Programmieren mit Python

Skript (Teil I)

5. Februar 2023

rechentrainer.py

```
1  import random
2  import time
3
4  operanden = [2, 3, 4, 5, 6, 7, 8, 9]
5
6  a = random.choice(operanden)
7  b = random.choice(operanden)
8
9  c = a * b
10
11 frage = '{} * {} = '.format(a, b)
12 antwort = '{} * {} = {}'.format(a, b, c)
13
14 print(frage)
15 time.sleep(5)
16 print(antwort)
```

Erläuterungen

Zeilen 1–2: Laden der Module für Zufalls- und Zeitfunktionen

Zeile 4: Die Liste mit den Zahlen 2, 3, . . . , 9 wird der Variablen `operanden` zugewiesen.

Zeilen 6–7: Mit der Methode `choice` wird jeweils ein zufälliges Element aus der Liste `operanden` ausgewählt und der Variablen `a` bzw. `b` zugewiesen.

Zeile 9: Die Werte der Variablen `a` und `b` werden mit dem Operator `*` multipliziert und das Resultat der Variablen `c` zugewiesen.

Zeilen 11–12: Die `format`-Methode setzt die Werte von `a`, `b` und `c` in dieser Reihenfolge an die Stelle der Platzhalter {} in der Zeichenkette (String) '. . .' ein. Die Strings werden den Variablen

Übung 1.1

1. Starte IDLE
2. Öffne eine leere Datei, speichere diese unter dem Namen `rechentrainer.py` ab.
3. Schreibe den obigen Code ab.
4. Führe den Code mit dem Menübefehl „Run/Run module“ aus.
5. Falls eine Fehlermeldung erscheint, lies sie aufmerksam durch, korrigiere den Fehler und mache bei 4. weiter; andernfalls gehe zum nächsten Schritt.
6. Falls das Programm nicht das Gewünschte macht, prüfe es nochmals auf Fehler und mache bei 4. weiter; andernfalls hast du die Aufgabe gelöst.

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

(b) Wie muss der Code abgeändert werden, wenn man mit dem Programm die Addition statt der Multiplikation üben will?

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

(b) Wie muss der Code abgeändert werden, wenn man mit dem Programm die Addition statt der Multiplikation üben will?

Zeile 9: `c = a + b`

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

(b) Wie muss der Code abgeändert werden, wenn man mit dem Programm die Addition statt der Multiplikation üben will?

Zeile 9: `c = a + b`

Zeilen 11 und 12: `*` durch `+` ersetzen

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

(b) Wie muss der Code abgeändert werden, wenn man mit dem Programm die Addition statt der Multiplikation üben will?

Zeile 9: `c = a + b`

Zeilen 11 und 12: `*` durch + ersetzen

(c) Wie muss der Code abgeändert werden, wenn bis zum Anzeigen des richtigen Resultats 3 Sekunden vergehen sollen?

Übung 1.2

(a) Wie muss der Code abgeändert werden, wenn auch der Operand 10 in den Rechnungen vorkommen soll?

Zeile 4: `operanden = [1, 2, ..., 9, 10]`

(b) Wie muss der Code abgeändert werden, wenn man mit dem Programm die Addition statt der Multiplikation üben will?

Zeile 9: `c = a + b`

Zeilen 11 und 12: `*` durch + ersetzen

(c) Wie muss der Code abgeändert werden, wenn bis zum Anzeigen des richtigen Resultats 3 Sekunden vergehen sollen?

Zeile 15: `time.sleep(3)`

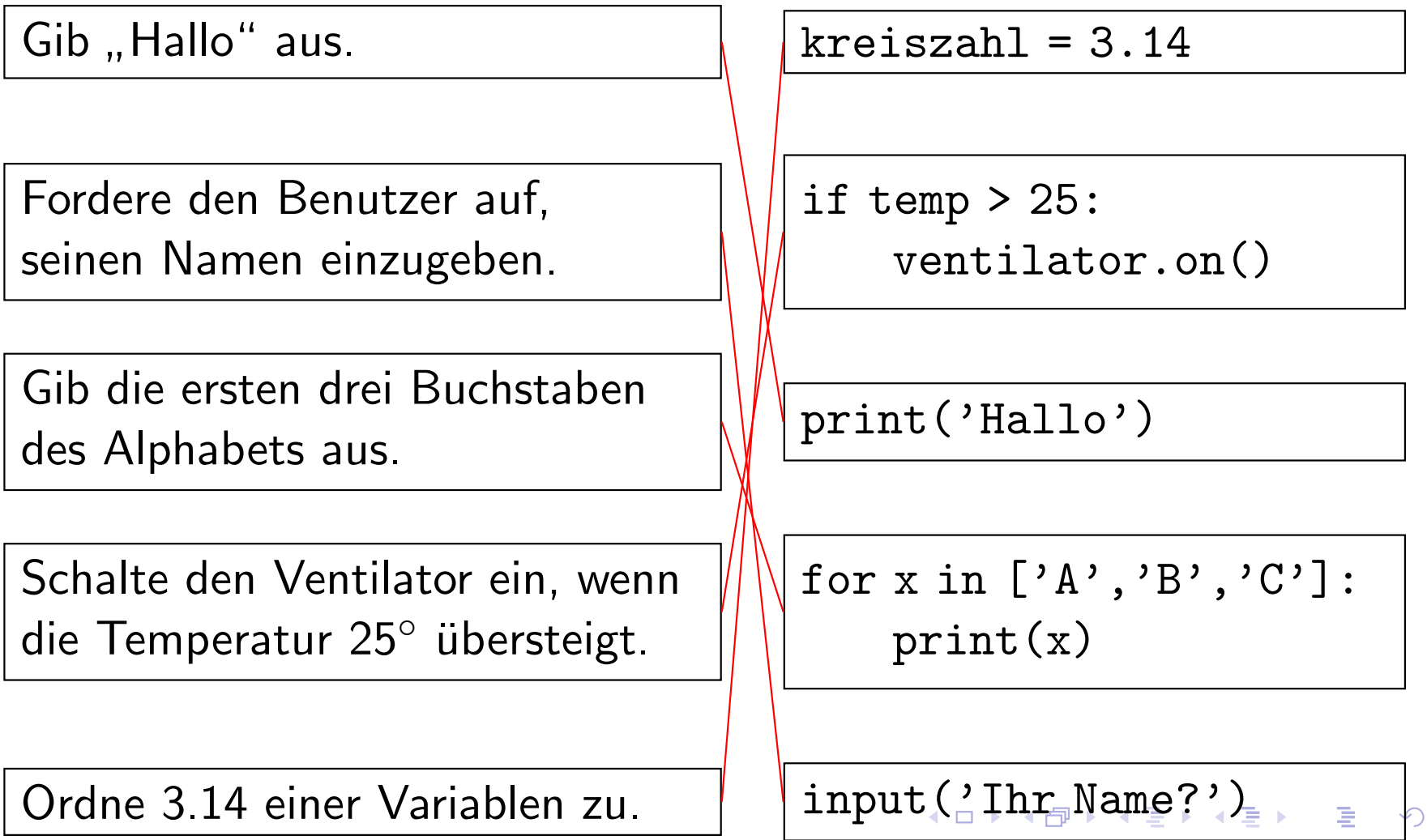
Übung 1.3

Finde alle Fehler im Quellcode.

```
1  import random
2  import time
3
4  operand = [2, 3, 4, 5, 6, 7  8, 9)
5
6  a = fandom.choice(operanden)
7  b = fandom.choice(operanden)
8
9  c = a * b
10
11 frage = '{} * {} = .'format(a, b)
12 antwort = '{} * {} = {}'.format(a, b, c)
13
14 print(frage)
15 time.sleep(5)
16 print(Antwort)
```

Übung 1.4

Verbinde jede Aktion mit dem zugehörigen Codefragment.



Aufgabe 1.5

Beantworte mit Hilfe des Internets die folgenden Fragen und fasse die Antwort in 1–2 kurzen Sätzen präzise zusammen.

(a) Wer hat die Programmiersprache Python entwickelt und woher hat sie ihren Namen?
Python wurde von Guido van Rossum anfangs der 1990er Jahre entwickelt. Der Name steht im Zusammenhang mit der Komikergruppe „Monty Python“, die zu jener Zeit sehr populär war.

(b) Was bedeutet der Begriff algorithmisches Denken?
Algorithmisches Denken beschreibt die Art, Probleme (Aufgaben) so zu analysieren, dass die von einem Computer bewältigt werden kann.

(c) Was ist in der Informatik ein Interpreter?
Ein Interpreter ist ein Computerprogramm, dass ein anderes Computerprogram (Quelltext) liest, analysiert, in

Auf den Punkt gebracht

- Algorithmisches Denken ist eine in der Informatik übliche Weise, Probleme so zu betrachten, dass sie von einem Computer gelöst werden können.
- Um eine Aufgabe zu lösen, muss sie in mehrere einfache Aktionen zerlegt werden die dann *Anweisungen* heissen.
- Eine solche Folge von Anweisungen wird *Algorithmus* genannt.
- Einen Algorithmus mit einem Kochrezept zu vergleichen ist nicht ganz falsch aber auch nicht ganz richtig. Zwar besteht auch ein Kochrezept aus einer Folge von Anweisungen aber Kochrezpte beschreiben die Aktionen nicht präzise genug, um von einem Computer ausgeführt werden zu können („Salz und Pfeffer nach Belieben“).

- Anweisungen führen elementare Aufgaben aus. Sie sorgen dafür, dass Daten gespeichert, Resultate berechnet oder die Ausführung anderer Anweisungen durch Verzweigungen und Schleifen gesteuert werden.
- Es gibt verschiedene Arten, einen Algorithmus zu beschreiben:
  - in natürlicher Sprache (missverständlich)
  - als Diagramm (anschaulich)
  - in Pseudocode
  - als Programm in einer Programmiersprache
- Programmieren bedeutet, die Einzelschritte eines Algorithmus in eine Programmiersprache zu übersetzen, die von einem Computer ausgeführt werden kann.

- Die Programmiersprache Python wurde von Guido van Rossum zu Beginn der 1990er-Jahre als Nachfolger der Programmiersprache ABC entwickelt. Der Name „Python“ geht auf die britische Komikergruppe „Monty Python“ zurück. Derzeit (im Jahr 2021) ist die Version 3.9 aktuell und kann für viele verschiedene Betriebssysteme kostenlos von `www.python.org` heruntergeladen werden.
- Der Python-Interpreter übersetzt die von Menschen lesbaren Anweisungen in Maschinencode, der wiederum vom Prozessor des Computers ausgeführt wird.
- Die Standardversion von Python stellt die Entwicklungsumgebung `IDLE` zur Verfügung. Damit können Programme geschrieben, ausgeführt und Fehler im Code aufgespürt werden.

Ein skalierbares Rezept

Wir entwickeln ein Programm, das uns das Kochrezept für eine bestimmte Anzahl von Personen berechnet und am Bildschirm ausgibt.

Dies soll am Beispiel eines Rezepts für mexikanische Fladenbrote (Tortillas) erfolgen.

Das Tortilla-Rezept für eine Person

Mehl:	40 g
Öl:	6 g
Salz:	1 Prise
Wasser:	18 g

Mehl, Wasser, Salz und Öl in eine Schüssel geben und mit den Händen mindestens 5 Minuten zu einem glatten Teig verkneten.

Den Teig in Frischhaltefolie wickeln und 20 Minuten in den Kühlschrank stellen.

Den Teig aus dem Kühlschrank nehmen, nochmals gut durchkneten und Teigstücke auf einer bemehlten Fläche zu Fladen von maximal 1 mm Dicke ausrollen.

Die Fladen in einer beschichteten Bratpfanne ohne Öl auf beiden Seiten kurz backen.

User Stories

Auch wenn dieses Programm nicht sehr kompliziert wird, verwenden wir ein Vorgehen, das auch bei grösseren Software-Projekten angewendet wird.

Dazu schreiben wir jeweils eine Anforderung aus der Sicht des Benutzers (die User Story) in wenigen Sätzen auf eine Karteikarte oder einen Klebezettel (die Story-Card).

Die User Story kann formlos oder unter Verwendung einer Schablone angelegt werden. Dabei ist es wichtig anzugeben, aus wessen Sicht und mit welchem Ziel und welchem Nutzen die User Story geschrieben wird.

User Stories für den Tortilla-Rechner

1. Als Benutzer möchte ich die Anzahl Personen eingeben, für die das Rezept berechnet wird. Das Programm soll diese Anzahl speichern.

2. Als Benutzer möchte ich, dass die Mengen der Zutaten aufgrund der gespeicherten Personenzahl berechnet wird.

3. Als Benutzer möchte ich, dass der Name des Rezepts (mit der Personenzahl), die Zutaten und eine Anleitung für die Zubereitung ausgegeben wird.

Schritt 1

Python verarbeitet Benutzereingaben mit der `input`-Funktion. Schreibe diese Anweisung in die Datei `tortillas.py`:

```
anzahl = input('Wie viele Personen? ')
```

Syntax: Unmittelbar nach dem Schlüsselwort `input` folgt ein Paar runder Klammern, zwischen denen nichts oder eine Zeichenkette steht. Zeichenketten müssen durch ein Paar einfacher oder doppelter Anführungszeichen begrenzt werden.

Semantik: Bei einer Zuweisung mit = wird zuerst der Code rechts – also die `input`-Funktion ausgeführt. Diese gibt den Text zwischen den runden Klammern auf der Shell aus und wartet, bis der Benutzer eine Eingabe gemacht und mit der ENTER-Taste abgeschlossen hat. Danach wird die Benutzereingabe als Rückgabewert an die Stelle der `input`-Anweisung gesetzt und schliesslich der Variable `anzahl` zugewiesen.

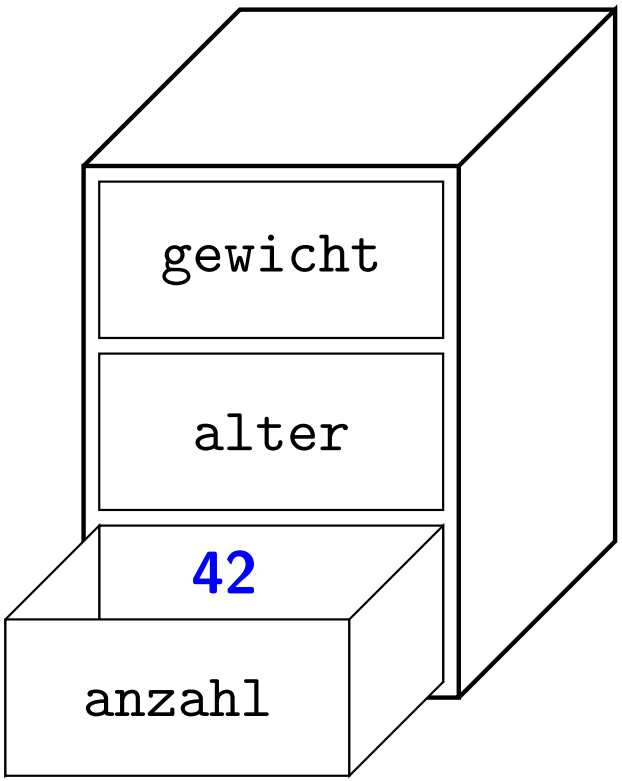
Übung 2.1

Mach dich mit der Funktionsweise der `input`-Funktion vertraut, indem du in der Python-Shell die folgenden Eingaben machst und jeweils mit der ENTER-Taste abschliesst:

- `name = input('Wie heisst du? ')`
- Gib (d)einen Namen ein
- `print(name)`

Variablen

Eine Variable besteht aus einem Bezeichner (dem Namen der Variablen) und einem Wert. Diese Informationen werden von Python in einer internen Tabelle gespeichert.



Eine Variable ist vergleichbar mit einer Schublade, die einen Wert enthält, und die mit dem Bezeichner angeschrieben ist.

Zuweisungen

Der Zuweisungsoperator (=) hat folgende Semantik:

1. Zuerst wird der Ausdruck rechts des Zuweisungsoperators ausgewertet, bis der Wert feststeht.
2. Dieser Wert wird der Variablen mit dem Bezeichner links vom Zuweisungsoperator zugewiesen.

Beispiel: `mittelwert` = $\underbrace{\texttt{(4 + 3.5 + 6) / 3}}_{\texttt{4.5}}$

Achtung: Anders als in der Algebra, wo $x = 5$ dasselbe bedeutet wie $5 = x$, muss bei einer Zuweisung der Bezeichner links und der Wert rechts stehen.

Variablen sind veränderlich

Der Wert einer Variablen kann sich im Laufe eines Programms verändern. Das kann bei einem Spiel sinnvoll sein, das den Punktestand einer Spielerin speichert.

```
1 score = 0
2 score = score + 100
3 score = score + 500
4 print(score) # Ausgabe?
5 score = score - 200
6 score = score + 100
7 print(score) # Ausgabe?
```

Nicht mehr benötigte Variablen müssen nicht gelöscht werden. Python erkennt solche Variablen und gibt ihren Speicherplatz automatisch frei. Wer dennoch selber „aufräumen“ will, der kann z. B. die Variable `score` mit der Anweisung `del score` löschen.

Übung 2.2

Welche Ausgaben macht das folgende Programm?

```
1 summe = 3 + 5 + 7 + 5
2 print(summe)
3 alter = 15
4 print(alter)
5 alter = alter + 1
6 print(alter)
7 summe = alter + summe
8 print(summe)
```

Regeln für gültige Bezeichner

- Ein Bezeichner muss mit einem Buchstaben oder einem Unterstrich beginnen. Danach dürfen Buchstaben, Ziffern und Unterstriche folgen. Beachte, dass Python bei Bezeichnern die Gross- und Kleinschreibung unterscheidet.
- Ein Bezeichner darf kein Python-Schlüsselwort sein:
  `and`, `as`, `assert`, `async`, `await`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `try`, `while`, `with`, `yield`, `False`, `None`, `True`
- Zwar können die Namen von Python-Funktionen wie `input` oder `print` theoretisch als Bezeichner verwendet werden. Davon ist aber abzuraten, da eine solche Zuweisung die entsprechende Funktion unbrauchbar macht.

Übung 2.3

Welche der Bezeichner sind gültig?

(a) `klasse3a`

(b) `klasse-3a`

(c) `klasse_3a`

(d) `3aKlasse`

(e) `class`

(f) `class3a`

Operatoren für Zahlen

Operator	Operation	Beispiel
+	Addition	`5 + 3 = 8`
-	Subtraktion	`7 - 2 = 5`
*	Multiplikation	`2 * 3 = 6`
/	Division	`7 / 2 = 3.5`
//	ganzzahlige Division	`7 // 2 = 3`
%	Divisionsrest	`7 % 2 = 1`
**	Potenzieren	`2 ** 3 = 8`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2`

(b) `18 // 5`

(c) `37 % 10`

(d) `2 ** 5`

(e) `5 // 9`

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5`

(c) `37 % 10`

(d) `2 ** 5`

(e) `5 // 9`

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10`

(d) `2 ** 5`

(e) `5 // 9`

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5`

(e) `5 // 9`

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5` 32

(e) `5 // 9`

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5` 32

(e) `5 // 9` 0

(f) `543 % 2`

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5` 32

(e) `5 // 9` 0

(f) `543 % 2` 1

(g) `10 / 4`

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5` 32

(e) `5 // 9` 0

(f) `543 % 2` 1

(g) `10 / 4` 2.5

(h) `2 % 543`

Übung 2.4

Bestimme den Wert des Ausdrucks.

(a) `3 ** 2` 9

(b) `18 // 5` 3

(c) `37 % 10` 7

(d) `2 ** 5` 32

(e) `5 // 9` 0

(f) `543 % 2` 1

(g) `10 / 4` 2.5

(h) `2 % 543` 2

Präzedenz (Vorrangstellung) der Operatoren:

Höchste	`**`	Potenzieren
	`-`	einstelliges Minus (Vorzeichen)
	`* / // %`	Multiplikation, Divisionen, Rest
Niedrigste	`+ -`	zweistelliges Plus und Minus

Innerhalb der gleichen Stufe wird von links nach rechts gerechnet.
Mit runden Klammern lässt sich die Auswertungsreihenfolge selber festlegen.

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2`

(b) `3 / (1 + 2) * 3 ** 2`

(c) `4 * 7 // 5 + 1`

(d) `4 * 7 // (5 + 1)`

(e) `2 + 12 / 4 + 3`

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2`

(c) `4 * 7 // 5 + 1`

(d) `4 * 7 // (5 + 1)`

(e) `2 + 12 / 4 + 3`

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1`

(d) `4 * 7 // (5 + 1)`

(e) `2 + 12 / 4 + 3`

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)`

(e) `2 + 12 / 4 + 3`

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)` 4

(e) `2 + 12 / 4 + 3`

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)` 4

(e) `2 + 12 / 4 + 3` 8.0

(f) `(2 + 12) / (4 + 3)`

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)` 4

(e) `2 + 12 / 4 + 3` 8.0

(f) `(2 + 12) / (4 + 3)` 2.0

(g) `- 5 ** 2`

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)` 4

(e) `2 + 12 / 4 + 3` 8.0

(f) `(2 + 12) / (4 + 3)` 2.0

(g) `- 5 ** 2` -25

(h) `(- 5) ** 2`

Übung 2.5

Bestimme den Wert des Ausdrucks.

(a) `3 / 1 + 2 * 3 ** 2` 21

(b) `3 / (1 + 2) * 3 ** 2` 9

(c) `4 * 7 // 5 + 1` 6

(d) `4 * 7 // (5 + 1)` 4

(e) `2 + 12 / 4 + 3` 8.0

(f) `(2 + 12) / (4 + 3)` 2.0

(g) `- 5 ** 2` -25

(h) `(- 5) ** 2` 25

Operatoren für Zeichenketten

Operator	Operation	Beispiel
+	Addition	`'H' + 'und'` $\rightarrow$ `'Hund'`
*	Multiplikation	`3 * 'ha'` $\rightarrow$ `'hahaha'`

Übung 2.6

Bestimme den Wert des Ausdrucks.

(a) `'a' + 2 * 'n' + 'a'`

(b) `'a' + 2 * 'na' + 's'`

(c) `10 * '-'`

Übung 2.6

Bestimme den Wert des Ausdrucks.

(a) `'a' + 2 * 'n' + 'a'` anna

(b) `'a' + 2 * 'na' + 's'`

(c) `10 * '-'`

Übung 2.6

Bestimme den Wert des Ausdrucks.

(a) `'a' + 2 * 'n' + 'a'` anna

(b) `'a' + 2 * 'na' + 's'` ananas

(c) `10 * '-'`

Übung 2.6

Bestimme den Wert des Ausdrucks.

(a) `'a' + 2 * 'n' + 'a'` anna

(b) `'a' + 2 * 'na' + 's'` ananas

(c) `10 * '-'` ----------

Schritt 2

Wir können nun die nächste Anforderung an unser Programm erfüllen. Ergänze das Programm `tortillas.py` um die Zeilen 3–6.

```
1 anzahl = input('Wie viele Personen? ')
2
3 mehl = anzahl * 40    # Gramm
4 oel = anzahl * 6      # Gramm
5 salz = anzahl * 1     # Prisen
6 wasser = anzahl * 18 # Gramm
```

Übung 2.7

(a) Welchen Wert sollte die Variable `oel` nach der Eingabe von 4 (Personen) haben?

(b) Lass das Programm laufen und gib dann in der Shell die Anweisung `print(oel)` ein. Welcher Wert wird ausgegeben?

(c) Wie erklärst du dir den Unterschied zwischen der erwarteten Ausgabe in (a) und der tatsächlichen in (b)?

Übung 2.7

(a) Welchen Wert sollte die Variable `oel` nach der Eingabe von 4 (Personen) haben?
24

(b) Lass das Programm laufen und gib dann in der Shell die Anweisung `print(oel)` ein. Welcher Wert wird ausgegeben?

(c) Wie erklärst du dir den Unterschied zwischen der erwarteten Ausgabe in (a) und der tatsächlichen in (b)?

Übung 2.7

(a) Welchen Wert sollte die Variable `oel` nach der Eingabe von 4 (Personen) haben?
24

(b) Lass das Programm laufen und gib dann in der Shell die Anweisung `print(oel)` ein. Welcher Wert wird ausgegeben?
444444

(c) Wie erklärst du dir den Unterschied zwischen der erwarteten Ausgabe in (a) und der tatsächlichen in (b)?

Übung 2.7

(a) Welchen Wert sollte die Variable `oel` nach der Eingabe von 4 (Personen) haben?
24

(b) Lass das Programm laufen und gib dann in der Shell die Anweisung `print(oel)` ein. Welcher Wert wird ausgegeben?
444444

(c) Wie erklärst du dir den Unterschied zwischen der erwarteten Ausgabe in (a) und der tatsächlichen in (b)?
Da die Input-Anweisung die Zeichenkette `'4'` zurückgibt, erhalten wir `'4' * 6 = '444444'`.

Fehler beim Programmieren

- Syntaxfehler entstehen, wenn man die „Grammatik“ von Python verletzt. Beispielsweise wenn man Variablen falsch schreibt oder Anführungszeichen nicht wieder schliesst. Python gibt die Zeilennummer und den Typ des Fehlers in einer Fehlermeldung auf der Shell aus.
- Laufzeitfehler sind Fehler, die erst beim Ausführen des Programms auftreten. Etwa dann, wenn eine Zahl durch Null dividiert wird oder eine nicht existierende Datei geöffnet werden soll. Auch hier gibt Python die Zeilennummer und den Typ des Fehlers in einer Fehlermeldung aus.
- Wenn ein Programm weder Syntax- noch Laufzeitfehler enthält, aber nicht das geforderte Ergebnis berechnet, sprechen wir von semantischen oder logischen Fehlern. Diese müssen wir mit Programmtests suchen, da keine Fehlermeldung erscheint.

Übung 2.8

Welche Art von Fehler haben wir im Programmcode von Schritt 2 gemacht?

Übung 2.8

Welche Art von Fehler haben wir im Programmcode von Schritt 2 gemacht?

Da das Programm ohne Fehlermeldung ausgeführt wurde aber nicht das geforderte Resultat berechnet, handelt es sich um einen semantischen (oder logischen) Fehler.

Datentypen

Bisher haben wir mit Python verschiedene Arten von Werten mit unterschiedlichen Operationen verknüpft, aber uns noch nicht genau damit befasst, von welchem Typ das Resultat ist. Hier ein paar Beispiele:

Operand 1	Operator	Operand 2	Resultat
-5	*	7	-35
7	/	2	3.5
7	//	2	3
'ab'	+	'cd'	'abcd'
3	*	'x'	'xxx'
'x'	*	3	'xxx'
7.0	-	3	4.0
7	**	2	49
7	**	2.0	49.0

In den obigen Beispielen kommen drei Datentypen vor.

- Ganze Zahl (integer): ein optionales Vorzeichen, gefolgt von mindestens einer Ziffer
- Gleitkommazahl (float:) ein optionales Vorzeichen, gefolgt von einer Ziffernfolge, die einen Dezimalpunkt enthält.
- Zeichenkette (string): eine Folge von Zeichen, die in einfachen oder doppelten Anführungszeichen eingeschlossen ist.

Typumwandlung

Da die `input`-Anweisung in unserem Tortilla-Programm immer eine Zeichenkette zurückgibt, auch wenn wir eine Zahl eingegeben haben, müssen wir dafür sorgen, dass diese Eingabe auch wieder in eine Zahl umgewandelt wird, damit die Mengenangaben mit einer Zahl und nicht mit einer Zeichenkette multipliziert werden. Python stellt uns dafür die Funktion

```
int(x)
```

zur Verfügung, die das Argument `x` (so gut es geht) in eine ganze Zahl umwandelt. *Beispiele:*

Ausdruck	Wert
`int('5')`	5
`int('abc')`	Fehler
`int(4.738)`	4 (!)
`int(-17)`	-17

Schritt 2 (verbessert)

```
1  anzahl = input('Wie viele Personen? ')
2
3  anzahl = int(anzahl)
4
5  mehl = anzahl * 40    # Gramm
6  oel = anzahl * 6      # Gramm
7  salz = anzahl * 1     # Prisen
8  wasser = anzahl * 18 # Gramm
```

Schritt 3

Nun müssen wir noch die berechneten Mengen und die Zubereitungsanleitung ausgeben.

Statt jede Zeile einzeln mit `print(...)` auszugeben, erstellen wir eine Textschablone, in die wir anschliessend die berechneten Mengen einfügen. Dazu verwenden wir triple quotes: Das ist ein Paar von drei einfachen (oder doppelten) Anführungszeichen, zwischen denen wir einen Text (Zeichenkette) einschliessen, die genau so ausgegeben werden soll.

Das fertige Programm

siehe Handout

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

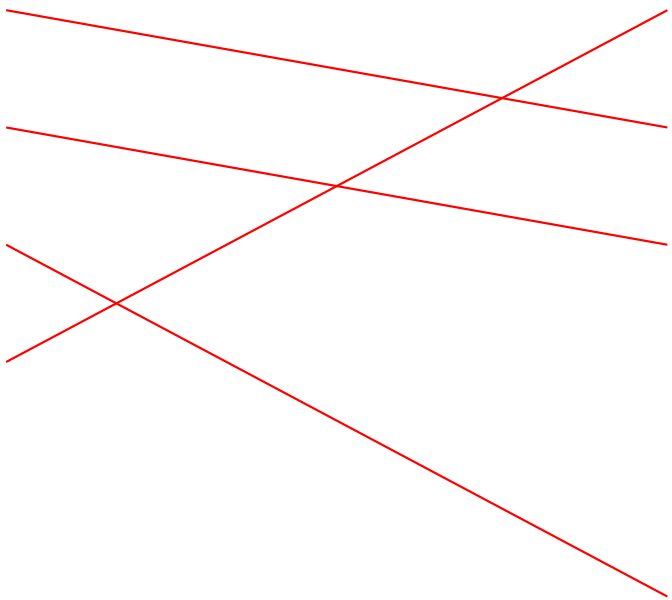
Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

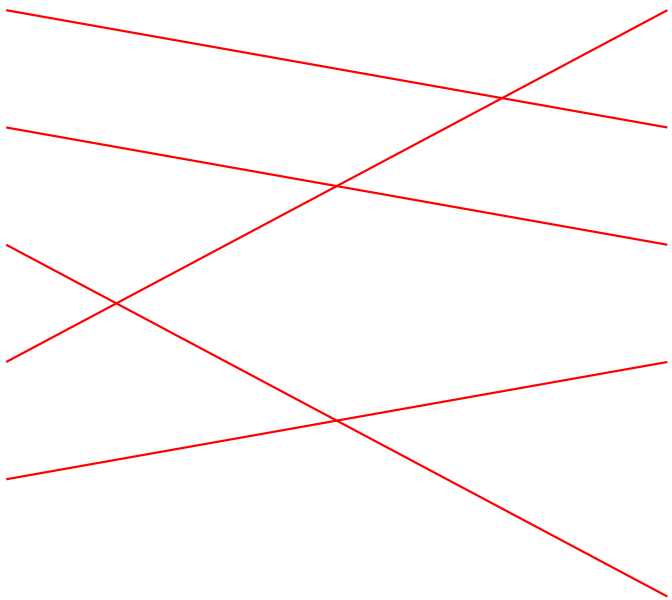
Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`



Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`



Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.9

Verbinde den Ausdruck links mit dem passenden Wert rechts.

Ausdruck	**Wert**
`18 - 6 * 2`	`'24'`
`17 % 7`	`6`
`- 4 ** 2`	`3`
`'2' + '4'`	`16`
`(- 2) ** 4`	`'22'`
`20 // 5`	`-16`
`2 * '2'`	`4.0`
`12 / 3`	`4`

Übung 2.10

Welches Passwort wird am Ende des Programms ausgegeben?

```
1  teil1 = 'eh'; teil2 = 'en'; teil3 = 'ch'
2  teil4 = 'es'; teil5 = 'ha'; teil6 = 'st'
3  teil7 = 'tg'; teil8 = 'du'; teil9 = 'ni'
4  passwort = teil3
5  passwort = passwort + teil7
6  passwort = passwort + teil4
7  passwort = teil9 + passwort
8  passwort = teil8 + passwort
9  passwort = passwort + teil1
10 passwort = teil6 + passwort
11 passwort = passwort + teil2
12 passwort = teil5 + passwort
13 print(passwort)
```

Um Platz zu sparen, wurden hier Anweisungen durch Strichpunkte getrennt.

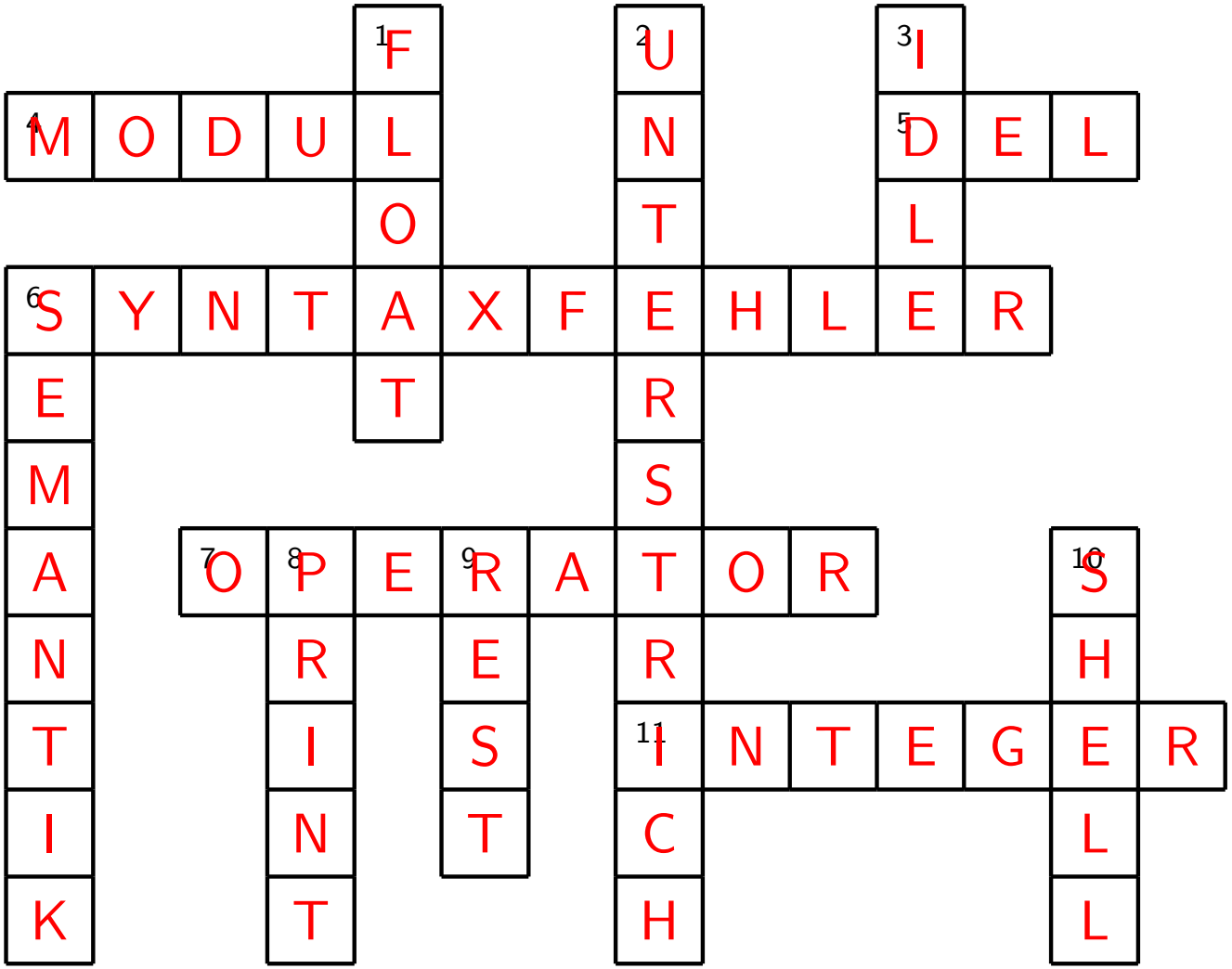
Übung 2.11

Waagrecht

4. Anderer Name für ein Python-Programm
5. Anweisung zum Löschen einer Variablen
6. Fehlerart beim Programmieren
7. Steht zwischen Operanden
11. Datentyp

Senkrecht

1. Datentyp
2. Erlaubtes Sonderzeichen in Bezeichnern
3. Name einer Python-Entwicklungsumgebung
6. Bedeutung
8. Python-Funktion zur Ausgabe auf der Shell
9. Wird von der Modulo-Funktion berechnet
10. Interaktiver Python-Interpreter


1
F
2
U
3
I
4
M O D U L
N
5
D E L
O
T
L
6
S Y N T A X F E H L E R
E
T
R
M
S
A
7
O
8
P E
9
R A T O R
10
S
N
R
E
R
H
T
I
S
11
I N T E G E R
I
N
T
C
L
K
T
H
L

Auf den Punkt gebracht

- Computer sind gut darin, Werte zu speichern und damit zu rechnen.
- User Stories helfen dabei, eine Software so zu programmieren, dass sie die Anforderungen ihrer Nutzer erfüllen.
- Die Syntax legt die Form der Python-Anweisungen fest.
- Die Semantik definiert die Bedeutung der Python-Anweisungen.
- Die `input`-Funktion gibt die Zeichenkette im Argument aus und fordert den Benutzer zu einer Eingabe auf. Diese wird eingelesen und als Zeichenkette zurückgegeben.
- Variablen bestehen aus einem Bezeichner (Namen) und einem Wert, der im Arbeitsspeicher abgelegt wird.

- Eine Variable wird erzeugt, indem man mit dem Zuweisungsoperator (=) einen Wert (rechts) einem Bezeichner (links) zuweist. Der Wert kann auch in Form eines Ausdrucks gegeben sein.
- Einer bereits vorhandenen Variable können im Verlauf eines Programms neue Werte zugewiesen werden.
- Tritt in einer Anweisung ein Bezeichner auf, dem bereits ein Wert zugewiesen wurde, dann weist Python diesem Bezeichner den aktuellen Wert zu.
- Variablen können mit dem Operator `del <Bezeichner>` gelöscht werden. Dies ist aber überflüssig, da Python nicht mehr benötigte Variablen erkennt und sie automatisch löscht.
- Für die Bildung von Bezeichnern gibt es wenige, einfache Regeln. Es dürfen nur Buchstaben, Ziffern und Unterstriche (_) darin vorkommen und Ziffern an erster Stelle sind verboten. Darüber hinaus sind Schlüsselwörter als Bezeichner ungültig.

- Eine Zeichenkette (String) ist ein Python-Datentyp, der aus einer (möglicherweise leeren) Folge von Zeichen besteht. Bei der Eingabe von Strings ist darauf zu achten, dass sie entweder von einfachen oder von doppelten Anführungszeichen eingeschlossen werden.
- Eine ganze Zahl (Integer) ist ein Python-Datentyp, der aus einem allfälligen Vorzeichen und einer Folge von Ziffern besteht.
- Eine Gleitkommazahl (Float) ist ein Python-Datentyp, der aus einem allfälligen Vorzeichen und einer Folge von Ziffern besteht, die einen Dezimalpunkt enthält.
- Die `int`-Funktion wandelt Zeichenketten, die aus lauter Ziffern bestehen, in eine ganze Zahl um. Es lassen sich aber noch mehr Typumwandlungen (*casts*) damit durchführen.
- Programmierer unterscheiden zwischen Syntax-, Laufzeit- und Semantikfehlern.

- Semantikfehler verursachen keine Fehlermeldungen. Um sie zu finden, muss der Code systematisch getestet werden.
- Für die Verarbeitung der Datentypen Integer, Float und String gibt es verschiedene Operatoren und Funktionen. Um semantische Fehler zu vermeiden, sollte man sich immer im Klaren sein, welchen Datentyp der Wert eines Ausdrucks hat.
- Werte, die an Funktionen übergeben werden, heissen Argumente und stehen, durch Kommas getrennt und von runden Klammern eingeschlossen, unmittelbar nach dem Namen der Funktion.

Ausblick

In dieser Lektion geht es darum, das Spiel „Schere, Stein, Papier“ zu programmieren, wobei der Computer die Rolle des Gegners einnimmt.

Daher müssen wir uns mit einigen zentralen Konzepten der Programmierung befassen:

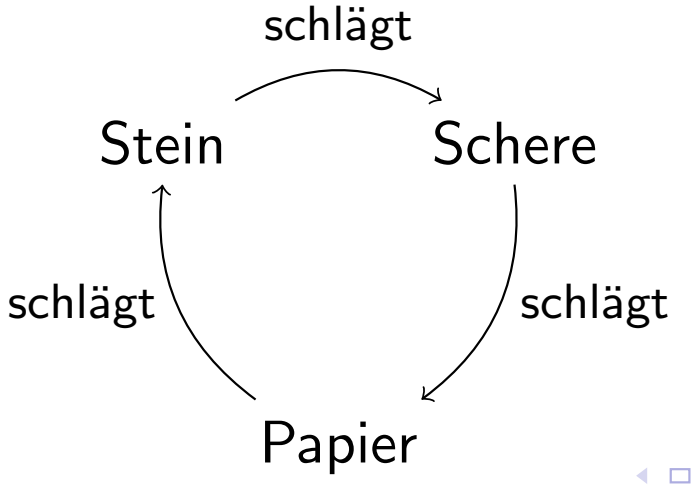
- Viele Spiele werden erst durch einen Zufallsmechanismus spannend.
- Der Spielverlauf hängt davon ab, welche Entscheidungen die Spieler treffen.
- Bestimmte Schritte müssen wiederholt ausgeführt werden.

Die Spielregeln

Zwei Spieler entscheiden sich jeweils insgeheim für eines der drei Objekte *Schere*, *Stein* und *Papier*.

Danach zeigen sie einander gleichzeitig ihre Wahl durch eine Handfigur, die das Objekt symbolisiert.

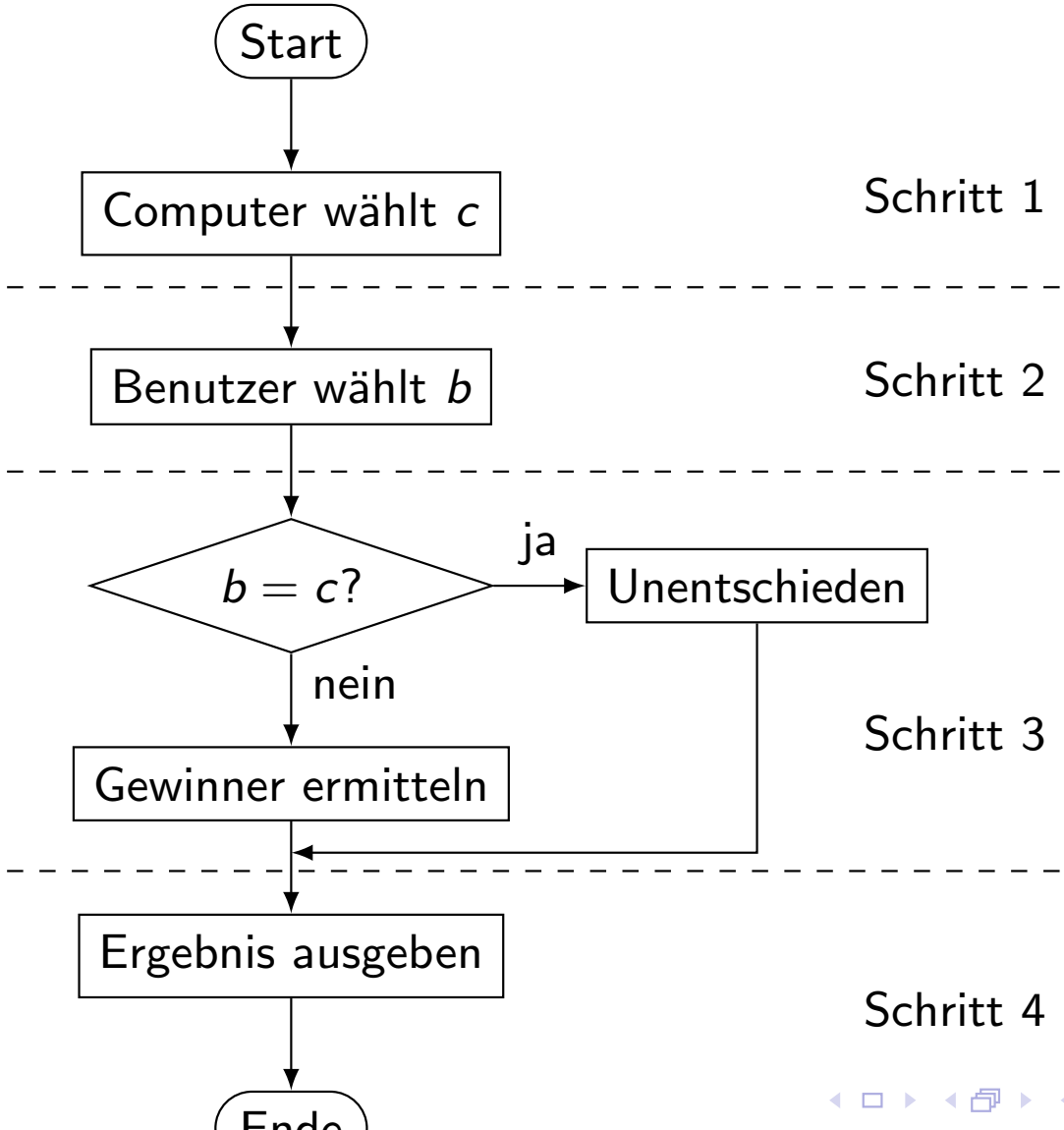
Wenn beide Spieler dasselbe Objekt anzeigen, dann endet die Runde unentschieden. Andernfalls wird der Gewinner wie folgt ermittelt:



User Stories

1. Als Benutzer möchte ich, dass sich mein Gegner (der Computer) unberechenbar verhält.
2. Als Benutzer möchte ich, dass mich der Computer so lange um eine Eingabe bittet, bis sie korrekt ist. Dabei sollen Gross- und Kleinschreibung akzeptiert werden.
3. Als Benutzer möchte ich, dass der Computer die Spielregeln korrekt anwendet, um das Resultat zu ermitteln.
4. Als Benutzer möchte ich, dass am Spielende die Wahl des Computers und das Ergebnis ausgegeben wird.

Der Spielverlauf als Flussdiagramm



Python-Listen

Listen werden in Python durch eckige Klammern (`[...]`) begrenzt, wobei benachbarte Elemente jeweils durch ein Komma separiert sein müssen. Ist ein Element vom Datentyp String, muss es von einfachen oder doppelten Anführungszeichen umgeben sein. Natürlich können Listen auch Variablen enthalten.

Wie wir später sehen werden, sind Listen in Python *dynamisch*; d. h. es können zur Laufzeit Elemente hinzugefügt oder entfernt werden. Darüber hinaus können die Elemente von Python-Listen unterschiedliche Datentypen haben oder selber Listen sein.

Beispiel: `mylist = ['Hallo', 42, "What?", 3.141592]`

Ein Element zufällig aus einer Liste auswählen

Das Python-Package `random` enthält verschiedene Methoden um Zufall zu simulieren. Eine davon wählt ein zufälliges Element aus einer Liste aus und liefert es als Wert zurück. Dazu muss man zunächst das `random`-Modul importieren:

```
import random
```

Ist eine Liste wie

```
wuerfel = [1, 2, 3, 4, 5, 6]
```

gegeben, so lässt sich mit der Anweisung

```
augenzahl = random.choice(wuerfel)
```

ein zufälliges Element auswählen. Beispielsweise könnte dann

```
print(augenzahl)
```

den Wert 2 ausgeben.

Schritt 1

Nun können wir mit der Programmierung unseres Spiels beginnen. Öffne eine leere Datei und speichere sie unter dem Namen `ssp.py` (Schere, Stein, Papier) ab.

Der folgende Code erfüllt die Anforderungen der ersten User Story:

```
1  import random
2
3  objekte = ['Schere', 'Stein', 'Papier']
4  wahl_comp = random.choice(objekte)
5
6  print(wahl_comp) # zu Testzwecken
```

Übung 3.1

Führe den Code von `ssp.py` 15-Mal aus. Wie oft wählt bei dir der Computer `'Schere'`, `'Stein'` und `'Papier'`?

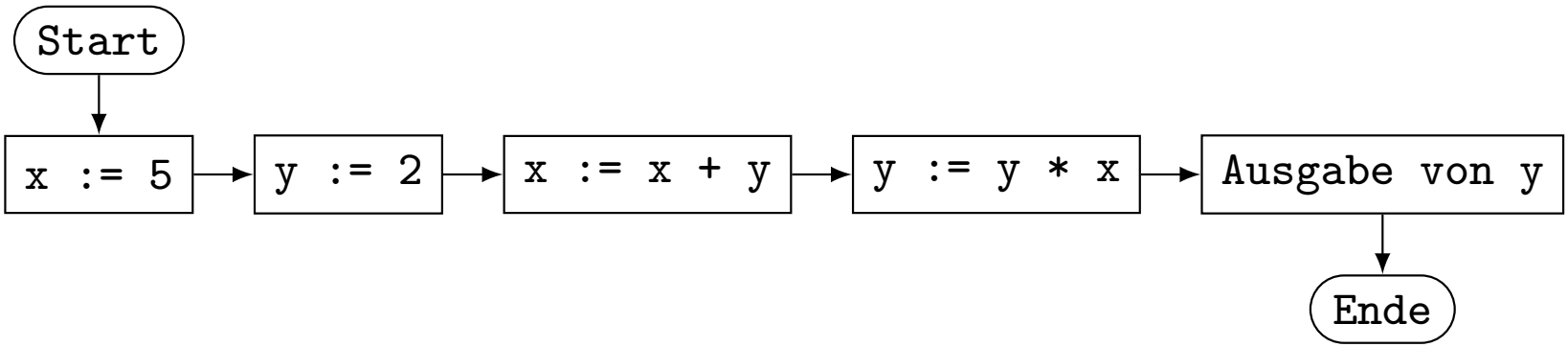
Übung 3.1

Führe den Code von `ssp.py` 15-Mal aus. Wie oft wählt bei dir der Computer `'Schere'`, `'Stein'` und `'Papier'`?

	`Schere`	`Stein`	`Papier`
Anzahl	6	4	5

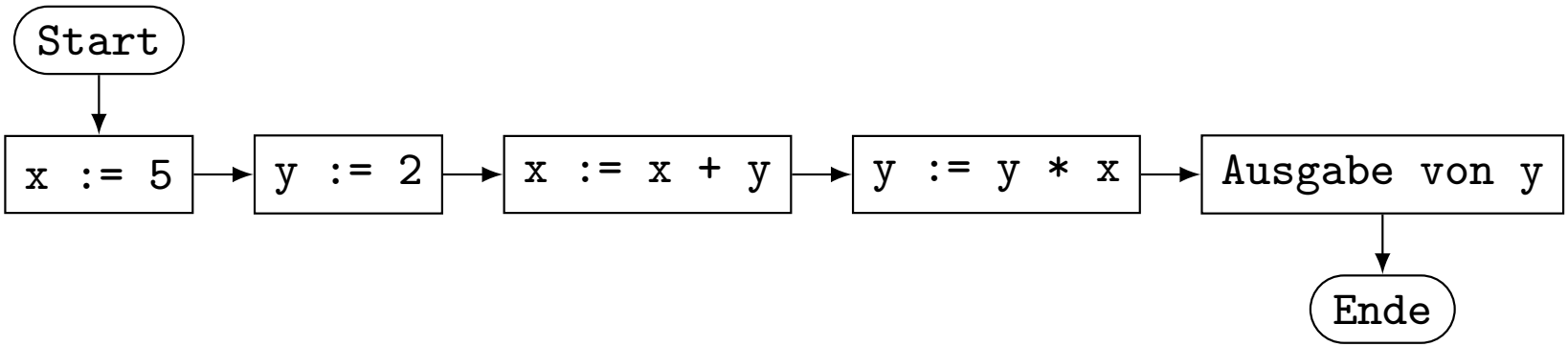
Übung 3.2

Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Übung 3.2

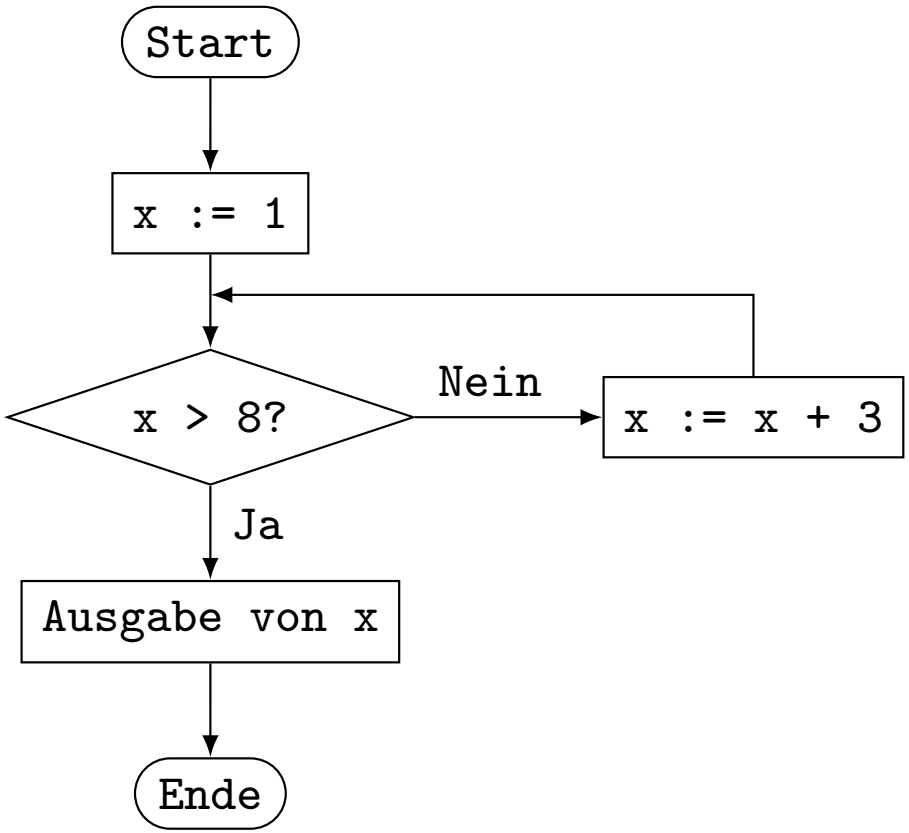
Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Ausgabe: 14

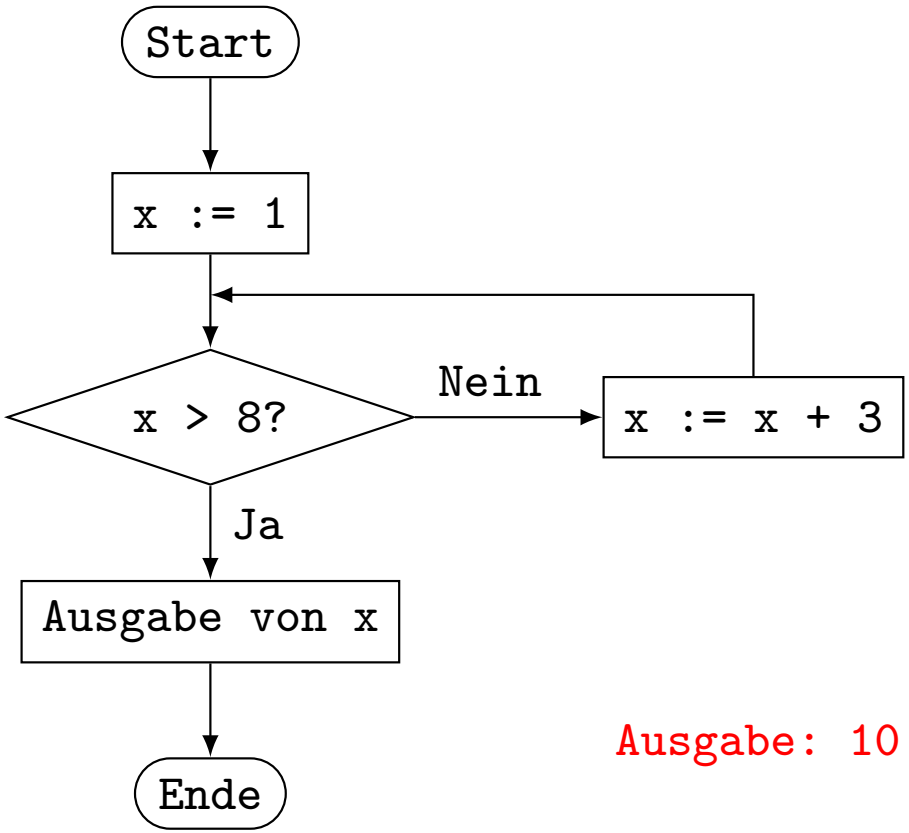
Übung 3.3

Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Übung 3.3

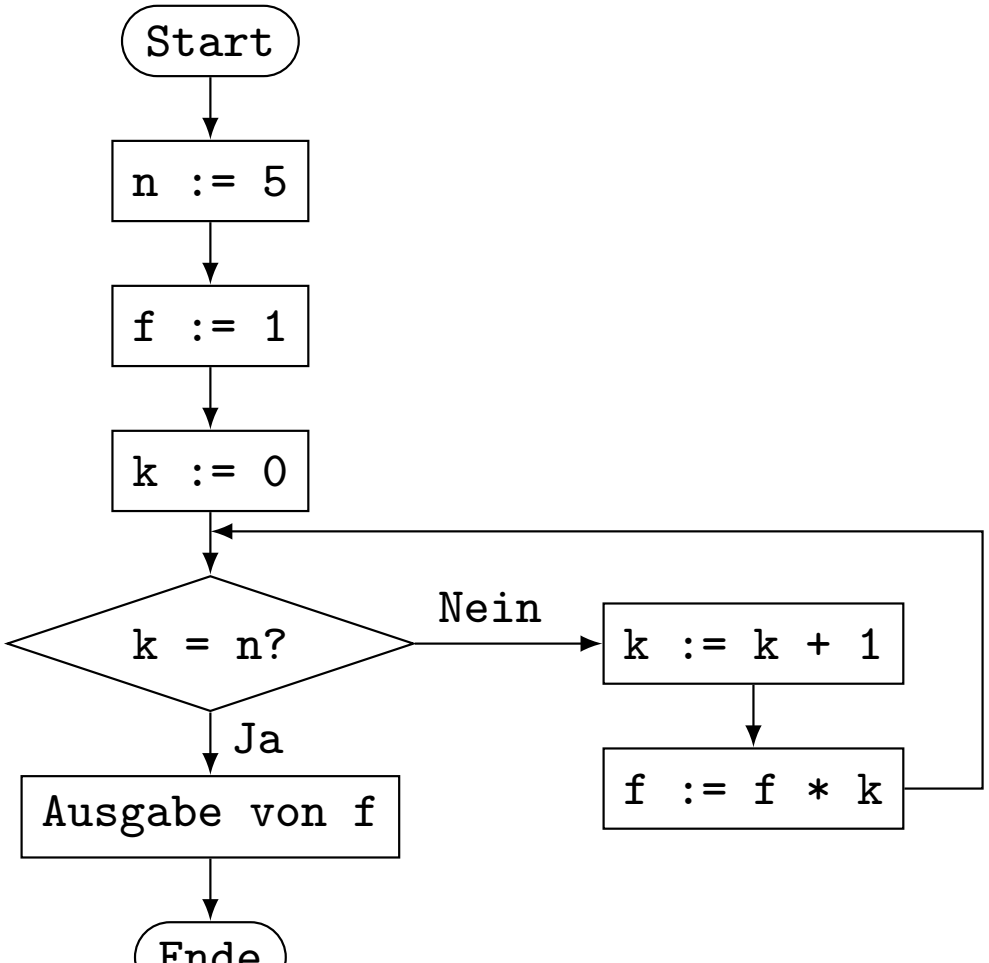
Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Ausgabe: 10

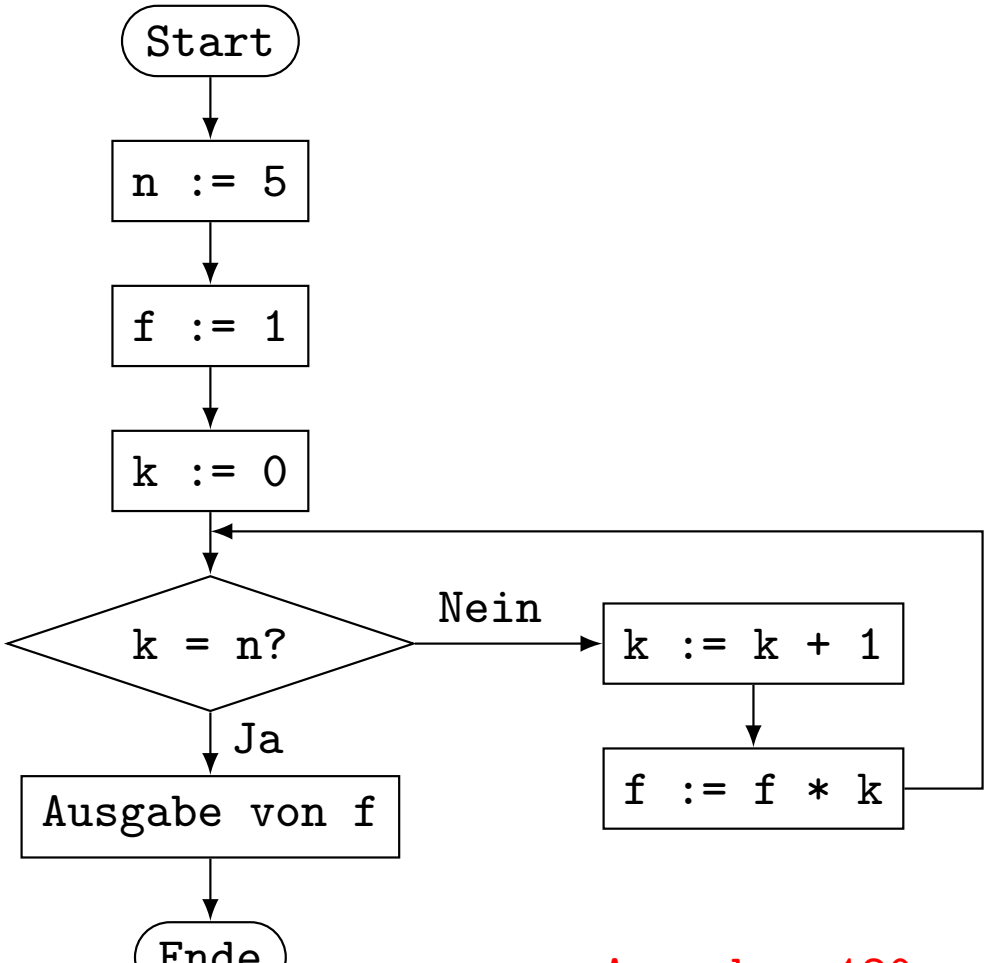
Übung 3.4

Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Übung 3.4

Welche Ausgabe(n) macht ein Programm, das folgendes Flussdiagramm implementiert?



Tests auf Enthaltensein in einer Liste

Mit den Operatoren `"in"` und `"not in"` können wir überprüfen, ob ein Wert in einer Liste ist oder nicht.

Der Ausdruck `<wert> in <liste>` hat den Wert `True`, wenn `<wert>` *in* der `<liste>` enthalten ist und sonst `False`.

Der Ausdruck `<wert> not in <liste>` hat den Wert `True`, wenn `<wert>` *nicht in* der `<liste>` enthalten ist und sonst `False`.

Beispiele:

Ausdruck	Wert
`3 in [1, 2, 3, 4, 5]`	
`3 not in [1, 2, 3, 4, 5]`	
`'Stein' in ['Schere', 'Stein', 'Papier']`	
`'Jein' not in ['Ja', 'Nein']`	

Tests auf Enthaltensein in einer Liste

Mit den Operatoren `"in"` und `"not in"` können wir überprüfen, ob ein Wert in einer Liste ist oder nicht.

Der Ausdruck `<wert> in <liste>` hat den Wert `True`, wenn `<wert>` *in* der `<liste>` enthalten ist und sonst `False`.

Der Ausdruck `<wert> not in <liste>` hat den Wert `True`, wenn `<wert>` *nicht in* der `<liste>` enthalten ist und sonst `False`.

Beispiele:

Ausdruck	Wert
`3 in [1, 2, 3, 4, 5]`	True
`3 not in [1, 2, 3, 4, 5]`	
`'Stein' in ['Schere', 'Stein', 'Papier']`	
`'Jein' not in ['Ja', 'Nein']`	

Tests auf Enthaltensein in einer Liste

Mit den Operatoren `"in"` und `"not in"` können wir überprüfen, ob ein Wert in einer Liste ist oder nicht.

Der Ausdruck `<wert> in <liste>` hat den Wert `True`, wenn `<wert>` *in* der `<liste>` enthalten ist und sonst `False`.

Der Ausdruck `<wert> not in <liste>` hat den Wert `True`, wenn `<wert>` *nicht in* der `<liste>` enthalten ist und sonst `False`.

Beispiele:

Ausdruck	Wert
`3 in [1, 2, 3, 4, 5]`	True
`3 not in [1, 2, 3, 4, 5]`	False
`'Stein' in ['Schere', 'Stein', 'Papier']`	
`'Jein' not in ['Ja', 'Nein']`	

Tests auf Enthaltensein in einer Liste

Mit den Operatoren `"in"` und `"not in"` können wir überprüfen, ob ein Wert in einer Liste ist oder nicht.

Der Ausdruck `<wert> in <liste>` hat den Wert `True`, wenn `<wert>` *in* der `<liste>` enthalten ist und sonst `False`.

Der Ausdruck `<wert> not in <liste>` hat den Wert `True`, wenn `<wert>` *nicht in* der `<liste>` enthalten ist und sonst `False`.

Beispiele:

Ausdruck	Wert
`3 in [1, 2, 3, 4, 5]`	True
`3 not in [1, 2, 3, 4, 5]`	False
`'Stein' in ['Schere', 'Stein', 'Papier']`	True
`'Jein' not in ['Ja', 'Nein']`	

Tests auf Enthaltensein in einer Liste

Mit den Operatoren `"in"` und `"not in"` können wir überprüfen, ob ein Wert in einer Liste ist oder nicht.

Der Ausdruck `<wert> in <liste>` hat den Wert `True`, wenn `<wert>` *in* der `<liste>` enthalten ist und sonst `False`.

Der Ausdruck `<wert> not in <liste>` hat den Wert `True`, wenn `<wert>` *nicht in* der `<liste>` enthalten ist und sonst `False`.

Beispiele:

Ausdruck	Wert
`3 in [1, 2, 3, 4, 5]`	True
`3 not in [1, 2, 3, 4, 5]`	False
`'Stein' in ['Schere', 'Stein', 'Papier']`	True
`'Jein' not in ['Ja', 'Nein']`	True

Wahrheitswerte

Die Werte `True` und `False` bilden zusammen mit den logischen Operationen, die wir später noch genauer kennen lernen werden, den Booleschen Datentyp, der nach ihrem Erfinder George Boole (1815–1864) benannt ist.

Eingaben mit einer `while`-Schleife prüfen

Wenn wir die erlaubten Eingaben in einer Liste zusammenfassen, können wir einen Benutzer mit einer `while`-Schleife so lange um eine Eingabe bitten, bis sie korrekt ist.

```
1 eingabe = None
2
3 while eingabe not in ['Ja', 'Nein']:
4     eingabe = input('Geben Sie Ja oder Nein ein: ')
5
6 print('Ihre Eingabe: {}'.format(eingabe))
```

Da der Wert von `eingabe` zu Beginn noch unbekannt ist, weisen wir dieser Variable in Zeile 1 den Wert `None` („Nichts“) zu.

Wenn der Wert von `eingabe` in Zeile 3 weder `'Ja'` noch `'Nein'` ist, wird die eingerückte Zeile 4 ausgeführt und erneut zur Zeile 3 gesprungen. Andernfalls wird Zeile 4 ausgelassen und das

Schritt 2a

Nun können wir den Schritt 2 wie im obigen Beispiel implementieren. Da die Liste mit den gültigen Eingaben bereits in der Variablen "objekte" gespeichert wurde, verwenden wir sie in der while-Schleife.

```
1  import random
2
3  objekte = ['Schere', 'Stein', 'Papier']
4  wahl_comp = random.choice(objekte)
5
6  wahl_user = None
7  while wahl_user not in objekte:
8      wahl_user = input('Schere, Stein oder Papier? ')
9
10 print(wahl_comp, wahl_user) # Test
```

Zeichenketten verändern

Es gibt viele nützliche Methoden, mit denen wir Zeichenketten verändern können.

Methoden sind Funktionen, die mit der Punkt-Schreibweise auf Objekte eines bestimmten Datentyps angewendet werden.

Drei davon werden hier an einem Beispiel erläutert:

Ausdruck	Wert
`'MiScHmAsCh'.lower()`	`mischmasch`
`'MiScHmAsCh'.upper()`	`MISCHMASCH`
`'MiScHmAsCh'.capitalize()`	`Mischmasch`

Dies macht die Benutzereingabe robuster und die zweite Anforderung (siehe User Story) ist erfüllt.

Schritt 2b

```
1  import random
2
3  objekte = ['Schere', 'Stein', 'Papier']
4  wahl_comp = random.choice(objekte)
5
6  wahl_user = None
7  while wahl_user not in objekte:
8      wahl_user = input('Schere, Stein oder Papier? ')
9      wahl_user = wahl_user.capitalize()
10
11 print(wahl_comp, wahl_user) # Test
```

Vergleiche

In Python sind Vergleiche mit unterschiedlichen Operatoren möglich. Wenn sich zwei Werte vergleichen lassen, dann ist das Resultat eines Vergleichsausdrucks ein Boolescher Wert; also `True` oder `False`.

Syntax	Semantik	Mathematik
`a == b`	`True`, wenn `a` gleich `b`	$a = b$
`a != b`	`True`, wenn `a` ungleich `b`	$a \neq b$
`a < b`	`True`, wenn `a` kleiner als `b`	$a < b$
`a > b`	`True`, wenn `a` grösser als `b`	$a > b$
`a <= b`	`True`, wenn `a` kleiner gleich `b`	$a \leq b$
`a >= b`	`True`, wenn `a` grösser gleich `b`	$a \geq b$

Übung 3.5

Warum verwendet man für den Test auf Gleichheit den Operator "==" und nicht "="?

Übung 3.5

Warum verwendet man für den Test auf Gleichheit den Operator "==" und nicht "="?

"=" wird bereits für Zuweisungen verwendet.

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	
4	>	5	
3	<=	3	
2	>=	7	
4	==	4	
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	
3	<=	3	
2	>=	7	
4	==	4	
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	False
3	<=	3	
2	>=	7	
4	==	4	
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	False
3	<=	3	True
2	>=	7	
4	==	4	
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	False
3	<=	3	True
2	>=	7	False
4	==	4	
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	False
3	<=	3	True
2	>=	7	False
4	==	4	True
4	!=	4	

Vergleiche zwischen ganzen Zahlen

Die folgenden Vergleiche sollten aus dem Mathematikunterricht bekannt sein.

Operand	Operator	Operand	Wert
4	<	5	True
4	>	5	False
3	<=	3	True
2	>=	7	False
4	==	4	True
4	!=	4	False

Vergleiche zwischen Gleitkommazahlen

Auch diese Vergleiche sind nicht neu. Da bei Gleitkommazahlen meist ein kleiner Fehler in der letzten Stelle der Binärdarstellung ensteht (siehe *Binärdarstellung von Zahlen*), können Vergleiche zwischen ihnen ein falsches Resultat liefern. Daher sollte man in Programmen, die Leben oder Umwelt gefährden könnten, keine Vergleiche zwischen Gleitkommazahlen durchführen.

Beispiel: `0.1 + 0.1 + 0.1 > 0.3`

korrekter Wahrheitswert:

Von Python bestimmter Wahrheitswert:

Vergleiche zwischen Gleitkommazahlen

Auch diese Vergleiche sind nicht neu. Da bei Gleitkommazahlen meist ein kleiner Fehler in der letzten Stelle der Binärdarstellung ensteht (siehe *Binärdarstellung von Zahlen*), können Vergleiche zwischen ihnen ein falsches Resultat liefern. Daher sollte man in Programmen, die Leben oder Umwelt gefährden könnten, keine Vergleiche zwischen Gleitkommazahlen durchführen.

Beispiel: `0.1 + 0.1 + 0.1 > 0.3`

korrekter Wahrheitswert: `False`

Von Python bestimmter Wahrheitswert:

Vergleiche zwischen Gleitkommazahlen

Auch diese Vergleiche sind nicht neu. Da bei Gleitkommazahlen meist ein kleiner Fehler in der letzten Stelle der Binärdarstellung ensteht (siehe *Binärdarstellung von Zahlen*), können Vergleiche zwischen ihnen ein falsches Resultat liefern. Daher sollte man in Programmen, die Leben oder Umwelt gefährden könnten, keine Vergleiche zwischen Gleitkommazahlen durchführen.

Beispiel: `0.1 + 0.1 + 0.1 > 0.3`

korrekter Wahrheitswert: `False`

Von Python bestimmter Wahrheitswert: `True`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6`

(b) `15 // 5 == 7 - 4`

(c) `144 * 118 != 118 * 144`

(d) `37 % 15 <= 29 % 11`

(e) `1.5 + 4.2 < 3.7`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4`

(c) `144 * 118 != 118 * 144`

(d) `37 % 15 <= 29 % 11`

(e) `1.5 + 4.2 < 3.7`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4` `True`

(c) `144 * 118 != 118 * 144`

(d) `37 % 15 <= 29 % 11`

(e) `1.5 + 4.2 < 3.7`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4` `True`

(c) `144 * 118 != 118 * 144` `False`

(d) `37 % 15 <= 29 % 11`

(e) `1.5 + 4.2 < 3.7`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4` `True`

(c) `144 * 118 != 118 * 144` `False`

(d) `37 % 15 <= 29 % 11` `True`

(e) `1.5 + 4.2 < 3.7`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4` `True`

(c) `144 * 118 != 118 * 144` `False`

(d) `37 % 15 <= 29 % 11` `True`

(e) `1.5 + 4.2 < 3.7` `False`

(f) `0.1 * 0.2 > 0.02`

Übung 3.6

Bestimme den Wahrheitswert des Vergleichsausdrucks ohne Taschenrechner. Beachte, dass in Python (und in den meisten anderen Programmiersprachen) arithmetische Operatoren eine höhere Priorität als die Vergleichsoperatoren haben und daher um die arithmetischen Ausdrücke keine Klammern nötig sind.

(a) `5 + 3 > 2 + 6` `False`

(b) `15 // 5 == 7 - 4` `True`

(c) `144 * 118 != 118 * 144` `False`

(d) `37 % 15 <= 29 % 11` `True`

(e) `1.5 + 4.2 < 3.7` `False`

(f) `0.1 * 0.2 > 0.02` problematisch!

Vergleiche zwischen Zeichenketten

- Der Ausdruck `<string1> == <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in jedem Zeichen übereinstimmen. Sonst hat er den Wert `False`.
- Der Ausdruck `<string1> != <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in mindestens einem Zeichen nicht übereinstimmen. Sonst hat er den Wert `False`.

Beispiele:

Operand	Operator	Operand	Wert
`'Java'`	`==`	`'Python'`	
`'Papier'`	`!=`	`'papier'`	
`'Papier'`	`==`	`'Papier␣'`	

Vorsicht vor „unsichtbaren“ Leerzeichen (hier mit ␣ hervorgehoben).

Vergleiche zwischen Zeichenketten

- Der Ausdruck `<string1> == <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in jedem Zeichen übereinstimmen. Sonst hat er den Wert `False`.
- Der Ausdruck `<string1> != <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in mindestens einem Zeichen nicht übereinstimmen. Sonst hat er den Wert `False`.

Beispiele:

Operand	Operator	Operand	Wert
`'Java'`	`==`	`'Python'`	False
`'Papier'`	`!=`	`'papier'`	
`'Papier'`	`==`	`'Papier␣'`	

Vorsicht vor „unsichtbaren“ Leerzeichen (hier mit ␣ hervorgehoben).

Vergleiche zwischen Zeichenketten

- Der Ausdruck `<string1> == <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in jedem Zeichen übereinstimmen. Sonst hat er den Wert `False`.
- Der Ausdruck `<string1> != <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in mindestens einem Zeichen nicht übereinstimmen. Sonst hat er den Wert `False`.

Beispiele:

Operand	Operator	Operand	Wert
`'Java'`	`==`	`'Python'`	False
`'Papier'`	`!=`	`'papier'`	True
`'Papier'`	`==`	`'Papier␣'`	

Vorsicht vor „unsichtbaren“ Leerzeichen (hier mit ␣ hervorgehoben).

Vergleiche zwischen Zeichenketten

- Der Ausdruck `<string1> == <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in jedem Zeichen übereinstimmen. Sonst hat er den Wert `False`.
- Der Ausdruck `<string1> != <string2>` hat den Wert `True` wenn `<string1>` und `<string2>` in mindestens einem Zeichen nicht übereinstimmen. Sonst hat er den Wert `False`.

Beispiele:

Operand	Operator	Operand	Wert
`'Java'`	`==`	`'Python'`	False
`'Papier'`	`!=`	`'papier'`	True
`'Papier'`	`==`	`'Papier␣'`	False

Vorsicht vor „unsichtbaren“ Leerzeichen (hier mit ␣ hervorgehoben).

Beim Vergleich von Zeichenketten verwendet man die lexikographische Ordnung: Sind v und w zwei Wörter, dann ist v kleiner als w (kurz: $v < w$), wenn entweder ...

- v den Anfang von w bildet [`'Haus'` $<$ `'Hausaufgaben'`]

oder

- an der ersten Position, an der sich v und w unterscheiden, das Zeichen von v eine kleinere Zeichennummer hat als das Zeichen von w [`'MAULWURF'` $<$ `'MAUS'`, da `ord('L')` $= 76$ und `ord('S')` $= 83$]

Operand	Operator	Operand	Wert
`'Haus'`	`<`	`'Haushalt'`	
`'Axt'`	`<`	`'Axiom'`	
`'anna'`	`>`	`'ananas'`	
`'Arzt'`	`<`	`'als'`	

Beim Vergleich von Zeichenketten verwendet man die lexikographische Ordnung: Sind v und w zwei Wörter, dann ist v kleiner als w (kurz: $v < w$), wenn entweder ...

- v den Anfang von w bildet [`'Haus'` $<$ `'Hausaufgaben'`]

oder

- an der ersten Position, an der sich v und w unterscheiden, das Zeichen von v eine kleinere Zeichennummer hat als das Zeichen von w [`'MAULWURF'` $<$ `'MAUS'`, da `ord('L')` $= 76$ und `ord('S')` $= 83$]

Operand	Operator	Operand	Wert
`'Haus'`	`<`	`'Haushalt'`	True
`'Axt'`	`<`	`'Axiom'`	
`'anna'`	`>`	`'ananas'`	
`'Arzt'`	`<`	`'als'`	

Beim Vergleich von Zeichenketten verwendet man die lexikographische Ordnung: Sind v und w zwei Wörter, dann ist v kleiner als w (kurz: $v < w$), wenn entweder ...

- v den Anfang von w bildet [`'Haus'` $<$ `'Hausaufgaben'`]

oder

- an der ersten Position, an der sich v und w unterscheiden, das Zeichen von v eine kleinere Zeichennummer hat als das Zeichen von w [`'MAULWURF'` $<$ `'MAUS'`, da `ord('L')` $= 76$ und `ord('S')` $= 83$]

Operand	Operator	Operand	Wert
`'Haus'`	`<`	`'Haushalt'`	True
`'Axt'`	`<`	`'Axiom'`	False
`'anna'`	`>`	`'ananas'`	
`'Arzt'`	`<`	`'als'`	

Beim Vergleich von Zeichenketten verwendet man die lexikographische Ordnung: Sind v und w zwei Wörter, dann ist v kleiner als w (kurz: $v < w$), wenn entweder ...

- v den Anfang von w bildet [`'Haus'` $<$ `'Hausaufgaben'`]

oder

- an der ersten Position, an der sich v und w unterscheiden, das Zeichen von v eine kleinere Zeichennummer hat als das Zeichen von w [`'MAULWURF'` $<$ `'MAUS'`, da `ord('L')` $= 76$ und `ord('S')` $= 83$]

Operand	Operator	Operand	Wert
`'Haus'`	`<`	`'Haushalt'`	True
`'Axt'`	`<`	`'Axiom'`	False
`'anna'`	`>`	`'ananas'`	True
`'Arzt'`	`<`	`'als'`	

Beim Vergleich von Zeichenketten verwendet man die lexikographische Ordnung: Sind v und w zwei Wörter, dann ist v kleiner als w (kurz: $v < w$), wenn entweder ...

- v den Anfang von w bildet [`'Haus'` $<$ `'Hausaufgaben'`]

oder

- an der ersten Position, an der sich v und w unterscheiden, das Zeichen von v eine kleinere Zeichennummer hat als das Zeichen von w [`'MAULWURF'` $<$ `'MAUS'`, da `ord('L')` $= 76$ und `ord('S')` $= 83$]

Operand	Operator	Operand	Wert
`'Haus'`	<	`'Haushalt'`	True
`'Axt'`	<	`'Axiom'`	False
`'anna'`	>	`'ananas'`	True
`'Arzt'`	<	`'als'`	True

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'`

(b) `'Friend' > 'Friendo'`

(c) `'unterwegs' <= 'und'`

(d) `'Zettel' < 'aufschreiben'`

(e) `'hello' <= 'Hello'`

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'` False

(b) `'Friend' > 'Friendo'`

(c) `'unterwegs' <= 'und'`

(d) `'Zettel' < 'aufschreiben'`

(e) `'hello' <= 'Hello'`

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'` False

(b) `'Friend' > 'Friendo'` False

(c) `'unterwegs' <= 'und'`

(d) `'Zettel' < 'aufschreiben'`

(e) `'hello' <= 'Hello'`

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'` False

(b) `'Friend' > 'Friendo'` False

(c) `'unterwegs' <= 'und'` False

(d) `'Zettel' < 'aufschreiben'`

(e) `'hello' <= 'Hello'`

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'` False

(b) `'Friend' > 'Friendo'` False

(c) `'unterwegs' <= 'und'` False

(d) `'Zettel' < 'aufschreiben'` True

(e) `'hello' <= 'Hello'`

Übung 3.7

Bestimme den Wahrheitswert der Stringvergleiche in Python auf der Grundlage von Unicode. Beachte, dass dort lateinische Grossbuchstaben eine tiefere Nummer haben als ihre entsprechenden Kleinbuchstaben. *Bemerkung:* In Python lässt sich die Codenummer von `<zeichen>` mit `ord('<zeichen>')` bestimmen.

(a) `'Teller' < 'Tee'` False

(b) `'Friend' > 'Friendo'` False

(c) `'unterwegs' <= 'und'` False

(d) `'Zettel' < 'aufschreiben'` True

(e) `'hello' <= 'Hello'` False

Bedingte Anweisungen

Eine bedingte Anweisung ist ein Programmabschnitt, der nur dann ausgeführt wird, wenn eine Bedingung erfüllt ist.

```
1 if alter >= 18:
2     print('stimmberechtigt')
```

Syntax: Beginnt mit dem Schlüsselwort `if`, dem ein boolescher Ausdruck folgt. Danach muss ein Doppelpunkt stehen. Alle Zeilen des auszuführenden Programmabschnitts sind um die gleiche Anzahl Zeichen (meist 4) einzurücken.

Verzweigungen

Eine Verzweigung besteht aus einer Bedingung und zwei Programmabschnitten.

Hat die Bedingung den Wert `True`, dann wird nur der erste Programmabschnitt ausgeführt. Sonst wird nur der zweite ausgeführt.

```
1 if klasse < 3:
2     print('Untergymnasium')
3 else:
4     print('Obergymnasium')
```

Syntax: Zunächst identisch mit der bedingten Anweisung. Danach folgt auf der Höhe von `if` das Schlüsselwort `else`, hinter dem wieder ein Doppelpunkt stehen muss. Daran schliesst der eingerückte zweite Programmabschnitt an.

Mehrfache Verzweigungen

Eine mehrfache Verzweigung besteht aus n Bedingungen und $n+1$ Programmabschnitten ($n \geq 2$).

Die erste Bedingung mit dem Wert `True` bewirkt die Ausführung des unmittelbar folgenden Programmabschnitts und das Überspringen der restlichen. Sind alle n Bedingungen `False`, dann wird der letzte Programmabschnitt nach `else` ausgeführt.

```
1 if <bedingung1>:
2     <codeblock1>
3 elif <bedingung2>:
4     <codeblock2>
5 else:
6     <codeblock3>
```

Syntax: Analog zu einer einfachen Verzweigung. Jedes weitere Paar Bedingung/Programmabschnitt hat die Form

```
1 elif <bedingung>:
2     <programmabschnitt>
```

Übung 3.8

Welche Ausgabe macht das folgende Programmfragment,

```
1 if x < 5:
2     x = x + 2
3 x = x + 1
4 print(x)
```

(a) wenn `x` den Wert `8` hat?

(b) wenn `x` den Wert `3` hat?

Übung 3.9

Welche Ausgabe macht das folgende Programmfragment,

```
1  if geschwindigkeit < 51:
2      print('ok')
3  elif: geschwindigkeit < 56:
4      print('Busse CHF 40.--')
5  elif: geschwindigkeit < 61:
6      print('Busse CHF 120.--')
7  elif: geschwindigkeit < 66:
8      print('Busse CHF 250.--')
9  else:
10     print('Führerausweisentzug')
```

(a) wenn `geschwindigkeit` den Wert 59 hat?

(b) wenn `geschwindigkeit` den Wert 67 hat?

(c) wenn `geschwindigkeit` den Wert 61 hat?

Übung 3.10

Finde und beschreibe alle Syntaxfehler im Programmfragment.

```
1  if note == 6:
2      print('sehr gut')
3  elif note == 5:
4      print('gut')
5  elif note == 4
6      print('genügend')
7  elif note = 3:
8      print('ungenügend')
9  elif note == 2:
10      print('schwach')
11 else if note == 1:
12     print("sehr schwach")
13 else:
14     print('keine gültige ganze Note")
```

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	
Schere	Stein	
Schere	Papier	
Stein	Schere	
Stein	Stein	
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	
Schere	Papier	
Stein	Schere	
Stein	Stein	
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	
Stein	Schere	
Stein	Stein	
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	
Stein	Stein	
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	K
Stein	Papier	
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	K
Stein	Papier	U
Papier	Schere	
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	K
Stein	Papier	U
Papier	Schere	U
Papier	Stein	
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	K
Stein	Papier	U
Papier	Schere	U
Papier	Stein	C
Papier	Papier	

Wer gewinnt?

Computer	User	Gewinner (C/U/K)
Schere	Schere	K
Schere	Stein	U
Schere	Papier	C
Stein	Schere	C
Stein	Stein	K
Stein	Papier	U
Papier	Schere	U
Papier	Stein	C
Papier	Papier	K

Die möglichen Resultate

Aufgrund der obigen Tabelle können wir drei Fälle erkennen, für die wir jeweils einen passenden Ausgabetext vorbereiten:

```
11 kg = 'unentschieden (beide wählen {})'.format(wahl_comp)
12 cg = 'Computer wählt {} und gewinnt.'.format(wahl_comp)
13 ug = 'Computer wählt {} und verliert.'.format(wahl_comp)
```

Boolesche Operatoren

In vielen Fällen ist das Ausführen einer Aktion vom Nichteintreten eines Zustands oder in bestimmter Weise vom Eintreten mehrerer Zustände abhängig.

- „Ich kaufe diese Turnschuhe, wenn sie *nicht* mehr als 100 Franken kosten.“
- „Du kannst den Film sehen, wenn du die Aufgaben erledigt *und* dein Zimmer aufgeräumt hast.“
- „Du bekommst 10 Franken, wenn du den Rasen mähst *oder* das Auto wäschst.“

Um solche Situationen in einem Programm zu formulieren, bedarf es der logischen Operatoren; also Operatoren, deren Operanden Wahrheitswerte sind.

Die Negation

Die Negation (logische Verneinung) ist eine einstellige logische Operation, wobei der Python-Operator `not` *vor* dem Operanden steht (Präfix-Notation).

`not <op>` hat genau dann den Wert `True`, wenn der Operand `<op>` den Wert `False` hat.

Ausdruck	Wert
`not True`	
`not False`	

Die Negation

Die Negation (logische Verneinung) ist eine einstellige logische Operation, wobei der Python-Operator `not` *vor* dem Operanden steht (Präfix-Notation).

`not <op>` hat genau dann den Wert `True`, wenn der Operand `<op>` den Wert `False` hat.

Ausdruck	Wert
`not True`	False
`not False`	

Die Negation

Die Negation (logische Verneinung) ist eine einstellige logische Operation, wobei der Python-Operator `not` *vor* dem Operanden steht (Präfix-Notation).

`not <op>` hat genau dann den Wert `True`, wenn der Operand `<op>` den Wert `False` hat.

Ausdruck	Wert
`not True`	False
`not False`	True

Die Konjunktion

Die Konjunktion (logisches UND) ist eine zweistellige logische Operation, wobei der Python-Operator `and` *zwischen* den Operanden steht (Infix-Notation).

`<op1> and <op2>` hat genau dann den Wert `True`, wenn beide Operanden `<op1>` und `<op2>` den Wert `True` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`and`	`False`	
`False`	`and`	`True`	
`True`	`and`	`False`	
`True`	`and`	`True`	

Die Konjunktion

Die Konjunktion (logisches UND) ist eine zweistellige logische Operation, wobei der Python-Operator `and` *zwischen* den Operanden steht (Infix-Notation).

`<op1> and <op2>` hat genau dann den Wert `True`, wenn beide Operanden `<op1>` und `<op2>` den Wert `True` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`and`	`False`	False
`False`	`and`	`True`	
`True`	`and`	`False`	
`True`	`and`	`True`	

Die Konjunktion

Die Konjunktion (logisches UND) ist eine zweistellige logische Operation, wobei der Python-Operator `and` *zwischen* den Operanden steht (Infix-Notation).

`<op1> and <op2>` hat genau dann den Wert `True`, wenn beide Operanden `<op1>` und `<op2>` den Wert `True` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`and`	`False`	False
`False`	`and`	`True`	False
`True`	`and`	`False`	
`True`	`and`	`True`	

Die Konjunktion

Die Konjunktion (logisches UND) ist eine zweistellige logische Operation, wobei der Python-Operator `and` *zwischen* den Operanden steht (Infix-Notation).

`<op1> and <op2>` hat genau dann den Wert `True`, wenn beide Operanden `<op1>` und `<op2>` den Wert `True` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`and`	`False`	False
`False`	`and`	`True`	False
`True`	`and`	`False`	False
`True`	`and`	`True`	

Die Konjunktion

Die Konjunktion (logisches UND) ist eine zweistellige logische Operation, wobei der Python-Operator `and` *zwischen* den Operanden steht (Infix-Notation).

`<op1> and <op2>` hat genau dann den Wert `True`, wenn beide Operanden `<op1>` und `<op2>` den Wert `True` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`and`	`False`	False
`False`	`and`	`True`	False
`True`	`and`	`False`	False
`True`	`and`	`True`	True

Die Disjunktion

Die Disjunktion (logisches ODER) ist eine zweistellige logische Operation, wobei der Python-Operator `or` *zwischen* den Operanden steht (Infix-Notation).

`<op1> or <op2>` hat genau dann den Wert `False`, wenn beide Operanden `<op1>` und `<op2>` den Wert `False` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`or`	`False`	
`False`	`or`	`True`	
`True`	`or`	`False`	
`True`	`or`	`True`	

Die Disjunktion

Die Disjunktion (logisches ODER) ist eine zweistellige logische Operation, wobei der Python-Operator `or` *zwischen* den Operanden steht (Infix-Notation).

`<op1> or <op2>` hat genau dann den Wert `False`, wenn beide Operanden `<op1>` und `<op2>` den Wert `False` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`or`	`False`	False
`False`	`or`	`True`	
`True`	`or`	`False`	
`True`	`or`	`True`	

Die Disjunktion

Die Disjunktion (logisches ODER) ist eine zweistellige logische Operation, wobei der Python-Operator `or` *zwischen* den Operanden steht (Infix-Notation).

`<op1> or <op2>` hat genau dann den Wert `False`, wenn beide Operanden `<op1>` und `<op2>` den Wert `False` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`or`	`False`	False
`False`	`or`	`True`	True
`True`	`or`	`False`	
`True`	`or`	`True`	

Die Disjunktion

Die Disjunktion (logisches ODER) ist eine zweistellige logische Operation, wobei der Python-Operator `or` *zwischen* den Operanden steht (Infix-Notation).

`<op1> or <op2>` hat genau dann den Wert `False`, wenn beide Operanden `<op1>` und `<op2>` den Wert `False` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`or`	`False`	False
`False`	`or`	`True`	True
`True`	`or`	`False`	True
`True`	`or`	`True`	

Die Disjunktion

Die Disjunktion (logisches ODER) ist eine zweistellige logische Operation, wobei der Python-Operator `or` *zwischen* den Operanden steht (Infix-Notation).

`<op1> or <op2>` hat genau dann den Wert `False`, wenn beide Operanden `<op1>` und `<op2>` den Wert `False` haben. Als Wertetabelle:

Operand	Operator	Operand	Wert
`False`	`or`	`False`	False
`False`	`or`	`True`	True
`True`	`or`	`False`	True
`True`	`or`	`True`	True

Präzedenz der logischen Operatoren

höchste	`not`
	`and`
niedrigste	`or`

Die Vergleichsoperatoren (==, <, . . .) haben alle eine höhere Präzedenz als die logischen Operatoren und wie bereits erwähnt, haben die arithmetischen Operatoren (+, -, . . .) eine höhere Präzedenz als die Vergleichsoperatoren.

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`

(b) `not True or not False`

(c) `True and not False or False`

(d) `False and True or not True or True`

(e) `False or False and True or False`

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`
`False or (not True) => False`

(b) `not True or not False`

(c) `True and not False or False`

(d) `False and True or not True or True`

(e) `False or False and True or False`

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`

`False or (not True) => False`

(b) `not True or not False`

`(not True) or (not False) => True`

(c) `True and not False or False`

(d) `False and True or not True or True`

(e) `False or False and True or False`

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`
`False or (not True) => False`

(b) `not True or not False`
`(not True) or (not False) => True`

(c) `True and not False or False`
`(True and (not False)) or False => True`

(d) `False and True or not True or True`

(e) `False or False and True or False`

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`
`False or (not True) => False`

(b) `not True or not False`
`(not True) or (not False) => True`

(c) `True and not False or False`
`(True and (not False)) or False => True`

(d) `False and True or not True or True`
`(False and True) or ((not True) or True)) => False`

(e) `False or False and True or False`

Übung 3.11

Setze Klammern gemäss den Präferenzen für logische Operatoren und bestimme damit den Wert des Ausdrucks.

(a) `False or not True`
`False or (not True) => False`

(b) `not True or not False`
`(not True) or (not False) => True`

(c) `True and not False or False`
`(True and (not False)) or False => True`

(d) `False and True or not True or True`
`(False and True) or ((not True) or True)) => False`

(e) `False or False and True or False`
`False or (False and True) or False => False`

Übung 3.12

(a) Bestimme den Wahrheitswert des Ausdrucks:

```
False and (True or True)
```

(b) Könnte man die Klammern in (a) auch weglassen?

Übung 3.12

(a) Bestimme den Wahrheitswert des Ausdrucks:

```
False and (True or True)
```

```
=> False and True => False
```

(b) Könnte man die Klammern in (a) auch weglassen?

Übung 3.12

(a) Bestimme den Wahrheitswert des Ausdrucks:

```
False and (True or True)
```

```
=> False and True => False
```

(b) Könnte man die Klammern in (a) auch weglassen?

```
False and True or True => (False and True) or
True => False or True => True => nein
```

Schritt 3

```
15  if wahl_user == wahl_comp:
16      resultat = kg # keiner gewinnt
17  elif wahl_user == 'Schere' and wahl_comp == 'Stein':
18      resultat = cg # Computer gewinnt
19  elif wahl_user == 'Stein' and wahl_comp == 'Papier':
20      resultat = cg # Computer gewinnt
21  elif wahl_user == 'Papier' and wahl_comp == 'Schere':
22      resultat = cg # Computer gewinnt
23  else:
24      resultat = ug # User gewinnt
```

Schritt 4

Die Ausgabe des Resultats ist nun kein Problem mehr.

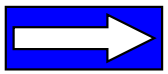
```
26 print(resultat)
```

Strukturierte Programmierung

Der Satz von Böhm und Jacopini (1966): Jedes algorithmisch berechenbare Problem lässt sich durch eine geeignete Kombination der folgenden drei Kontrollstrukturen darstellen:

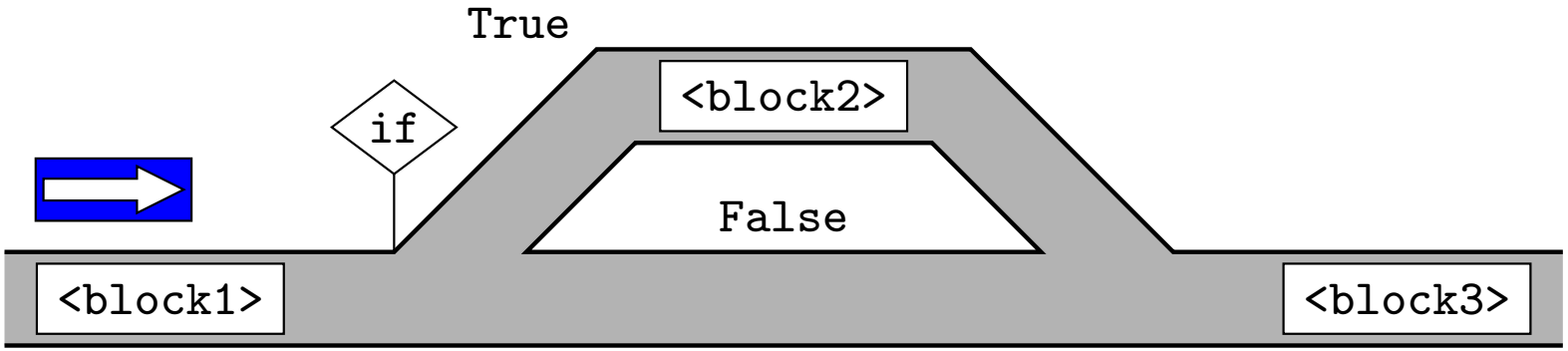
- Sequenz
- Selektion (Auswahl)
- Iteration (Wiederholung)

Sequenz



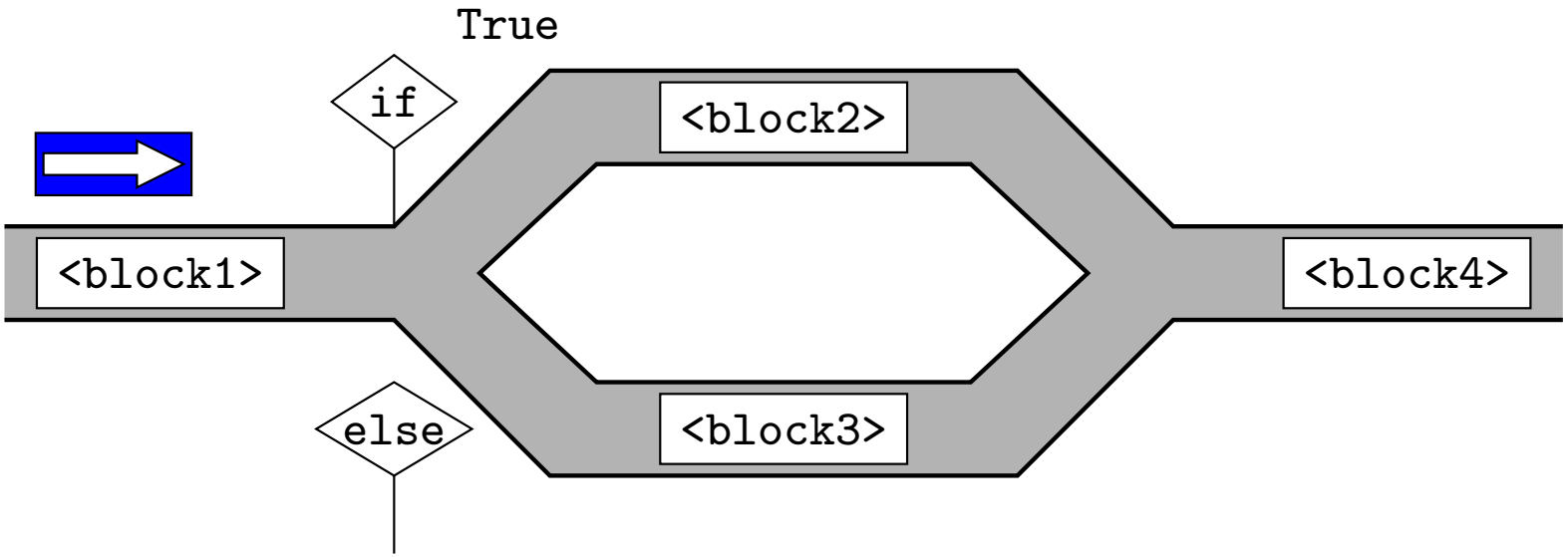

```
1 <anweisung1>
2 <anweisung2>
3 <anweisung3>
```

Bedingte Anweisung



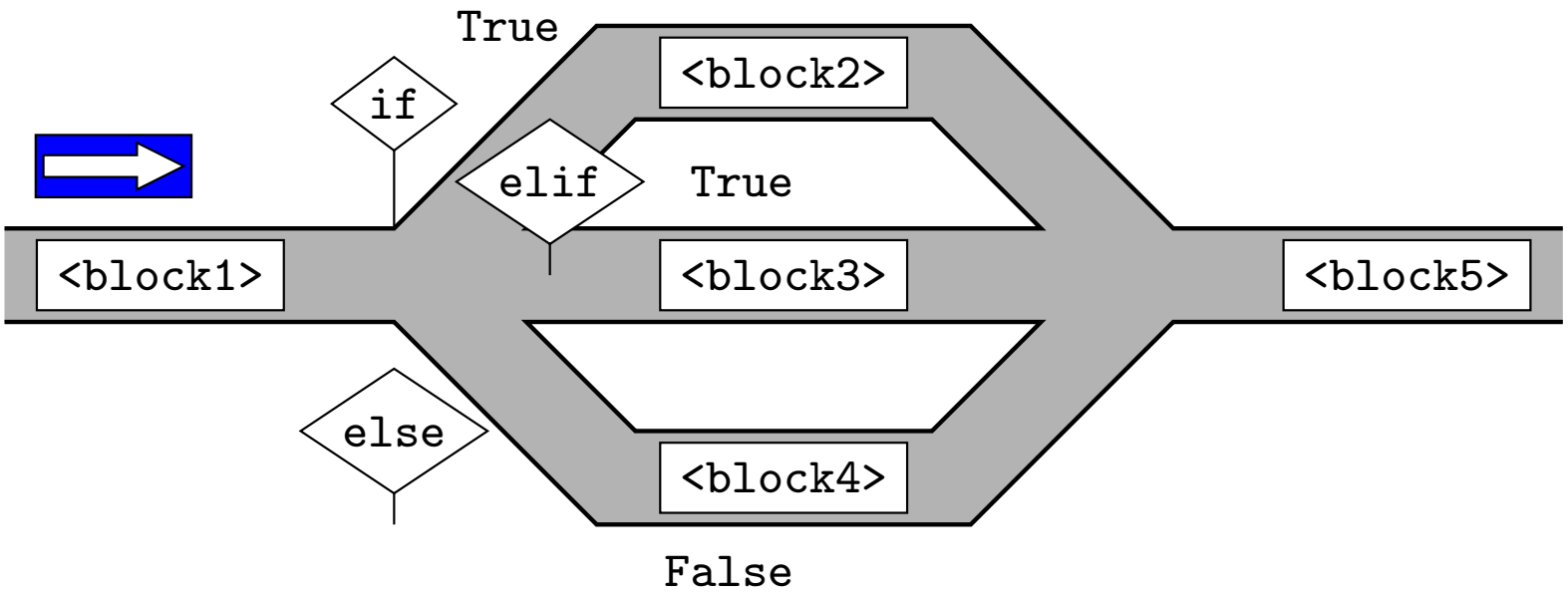

```
1 <block1>
2 if <bedingung>:
3     <block2>
4 <block3>
```

Einfache Alternative



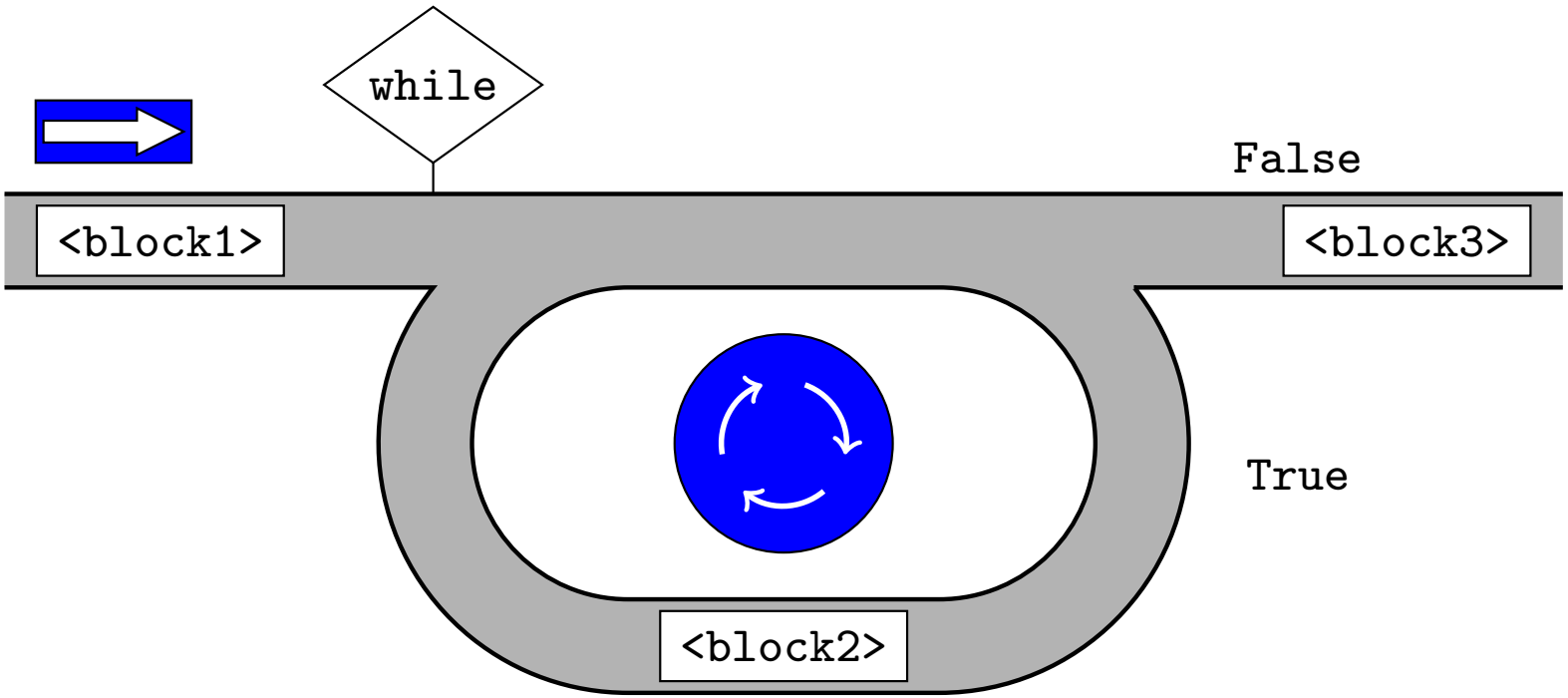

```
1 <block1>
2 if <bedingung>:
3     <block2>
4 else:
5     <block3>
6 <block4>
```

Mehrfache Alternative




```
1  <block1>
2  if <bedingung1>:
3      <block2>
4  elif <bedingung2>:
5      <block3>
6  else:
7      <block4>
```

Schleife




```
1 <block1>
2 while <bedingung>:
3     <block2>
4 <block3>
```

Auf den Punkt gebracht

- Flussdiagramme sind ein Hilfsmittel, um den Ablauf eines Algorithmus zu visualisieren. Sie bestehen aus Ovalen (Start/Ende), Rechtecken (Anweisungen), Rhomben (Verzweigungen) und Pfeilen (Programmfluss). Da Flussdiagramme viel Platz brauchen und bei umfangreichen Prozessen unübersichtlich werden, eignen sie sich nur für kleinere Projekte.
- Die Liste ist eine Datenstruktur von Python, die es erlaubt, beliebige Elemente in einer festen Reihenfolge zu speichern. Listen werden durch eckige Klammern (`[...]`) begrenzt und die Elemente durch Kommas getrennt.
- In Python hat man über den Import des Moduls `random` Zufgriff auf Methoden zur Erzeugung zufälliger Zahlen und Stichproben. Die Methode `random.choice(<liste>)` wählt ein zufälliges Element aus der Liste im Argument aus.

- Die Ausdrücke `<objekt> in <liste>` und `<objekt> not in <liste>` liefern jeweils den booleschen Wert `True`, wenn das Objekt in der Liste enthalten bzw. nicht enthalten ist.
- Die logischen Konstanten `True` und `False` sind die einzigen Werte des Datentyps `Boolean`.
- Die `while`-Schleife hat folgende Syntax:

```
while <wahrheitswert>:
    <codeblock>
```

  Der eingerückte Codeblock wird so oft ausgeführt, wie der Wahrheitswert `True` ist. Ist der Wahrheitswert `False`, wird das Programm unterhalb des Codeblocks fortgesetzt.
- Python verwendet Einrückungen anstelle von Klammern, um zusammengehörende Programmteile zu kennzeichnen. Meist wird um 4 Leerzeichen eingerückt.

- Die Stringmethoden `<str>.lower()`, `<str>.upper()` bzw. `<str>.capitalize()` liefern die Zeichenkette `<str>` in Kleinbuchstaben, Grossbuchstaben bzw. mit grossem Anfangsbuchstaben zurück.
- Ausdrücke mit den zweistelligen Vergleichsoperatoren <, >, <=, >=, ==, != liefern jeweils einen booleschen Wert zurück. Python gibt eine Fehlermeldung aus, wenn der Test auf Gleichheit (==) irrtümlich mit dem Zuweisungsoperator (=) durchgeführt wird.
- Vergleichsoperatoren haben eine niedrigere Priorität als arithmetische Operatoren aber eine höhere als die logischen.
- Mit den Vergleichsoperatoren können ganze Zahlen und Gleitkommazahlen verglichen werden. Vergleiche mit Gleitkommazahlen sollte man wegen möglicher Rundungsprobleme vermeiden.

- Zeichenketten lassen sich lexikographisch ordnen. Ist die erste Zeichenkette ein echtes Präfix einer zweiten, dann ist die erste Zeichenkette kleiner als die zweite. Ist eine Zeichenkette kein Präfix einer zweiten, dann gibt es eine erste Position, an der sich beide Zeichenketten unterscheiden. Hat die erste Zeichenkette an dieser Position ein Zeichen, deren Zeichennummer kleiner ist als die des entsprechenden Zeichens der zweiten Zeichenkette, dann ist die erste Zeichenkette kleiner als die zweite.
- Die bedingte Anweisung hat folgende Syntax:

```
if <bedingung>:
    <codeblock>
```

Der Codeblock wird genau dann ausgeführt, wenn die Bedingung `True` ist.

- Die (einfache) Verzweigung hat folgende Syntax:

```
if <bedingung>:
    <codeblock1>
else:
    <codeblock2>
```

Der erste Codeblock wird ausgeführt, wenn die Bedingung `True` ist; sonst der zweite.

- Die mehrfache Verzweigung hat folgende Syntax:

```
if <bedingung1>:
    <codeblock1>
elif <bedingung2>:
    <codeblock2>
else:
    <codeblock3>
```

Es wird nur der Codeblock ausgeführt, dessen Bedingung erstmals erfüllt ist. Alle anderen nicht. Ist keine der