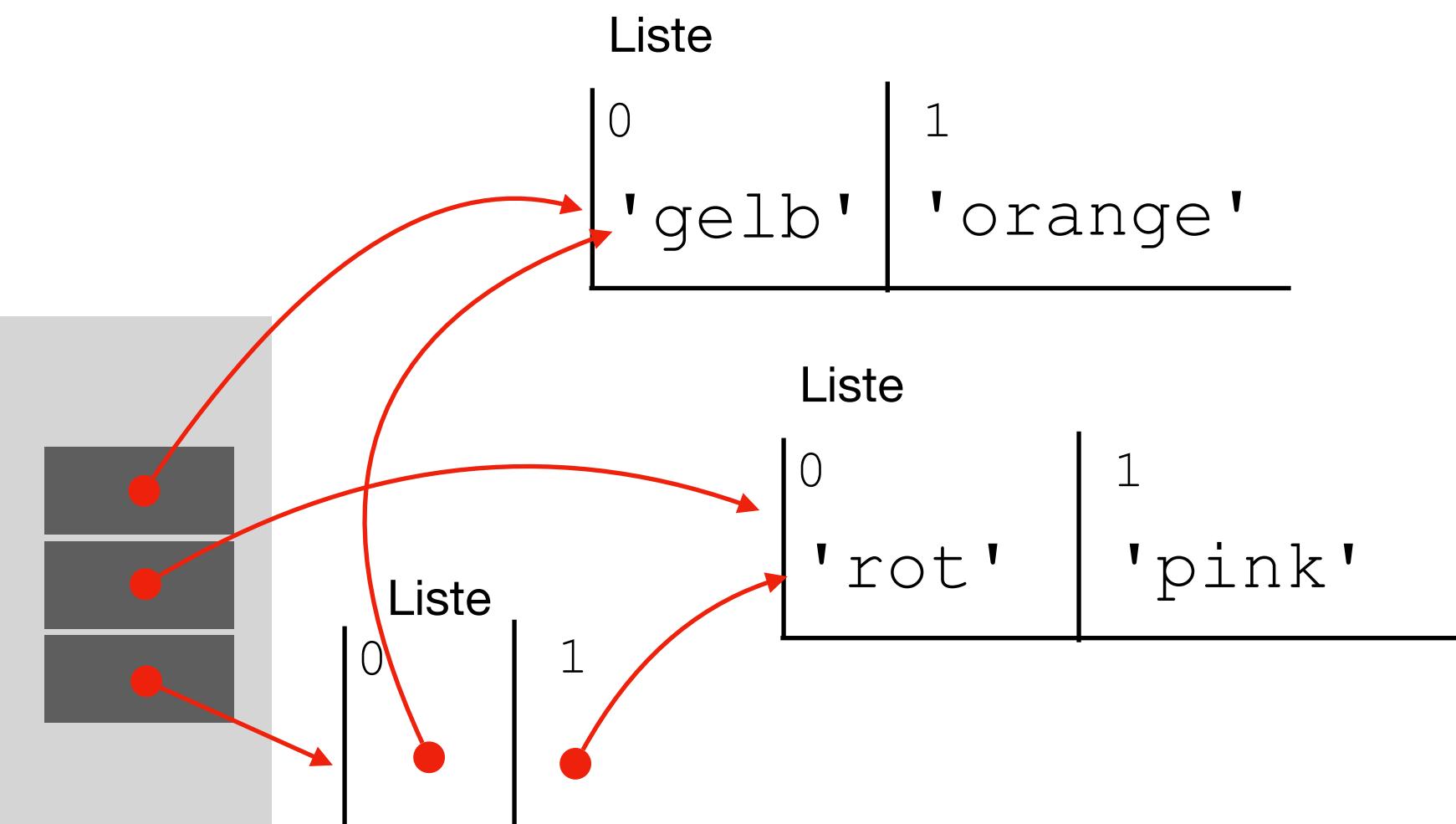
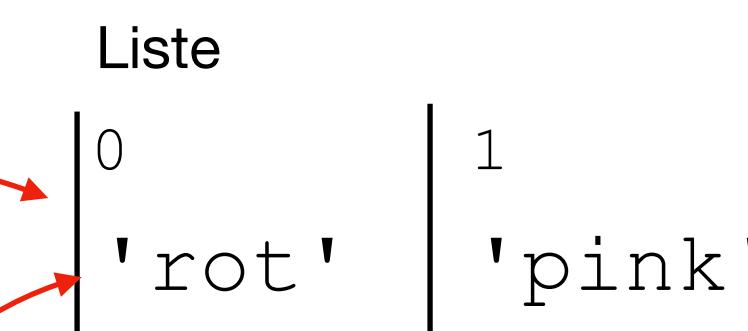
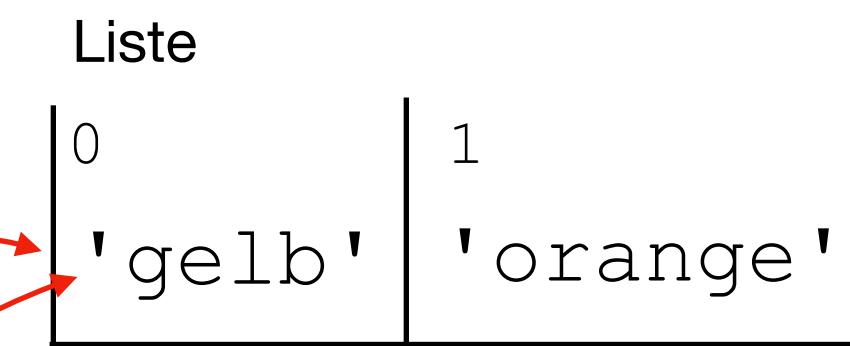
```
[ ['gelb', 'orange'], ['rot'] ]
['rot', 'pink']
[ ['gelb', 'orange'], ['rot', 'pink'] ]
```

Frames

Globaler Frame

warm
heiss
hellefarben

Objekte

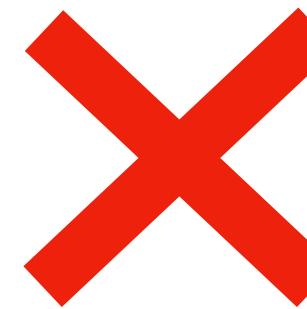


Veränderung und Iteration

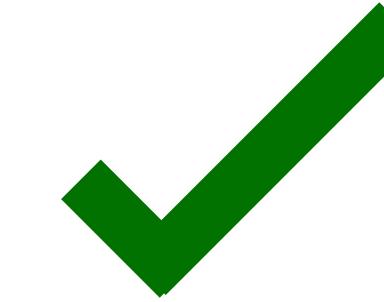
- eine Liste zu verändern, über die gerade iteriert wird, sollte vermieden werden

```
def entferne_duplikate(L1,L2):  
    for e in L1:  
        if e in L2:  
            L1.remove(e)
```

```
L1=[1,2,3,4]  
L2=[1,2,5,6]  
entferne_duplikate(L1,L2)
```



```
def entferne_duplikate(L1,L2):  
    L1_kopie=L1[:]  
    for e in L1_kopie:  
        if e in L2:  
            L1.remove(e)
```



- Nach Ausführung des linken Programms ist L1 gleich [2, 3, 4] statt [3, 4].

Warum?

- Python benutzt einen internen Zähler, um sich den aktuellen Index im Schleifendurchlauf zu merken
- Die Veränderung der Liste ändert auch die Länge und die Indizierung der Elemente
- Die Schleife hat also keinen Zugriff auf Index 2 der ursprünglichen Liste L1