

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1 Einführung	3
2 Datentypen	7
3 Variablen	9
4 Arithmetische Operationen	19
5 Ausgabe	25
6 Vergleichsoperatoren	28
7 Bedingte Anweisungen	36
8 for-Schleifen	46
9 while-Schleifen	53
10 Listen	61

1 Einführung

In diesem Kurs lernen Sie die Grundlagen von Python, einer der beliebtesten Programmiersprachen der Welt.

In diesem Kurs lernen Sie das Programmieren anhand von vier verschiedenen Fähigkeiten, die für das Programmieren entscheidend sind:

1. **Code lesen:** Sie werden lernen, Codezeilen zu lesen und vorherzusagen, wie der Code ausgeführt wird.
2. **Code schreiben:** Sie werden lernen, korrekten Code zu schreiben, den ein Computer ausführen kann.
3. **Code Templates lesen:** Templates sind wiederverwendbare Muster im Code. Sie werden lernen, einige gängige Templates und ihre Verwendung zu erkennen.
4. **Code Templates schreiben:** Anhand einer Problembeschreibung lernen Sie, ein Template zu finden, das dieses Problem löst, und den entsprechenden Code zu schreiben.

Obwohl die meisten Menschen beim Programmieren an das Schreiben von Code zur Lösung von Problemen denken, ist auch das Lesen von Code eine wichtige Fähigkeit! Tatsächlich verbringen Programmierer die meiste Zeit mit dem Lesen von Code.

In etwa 10 Stunden Unterricht und Übung werden Sie in der Lage sein, einfache Python-Programme zu schreiben! Los geht's!

1.1 Planen ihres Codes

Bei einigen Übungsaufgaben werden wir Sie bitten, einen Schritt-für-Schritt-Plan zu erstellen, der den von Ihnen zu schreibenden Code beschreibt. Dieser Plan sollte so detailliert sein, dass ein Fremder ihn sich ansehen und Ihren Gedankengängen folgen kann, wenn Sie programmieren.

Ein Beispiel: Meine Aufgabe ist es, einem Kunden Wechselgeld zu geben, wenn er mit einem 20€-Schein für einen Artikel zahlt, der 3,59€ kostet.

Mein Plan könnte folgendermaßen aussehen:

1. Ermitteln des Gesamtbetrages, den ich dem Kunden als Wechselgeld geben muss, indem ich die Kosten des Artikels von dem Betrag abziehe, den er mir gegeben hat.
3. 1 Ermitteln der Anzahl der benötigten 10€-Scheine, indem 10 vom Gesamtbetrag abgezogen werden, bis weniger als 10€ für den Kunden übrig sind.
2. Wiederhole Schritt 2 für 5€.
3. Wenn weniger als 5€ übrig bleibt, wiederhole Schritt 2 mit 2€, 1€, 50ct, 20ct, 10ct, 5ct, 2ct, 1c.
4. Höre auf, wenn der verbleibende Betrag, den ich dem Kunden geben muss, 0 € beträgt.

Zusammenfassend lässt sich sagen, dass Sie sich sowohl mit den metakognitiven Aufforderungen (kommentieren Sie Ihren Code, planen Sie Ihren Code) als auch mit dem Lesen und Schreiben von Code befassen sollten. **Die Aufforderungen und Übungen werden Ihnen helfen, besser zu lernen, wie man Code schreibt!**

1.2 Code kommentieren

Programmierer fügen häufig Kommentare zu ihrem Code hinzu. Kommentare ermöglichen es anderen, die Vorgänge im Code besser zu verstehen. In Python stehen die Kommentare rechts von einer `#` und reichen bis zum Ende der Zeile.

Im folgenden Beispiel sind die 3 Kommentare hervorgehoben. Wir sehen, dass Hervorhebungen in einer eigenen Zeile erscheinen können (erster Kommentar, dritter Kommentar) oder in einer Zeile mit einem Code, den der Computer tatsächlich ausführt (zweiter Kommentar)).

```
#Dies ist der erste Kommentar

spam = 1 #und dies ist der zweite Kommentar
        #... und hier der dritte!
text = "#Dies ist kein Kommentar wegen der Anführungszeichen"
```

Während die Kommentare in diesem Beispiel etwas sinnlos sind, bitten wir Sie, Ihren Code mit sinnvollen Kommentaren zu versehen. Denken Sie daran: Kommentare befinden sich auf der rechten Seite einer `#`.

1.3 Wie Code ausgeführt wird

Computer führen Code in der Regel zeilenweise von oben nach unten und von links nach rechts aus. Beim Lesen und Schreiben von Code sollten auch Sie den Code von oben nach unten, von links nach rechts, ausführen.

Wenn Sie Probleme lösen, bei denen Sie Code lesen sollen, müssen Sie "der Computer sein". Das können Sie tun, indem Sie die folgenden Schritte befolgen:

1. **Lesen Sie die Frage.**
2. **Finden Sie den Anfang der Codeausführung.**
3. **Führen Sie den Code zeilenweise aus.**
 - 3.1 Bestimmen Sie anhand des Codes die Regeln für jeden Teil der Zeile (Sie werden diese Regeln lernen).
 - 3.2 Befolgen Sie die Regeln.
 - 3.3 Finden Sie den Code für den nächsten Teil.
 - 3.4 Wiederholen Sie den Vorgang, bis das Programm beendet wird.

Ein Schritt dieser Strategie hat damit zu tun, gespeicherte Werte (Variablen) im Auge zu behalten. Wir werden darauf zurückkommen, wenn wir Variablen unterrichten.

1.4 Überblick

Dies ist ein Überblick über das, was wir behandeln werden. Am Ende des Kurses wird das alles einen Sinn ergeben!

Konzept	Beschreibung	Beispiele
Datentypen	Verschiedene Klassifizierungen von Daten (string, boolean, integer, float).	"Hello" True 3 3.1
Variablen	Speichern Werte, die später verwendet werden können. Können aktualisiert werden.	cost = 1.50 cost = 1
Arithmetische Operationen	Rechenoperationen zwischen Zahlen.	8 / 2 (3 + 1) * 4 7 % 3
Ausgabe	Ausgabewerte, die auf der Konsole angezeigt werden sollen.	print("Hello World!")
Vergleichsoperatoren	Bestimmen, ob eine Beziehung zwischen zwei Werten gültig ist oder nicht.	3 < 7 "hi" == "HI"
Bedingte Anweisung und Verzweigung	Ausführen von unterschiedlichem Code je nach Bedingung.	cost = 1.4 if cost < 1: print("Kauf es!") else: print("Kauf es nicht!")
for-Schleifen	Mehrmaliges Ausführen von Codestücken.	for i in range(5): print("Hello World!")
while-Schleifen	Mehrmaliges Ausführen von Codestücken.	count = 0 while count < 5: print("Hello World!")
Listen	Speichern mehrerer Elemente in einer Liste.	number_list = [1, 2, 3, 4] for number in number_list: print(number)

2 Datentypen

2.1 Einführung: Lesen

Computer benutzen verschiedene Datentypen, damit sie präzise sein können. Das ist die gleiche Art von Präzision wie in der Mathematik: Zwei Äpfel und drei Hunde zu addieren, würde keinen Sinn ergeben. Durch unterschiedliche Datentypen kann Python Ihnen helfen, solche Fehler zu vermeiden!

Typen sind verschiedene Klassifizierungen für Daten. In Python gibt es 3 gängige Datentypen: Zahlen, Strings und Boolesche Werte. Im Folgenden werden Beispiele für jeden Datentyp gezeigt.

Datentyp	Beispiel	Beschreibung
integer (Zahl)	1, 2018	Eine Zahl ohne Dezimalstelle.
float (Zahl)	1.0, 3.1415	Eine Zahl mit Dezimalstelle.
string (Zeichenkette)	"Hello", 'hello', "123", "%1+2"	Buchstaben und Zeichen, die von Anführungszeichen umgeben sind ('einfach' oder "doppelt").
boolean	True, False	Wahrheitswerte (kann nur True oder False sein).

Zahlen können ganze Zahlen oder Dezimalzahlen sein. Ganze Zahlen werden auch **integer** genannt (z.B. 1, 2, 3) und Dezimalzahlen sind **floats** (z.B. 3.14, 1.0). Beachten Sie, dass Floats einen Dezimalpunkt haben, Integer dagegen nicht. Demnach sind 1.0 und 1. Floats, aber 1 (ohne einen Dezimalpunkt) ist ein Integer.

Strings sind Folgen von Zeichen, die in 'einfache Anführungszeichen' oder "doppelte Anführungszeichen" eingeschlossen sind. Sie werden als Literale behandelt, was bedeutet, dass die Zeichen innerhalb der Anführungszeichen in der Regel nicht ausgeführt werden und "buchstäblich" so erscheinen, wie sie sind.

Boolean sind Wahrheitswerte für die Logik. Sie können nur **True** (Wahr) oder **False** (Falsch) sein. Beachten Sie, dass der erste Buchstabe in Großbuchstaben geschrieben wird. true oder false (Kleinbuchstaben) würde dazu führen, dass Ihr Code nicht ausgeführt wird.

Jetzt wollen wir die Datentypen von Python üben!

Bearbeiten Sie bitte **Übung 1.1.**

2.2 Einführung: Schreiben

Hier fassen wir nochmal zusammen, was beim Schreiben von Code mit Datentypen zu beachten ist:

- Strings werden immer in Anführungszeichen gesetzt. In Python können diese Anführungszeichen 'einfache' oder "doppelte" Anführungszeichen sein. Achten Sie aber darauf, dass die Zeichenketten in die gleichen Anführungszeichen eingeschlossen sind! So sind 'hallo' und "hallo" beides gültige Zeichenketten, aber "hallo' (mit unterschiedlichen Anführungszeichen auf jeder Seite) würde zu einem Fehler führen.
- Zahlen sind floats, wenn sie einen Dezimalpunkt haben (1.1, 1.0, 1.) und integer, wenn sie keinen haben (1, 2, 3).
- Boolean sind entweder **True** oder **False**. Sie werden nicht in Anführungszeichen gesetzt, sonst wären sie Strings. Denken Sie daran, den ersten Buchstaben groß zu schreiben!

Jetzt wollen wir üben, wie man Datentypen schreibt!

Bearbeiten Sie bitte **Übung 1.2.**

3 Variablen

3.1 Einführung: Lesen

Oft wollen wir Werte verschiedener Datentypen speichern und sie später verwenden. Wir tun dies mit Variablen.

Nehmen wir zum Beispiel an, wir wollen die Anzahl der Personen, die an einer Party teilnehmen, festhalten. Zu diesem Zweck könnte man eine Variable erstellen, beziehungsweise "deklarieren":

num_people = 25

Name Zuweisungsoperator Wert

Um diese Variable zu deklarieren, benötigen wir einen Variablennamen auf der linken Seite (in diesem Fall `num_people`), einen Zuweisungsoperator (ein einzelnes Gleichheitszeichen `=`) in der Mitte und einen Variablenwert (in diesem Fall `25`) auf der rechten Seite. Damit wird der Wert in der Variable gespeichert, sodass wir später über den Variablennamen auf sie verweisen können.

Der Wert der Variable (rechts vom Gleichheitszeichen) ist im obigen Beispiel ein Integer, kann aber auch ein String oder ein boolescher Wert sein. Er kann sogar eine andere Variable sein (dazu später mehr)!

3 Variablen

Den Namen der Variable (links vom Gleichheitszeichen) können sie sich frei ausdenken, aber es gibt ein paar Regeln, die für Variablennamen in Python gelten:

Regeln für Variablennamen	Schlechte Beispiele (nicht erlaubt in Python)	Gute Beispiele (erlaubt in Python)
Variablennamen dürfen nur aus Buchstaben, Zahlen und Unterstrichen bestehen. Andere Sonderzeichen als Unterstriche sind in Variablennamen nicht zulässig (z. B. %, -, !, @, #, usw.).	<code>greeting%message</code> <code>excited_response!</code> <code>phone#</code>	<code>greeting_messages</code> <code>excited_response</code> <code>phone_num</code>
Variablennamen dürfen nicht mit Zahlen beginnen.	<code>1st_period</code>	<code>first_period</code>
Einige Wörter in Python sind speziell und Variablennamen können nicht identisch mit diesen sein. Zum Beispiel wären <code>True</code> und <code>False</code> keine gültigen Variablennamen, da es sich um boolesche Werte handelt und der Computer verwirrt werden würde!	<code>True</code> <code>False</code>	<code>is_true</code> <code>is_false</code>
Bei Variablennamen wird zwischen Groß- und Kleinschreibung unterschieden (<code>my_var</code> und <code>MY_VAR</code> wären also Namen für unterschiedliche Variablen).	Die folgenden Variablennamen sind alle unterschiedlich: <code>my_var</code> <code>My_Var</code> <code>MY_VAR</code>	

Lassen Sie uns üben, wie man feststellt, ob Variablennamen gültig sind!

Bearbeiten Sie bitte **Übung 2.1.**

3 Variablen

Hier sind einige Beispiele für Code, mit dem Variablen deklariert werden:

Code zur Deklaration von Variablen	Erklärung
<code>first_name = "timmy"</code>	Eine Variable mit dem Namen "first_name" wird mit dem String-Wert "timmy" deklariert.
<code>is_happy = True</code>	Eine Variable mit dem Namen "is_happy" wird mit dem Booleschen-Wert True deklariert.
<code>age = 18</code>	Eine Variable mit dem Namen "age" wird mit dem Integerwert 18 deklariert.
<code>cost = 3.5</code>	Eine Variable mit dem Namen "cost" wird mit dem Floatwert 3.5 deklariert.

Lassen Sie uns üben, ob Variablendeklarationen korrekt sind!

Bearbeiten Sie bitte **Übung 2.2.**

3.1.1 Vervollständigung der Strategie zum Lesen des Codes

Bevor wir uns weiter mit dem Lesen von Variablen beschäftigen, sollten wir eine Pause einlegen, um unsere Strategie zum Lesen von Code zu vervollständigen. Erinnern Sie sich daran, dass wir Ihnen zu Beginn dieser Anleitung eine Strategie zum Lesen von Code vorgestellt haben. Wir haben festgestellt, dass in dieser Strategie ein Teil fehlt, der sich auf das Einprägen von Variablen bezieht. Vervollständigen wir nun die Strategie mit der hervorgehobenen Zeile.

Wenn Sie Probleme lösen, bei denen Sie Code lesen sollen, müssen Sie "der Computer sein". Das können Sie tun, indem Sie die folgenden Schritte befolgen:

1. **Lesen Sie die Frage**
2. **Finden Sie den Anfang der Codeausführung**
3. **Führen Sie den Code zeilenweise aus**
 - 3.1 Bestimmen Sie anhand des Codes die Regel für jeden Teil der Zeile (Sie werden diese Regeln lernen).
 - 3.2 Befolgen Sie die Regeln.
 - 3.3 **Aktualisieren Sie die Speichertabelle(n).**
 - 3.4 Finden Sie den Code für den nächsten Teil.
 - 3.5 Wiederholen Sie den Vorgang, bis das Programm beendet wird.

Speichertabellen sind abstrahierte Darstellungen des Computerspeichers, in dem Variablen und ihre Werte gespeichert werden. Das könnte zum Beispiel so aussehen:

Name	Wert

Zur Verwendung einer Speichertabelle:

1. Wenn eine Variable erstellt wird, fügen Sie sie als Zeile in die Tabelle ein (Variablenname in der Spalte "Name"; Variablenwert in der Spalte "Wert").
2. Wenn eine Variable aktualisiert wird (was Sie als Nächstes lernen werden), suchen Sie die Variable anhand ihres Namens, streichen Sie den vorherigen Wert durch und tragen Sie den neuen Wert ein.

Wenn also eine Variable erstellt wird, wird der Tabelle eine neue Zeile hinzugefügt. Wenn eine Variable aktualisiert wird, wird der vorherige Wert (in der Spalte "Wert") durchgestrichen und ein neuer Wert notiert. Ein Beispiel dafür sehen Sie auf der nächsten Seite.

3.1.2 Lesen einer Variablenaktualisierung

Nachdem wir eine Variable deklariert haben, können wir den Wert dieser Variable durch einen anderen Wert ersetzen. Hier ist ein Beispiel:

```
x = 1
y = 2
x = y
y = 3
```

In der ersten Zeile deklarieren wir eine Variable mit dem Variablennamen x und setzen sie auf den Integerwert 1. In der zweiten Zeile deklarieren wir eine weitere Variable y und setzen sie auf 2. Zeigen wir das in einer Speichertabelle mit Variablennamen und Werten:

Variablenname	Wert
x	1
y	2

In der 3. Zeile aktualisieren wir die Variable x so, dass sie auch dem Wert von y entspricht:

Variablenname	Wert
x	1 → 2
y	2

Die Variable x hat ihren früheren Wert 1 "gelöscht" und speichert nun den Wert 2.

In der 4. Zeile aktualisieren wir die Variable y so, dass sie 3 ist.

Variablenname	Wert
x	1 → 2
y	2 → 3

Lassen Sie uns das Aktualisieren von Variablen üben. Lassen Sie uns üben, wie man feststellt, ob Variablennamen gültig sind!

Bearbeiten Sie bitte **Übung 2.3.**

3.2 Einführung: Schreiben

Wenn Sie eine Variable aktualisieren, müssen Sie vor allem sicherstellen, dass die Variable, die Sie aktualisieren wollen, bereits deklariert wurde und dass Sie auf die richtige Variable verweisen! Ein häufiger, aber gefährlicher Fehler ist es, die falsche Variable zu aktualisieren, denn Ihr Code läuft weiter, obwohl Sie einen Fehler gemacht haben.

Ein Beispiel für eine korrekte Variablenaktualisierung:

```
amount_to_pay = 0.00
drink = ""

drink = "soda"
amount_to_pay = 1.00
drink = "juice"
amount_to_pay = 1.50
```

In diesem Beispiel deklarieren wir die Variablen `amount_to_pay` und `drink`. Die Variable `amount_to_pay` wird aktualisiert, nachdem wir `drink`, unsere Getränkewahl, ändern.

Den Datentyp einer Variable ändern? Normalerweise ist es eine gute Idee, in einer Variable stets denselben Datentyp zu speichern, auch wenn sie aktualisiert wird. Auf diese Weise ist es einfacher, den Überblick über die Variablen zu behalten.

Wenn Sie also eine Variable als String deklariert haben, sollte sie ein String bleiben. Wenn Sie eine Variable als boolean deklarieren, sollte sie boolean bleiben. Und wenn Sie eine Variable als Zahl deklariert haben, sollte sie auch eine Zahl bleiben.

```
cost = 1.50           #deklariere float Variable
item = "drink"        #deklariere string Variable
should_buy = False    #deklariere boolean Variable

cost = 1.00           #aktualisiere Variable
should_buy = True     #aktualisiere Variable
```

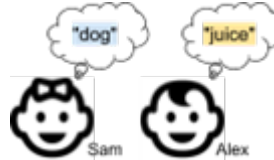
Beachten Sie, dass neben jeder Zeile des Codes ein Kommentar steht, der erklärt, was sie bewirkt!

Bearbeiten Sie bitte **Übung 2.4**.

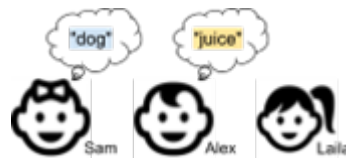
3.3 Template: Lesen von Variablenswaps

Sie haben gelernt, wie man Werte verschiedener Datentypen und Variablendeklarationen liest und schreibt. Was können wir nun mit diesem Wissen anfangen? Lassen Sie uns den Begriff des **Variablenswaps** anhand eines Beispiels einführen:

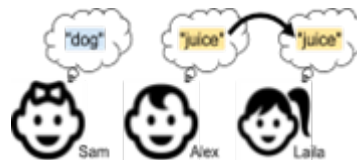
Nehmen wir an, wir haben 2 Babys, Sam und Alex. Jedes Baby kann jeweils nur ein einziges Wort denken.



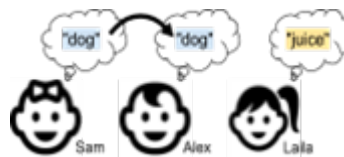
Sam und Alex wollen "Wörter tauschen", sodass sie am Ende an das Wort des anderen Babys denken. Aber sie können sich jeweils nur an ein Wort erinnern, wie können sie also Gedanken tauschen? Dazu können sie die Hilfe eines anderen Babys, Laila, in Anspruch nehmen!



Dieses neue Baby kann sich ein Wort vorübergehend merken, während die ersten 2 Babys ihre Gedanken austauschen! So teilt eines der ersten 2 Babys (z. B. Alex) seine Gedanken mit Laila:

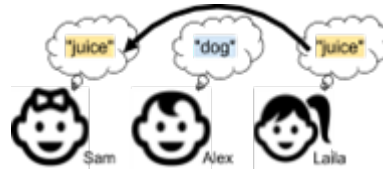


Da Laila nun Alex' ursprünglichen Gedanken speichert, muss sich Alex nicht mehr darum kümmern, sich daran zu erinnern. Sam kann nun ihre Gedanken mit Alex teilen:



3 Variablen

Jetzt hat Alex also Sams Gedanken! Jetzt braucht Sam nur noch den ursprünglichen Gedanken von Alex. Gut, dass Laila sich daran erinnert! Laila kann den ursprünglichen Gedanken von Alex ("juice") mit Sam teilen.



Dank der Hilfe von Laila konnten Sam und Alex ihre Gedanken austauschen! Wenn wir uns vorstellen, dass jedes Baby eine Variable ist, können wir die gleiche Logik auf den Austausch von Variablen anwenden.

Eine häufige Aufgabe, die wir erledigen wollen, ist das Vertauschen der Werte in zwei Variablen, sodass jede Variable den ursprünglichen Wert der anderen Variable speichert. Da der Code eine Zeile nach der anderen abläuft, gibt es keine Möglichkeit, die Variablen gleichzeitig zu vertauschen. So wie Sam und Alex die Hilfe eines dritten Babys, Laila, benötigen, müssen wir eine dritte temporäre Variable verwenden, um einen Wert während des Austauschs zu speichern.

Ein Swap hat also 2 Komponenten:

- 2 deklarierte Variablen, die aktualisiert werden.
- 1 temporäre Variable, die deklariert wird und den Wert einer der anderen Variablen speichert.

Schauen wir uns ein Beispiel für den Swap von Variablen an, bei dem wir die Namen des Gewinners und des Verlierers austauschen:

```
winner = "Abby"  
loser = "Julian"  
prev_winner = winner  
winner = loser  
loser = prev_winner
```

In den ersten 2 Zeilen deklarieren wir 2 Variablen:

Variablenname	Wert
winner	"Abby"
loser	"Julian"

3 Variablen

In der 3. Zeile deklarieren wir eine temporäre Variable (`prev_winner`) und setzen sie so, dass sie denselben Wert hat wie eine der beiden deklarierten Variablen (in diesem Fall `winner`).

Variablenname	Wert
<code>winner</code>	"Abby"
<code>loser</code>	"Julian"
<code>prev_winner</code>	"Abby"

Nachdem wir nun den vorherigen Wert des Gewinners gespeichert haben, gehen wir zur 4. Zeile, in der wir den Gewinner mit dem neuen Wert (dem aktuellen Wert des Verlierers) aktualisieren.

Variablenname	Wert
<code>winner</code>	" Abby " → "Julian"
<code>loser</code>	"Julian"
<code>prev_winner</code>	"Abby"

In der letzten Zeile aktualisieren wir die Variable `loser` mit dem ursprünglichen Wert von `winner`, der in unserer temporären Variable (`prev_winner`) gespeichert ist.

Variablenname	Wert
<code>winner</code>	" Abby " → "Julian"
<code>loser</code>	" Julian " → "Abby"
<code>prev_winner</code>	"Abby"

Üben wir das Lesen von Swaps!

Bearbeiten Sie bitte **Übung 2.5**.

3.4 Template: Variablen-Swaps schreiben

Da wir nun wissen, wie man Variablenswaps liest und erkennt, sollten wir uns darin üben, sie zu schreiben!

Um es noch einmal zu wiederholen: Das Ziel eines Swaps ist es, dass zwei Variablen Werte "tauschen".

Das Wichtigste beim Schreiben des Codes für einen Variablentausch ist, dass wir die Variablen in der richtigen Reihenfolge vertauschen. Wenn wir die Variablen nicht in der richtigen Reihenfolge vertauschen, entsprechen die beiden Variablen am Ende demselben Wert! So führen Sie einen korrekten Variablentausch durch:

1. Wir müssen zwei "permanente" Variablen deklarieren, deren Werte wir austauschen werden. Nennen wir sie `var1` und `var2`.
2. Wir setzen eine temporäre Variable, die den gleichen Wert wie eine der anderen Variablen hat. Nennen wir diese Variable `temp` und setzen sie auf denselben Wert wie `var2`.
3. Wir aktualisieren die Variable, deren Wert wir in der temporären Variable gesichert haben. Aktualisieren Sie also `var2` auf den gleichen Wert wie `var1`.
Da `temp` den ursprünglichen Wert von `var2` speichert, ist es in Ordnung, dass `var2` diesen Wert nicht mehr gespeichert hat.
4. Wir aktualisieren den Wert von `var1` so, dass er denselben Wert hat wie der in `temp` gespeicherte Wert (der ursprüngliche Wert von `var2`).

Wenn dies verwirrend ist, sehen Sie sich das Baby-Beispiel aus Reading Variable Swaps (ein paar Seiten zurück) an. Bezogen auf dieses Beispiel ist Sam `var1`, Alex ist `var2` und Laila ist `temp`.

Lasst uns üben, einen Swap zu schreiben!

Bearbeiten Sie bitte **Übung 2.6**.

4 Arithmetische Operationen

4.1 Einführung: Lesen

Viele der Dinge, die wir am Computer erleben (Filme ansehen, Tabellenkalkulationen verwenden usw.), beinhalten Arithmetik mit Zahlen (Ganzzahlen und Fließkommazahlen). Python bietet arithmetische Operatoren, die bei der Umsetzung dieser Verhaltensweisen helfen. Diese Operationen sollten Ihnen bekannt vorkommen, wenn Sie sich an den Mathematikunterricht erinnern. Es gibt allerdings einige subtile, aber wichtige Unterschiede im Umgang mit Mathematik in Python.

Hier sind die gängigen arithmetischen Operatoren in Python:

Operator	Erklärung	Beispiel
+ Addition	Addiert Werte auf beiden Seiten des Operators	$30 + 21$ (Ergebnis: 51) $4.1 + -1.0$ (Ergebnis: 3.1)
- Subtraktion	Subtrahiert den Wert auf der rechten Seite vom Wert auf der linken Seite des Operators	$5 - 2$ (Ergebnis: 3) $3.5 - 1.0$ (Ergebnis: 2.5)
* Multiplikation	Multipliziert Werte auf beiden Seiten des Operators	$3 * 2$ (Ergebnis: 6) $2.1 * 3$ (Ergebnis: 6.3)
/ Division	Dividiert den Wert auf der linken Seite durch den Wert auf der rechten Seite	$8 / 2$ (Ergebnis: 4.0) $4.4 / 2.0$ (Ergebnis: 2.2)
% Modulo	Dividiert den Wert auf der linken Seite durch den Wert auf der rechten Seite und gibt den Rest zurück	$4 \% 2$ (Ergebnis: 0) $5 \% 3$ (Ergebnis: 2)
// Integerdivison	Dividiert den Wert auf der linken Seite durch den Wert auf der rechten Seite und gibt den ganzzahligen Anteil zurück. Dieser gibt immer die niedrigere der beiden Zahlen aus.	$5 // 5$ (Ergebnis: 1) $3 // 2$ (Ergebnis: 1)

Der wohl am wenigsten bekannte arithmetische Operator ist der Modulo-Operator (%), mit dem wir den Rest einer Divisionsoperation bestimmen.

Die Reihenfolge, in der Operationen ausgeführt werden sollen, ist ein wichtiger Bestandteil der arithmetischen Operatoren. Im Matheunterricht haben Sie wahrscheinlich folgende Themen gelernt: Klammern, Exponenten, Multiplikation, Division, Addition und Subtraktion. Zum Glück gilt dies auch in der Programmierung. Die Modulo-Operation (%) steht in der gleichen Rang- oder Reihenfolge wie Multiplikation und Division. Wir rechnen also mit Klammern, dann mit Exponenten, dann mit Multiplikation, Division und Modulo, dann mit Addition und Subtraktion.

Im Beispiel $5 \% 2 + 1$ ergibt 2, weil Modulo eine Division ist, also berechnen wir zuerst den Modulo ($5 \% 2$ ergibt 1) und dann die Additionsoperation ($1 + 1$).

Bis hierhin sollte alles ziemlich vertraut sein. Aber denken Sie daran, dass es in Python zwei Typen von Zahlen gibt: Integer (Zahlen ohne Dezimalpunkt) und Float (Zahlen mit Dezimalpunkt). Wir sollten uns zwei Dinge merken:

1. Integer lassen den Dezimalwert weg; sie runden nicht.
2. Berechnungen mit einem Float ergeben immer einen Float.

Lassen Sie uns auf diese beiden wichtigen Punkte etwas näher eingehen:

4.1.1 Integer werden nicht gerundet!

Was ist 5 geteilt durch 3? Sie könnten 1,66 antworten. Oder vielleicht haben Sie im Mathematikunterricht gelernt, die Antwort auf 2 aufzurunden. Aber wenn Sie $5 // 3$ in Python eingeben, erhalten Sie als Ergebnis den Integer 1. Das liegt daran, dass **Integer in Python nicht gerundet werden. Stattdessen wird der Dezimalwert weggelassen.** Dies ist gleichbedeutend mit "Abrunden" oder der "Untergrenze" des Ergebnisses. Python lässt bei Integerdivisionen immer den Dezimalwert weg.

4.1.2 Weitere Hinweise: Typenzwang und Variablen

Manchmal werden wir Berechnungen durchführen, die sowohl Integer als auch Floats beinhalten. In diesem Fall ist das Ergebnis immer ein Float. $1.0 * 2$ würde also 2.0 ergeben.

Ein letzter Punkt ist, dass wir in unseren Berechnungen Variablen verwenden können, die Zahlen speichern. Hier ist ein Beispiel:

```
cost_juice = 1.50
cost_juice * 10
```

Das Ergebnis dieses Codebeispiels ist 15.0

Bearbeiten Sie bitte **Übung 3.1.**

4.2 Einführung: Schreiben

Wenn Sie Code mit arithmetischen Operatoren schreiben, ist es wichtig, dass Sie die Datentypen der Werte, mit denen Sie arbeiten, im Auge behalten. Ungeeignete Datentypen können dazu führen, dass Ihr Code nicht läuft oder dass Ihr Code läuft und ungewollte Ergebnisse ausgibt! Hier sind ein paar Dinge, die Sie beachten sollten:

1. **Arithmetische Operationen können nur mit Zahlen (Integer und Floats) durchgeführt werden.** Wir können keine arithmetischen Operationen mit anderen Datentypen als Zahlen durchführen. `3 + "4"` würde also dazu führen, dass der Code fehlschlägt, weil `"4"` keine Zahl ist (durch die Anführungszeichen ist es ein String).
2. **Arithmetische Operationen mit Floats und Integer verhalten sich unterschiedlich.** Wenn die Berechnungen nur Ganzzahlen umfassen, ist die Ausgabe ein Integer. Bei allen Berechnungen mit Floats wird ein Float ausgegeben. Dies ist wichtig, weil Integer nicht gerundet werden!

Bearbeiten Sie bitte **Übung 3.2.**

4.3 Template Lesen: Verarbeitung von Ziffern

Heutzutage verwenden wir numerische Passwörter, um viele Dinge zu sichern, z. B. Mobiltelefone, Debitkarten, Türöffnungen und Benutzerkonten. Die Überprüfung, ob ein numerisches Passwort, wie z. B. die vierstellige persönliche Identifikationsnummer (PIN) einer Debitkarte, korrekt ist, ist entscheidend für die Sicherheit.

Eine der einfachsten Möglichkeiten, die Gültigkeit einer Zahl zu überprüfen, besteht darin, nach einer zugrundeliegenden Beziehung zwischen den Ziffern einer Zahl zu suchen. Ein sehr einfaches Beispiel ist die Aussage, dass eine vierstellige PIN-Nummer gültig ist, wenn die zweite Ziffer größer ist als die dritte Ziffer. Dazu müssen wir in der Lage sein, einzelne Ziffern aus der PIN-Nummer zu extrahieren, die wir uns als eine Ganzzahl mit mehreren Ziffern vorstellen können. In diesem Abschnitt lernen Sie, wie Sie einzelne Ziffern aus einem mehrstelligen Integer extrahieren können.

Wenn wir einen Integer mit mehreren Ziffern haben (z. B. 23, 1234), können wir arithmetische Operatoren verwenden, um die ganze Zahl Ziffer für Ziffer zu betrachten. Dazu müssen wir bedenken, dass ganze Zahlen nicht gerundet werden; beim Multiplizieren oder Dividieren werden alle Nachkommastellen weggelassen. Wenn wir dies wissen und den Modulo-Operator verwenden, können wir wiederholt die Ziffer ganz rechts betrachten, sie vom ursprünglichen Wert entfernen und dann wieder die Ziffer ganz rechts betrachten.

Hier sind die Schritte zur Verarbeitung von Ziffern:

1. Beginnen Sie mit einem Integer als Eingabe.
2. Wiederholen Sie die folgenden Schritte, bis alle Ziffern abgearbeitet sind:
 - 2.1 Verwenden Sie den Modulo-Operator, um die letzte Ziffer zu ermitteln und diesen Wert zu speichern.
 - 2.2 Verwenden Sie den Divisionsoperator, um die letzte Ziffer zu entfernen.

Hier ein Beispiel, bei dem wir jede Ziffer des 3-stelligen Wertes in der Variablen `input` speichern:

```
input = 123
last_digit = input % 10
input = input // 10
second_digit = input % 10
input = input // 10
first_digit = input % 10
```

4 Arithmetische Operationen

Der erste Teil des Codes definiert die Eingabevariable:

Variablenname	Wert
<code>input</code>	123

Der nächste Teil des Codes verwendet den Modulo-Operator, um die letzte Ziffer zu ermitteln und speichert sie in der Variable `last_digit`. Anschließend wird die Variable `input` aktualisiert, indem `input` durch 10 dividiert wird.

Variablenname	Wert
<code>input</code>	123 \rightarrow 12
<code>last_digit</code>	3

Der nächste Teil des Codes wiederholt den Vorgang im vorherigen Teil des Codes, wobei der Modulo-Operator verwendet wird, um die nächste Ziffer zu erhalten, und dann die Division, um diesen Wert aus der Eingabe zu entfernen.

Variablenname	Wert
<code>input</code>	123 \rightarrow 12 \rightarrow 1
<code>last_digit</code>	3
<code>second_digit</code>	2

In der letzten Zeile des Codes wird die erste Ziffer gespeichert. Wir brauchen die Eingabe nicht mehr zu aktualisieren, da es keine weiteren Ziffern zu speichern gibt.

Variablenname	Wert
<code>input</code>	123 \rightarrow 12 \rightarrow 1
<code>last_digit</code>	3
<code>second_digit</code>	2
<code>first_digit</code>	1

Üben wir uns im Lesen von Code, der Ziffern verarbeitet!

Bearbeiten Sie bitte **Übung 3.3**.

4.4 Template Schreiben: Verarbeitung von Ziffern

Wir wollen das Template für die Verarbeitung von Ziffern immer dann verwenden, wenn wir auf eine bestimmte Ziffer in einer Ganzzahl mit mehreren Ziffern zugreifen wollen.

Plan zum Verarbeiten von Ziffern:

1. Sicherstellen, dass die Eingabe ein Integer ist
2. Für jede Ziffer in der Eingabe:
 - 2.1 Extrahieren Sie die Ziffer ganz rechts, indem Sie den Modulo 10 ($\% 10$) der Eingabe nehmen und in einer neuen Variablen speichern.
 - 2.2 Entfernen Sie die äußerste rechte Ziffer aus der Eingabe, indem Sie die Eingabe durch 10 dividieren und die Variable aktualisieren.

Denken Sie daran, den Eingabewert erst dann zu aktualisieren und die äußerste rechte Ziffer zu entfernen, wenn Sie diesen Wert gespeichert haben, sonst ist er endgültig verloren!

Lassen Sie uns üben, Code zu schreiben, der die Vorlage für die Verarbeitung von Ziffern verwendet!

Bearbeiten Sie bitte **Übung 3.4**.

5 Ausgabe

5.1 Lesen

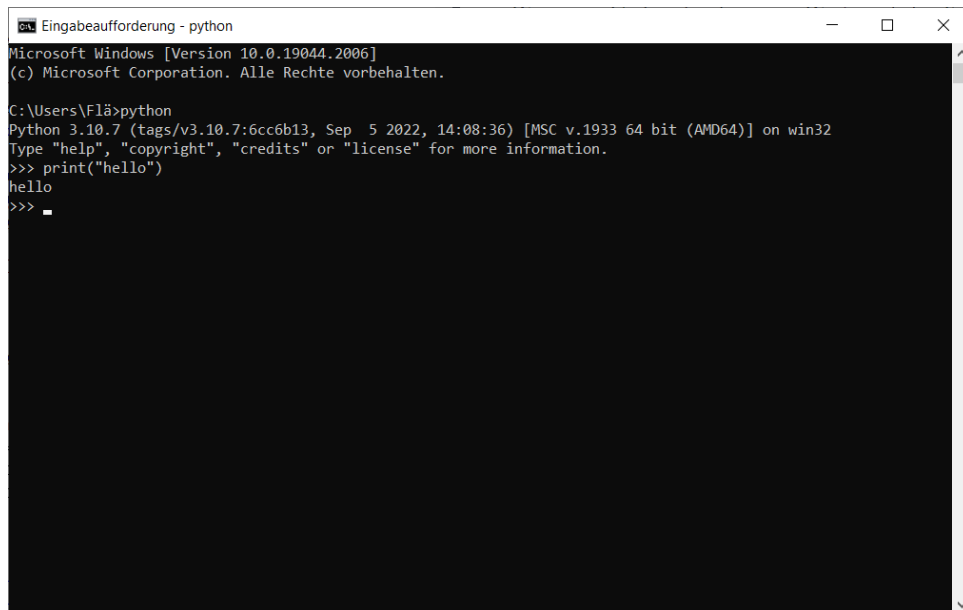
Bisher haben Sie die verschiedenen Datentypen kennengelernt, die Werte haben können. Sie haben gelernt, wie man Werte mit Variablen speichert, und Sie haben gelernt, mathematische Operationen mit Werten durchzuführen. Obwohl der Computer all diese Werte gespeichert hat, möchten wir manchmal die Werte auch sehen. Zu diesem Zweck müssen wir Ausgabeanweisungen verwenden.

Die Ausgabeanweisungen sendet eine Ausgabe an Ihren Monitor. Lassen Sie sich nicht vom Namen täuschen, Ausgabeanweisungen haben normalerweise nichts mit einem Drucker zu tun!

Eine Ausgabeanweisung ist eine Funktion, die in Python eingebaut ist. Eine Funktion ist ein Codeblock, der eine einzelne, wiederverwendbare Aktion ausführt. Wir können den wiederverwendbaren Code in einer Funktion ausführen, indem wir sie aufrufen. Wir rufen eine Funktion durch ihren Namen auf, gefolgt von einer Klammer, wobei alles in der Klammer ein Eingabewert ist, der in den Code der Funktion übergeben wird.

5 Ausgabe

Ein Beispiel für eine Ausgabeanweisung ist `print("hello!")`. Dies würde "hello!" ausgeben, wie in der folgenden Abbildung einer Konsole, einer Texteingabeanzeige auf einem Computer, zu sehen ist:



```
Eingabeaufforderung - python
Microsoft Windows [Version 10.0.19044.2006]
(c) Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Flä>python
Python 3.10.7 (tags/v3.10.7:6cc6b13, Sep 5 2022, 14:08:36) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello")
hello
>>> _
```

Hier wird ">>>" in der ersten Zeile nicht getippt; es ist nur die Konsole, die signalisiert, dass diese Zeile getippt wurde. Der Programmierer tippt also `print("hello!")` in die oberste Zeile und der Computer antwortet mit der zweiten Zeile "hallo!".

Hier sind 3 Regeln, die Sie bei der Ausgabe beachten sollten:

Regel	Beispiel
Operationen (z. B. Addition) in der Eingabe werden ausgeführt, bevor die Werte ausgegeben werden.	<code>print(1+2+3)</code> Ausgabe: 6
Wenn wir Variablen übergeben, geben wir die Werte der Variablen aus	<code>cost = 1.50</code> <code>print(cost)</code> Ausgabe: 1.5
Jede Anweisung wird in einer neuen Zeile ausgegeben.	<code>print("hi")</code> <code>print("bye")</code> Ausgabe: hi bye

Bearbeiten Sie bitte **Übung 4.1.**

5.2 Schreiben von Ausgabeanweisungen

Wenn wir Code schreiben, um Werte auf der Konsole auszugeben, verwenden wir eine Ausgabeanweisung, indem wir das Wort "print" gefolgt von dem, was wir ausgeben wollen, eingeben. Der auszugebende Wert wird in Klammern eingeschlossen.

Bei der Verwendung einer Ausgabeanweisung gibt es einige Dinge zu beachten, um sicherzustellen, dass Sie korrekten Code schreiben:

1. Den auszugebenden Wert in Klammern einschließen.
2. Wir können die Ausgabeanweisung nicht in einer Variablen speichern (das würde keinen Sinn machen!).

Hier sind einige Beispiele für Code mit Ausgabeanweisung, der NICHT ausgeführt werden würde:

Schlechter Code (würde nicht ausgeführt werden)	Korrigierter Code	Erklärung
<code>("hello")print</code>	<code>print("hello")</code>	Das Wort "print" steht vor dem auszugebenden Wert.
<code>print(1 + 3</code>	<code>print(1 + 3)</code>	Klammer nicht geschlossen.
<code>x = print(1)</code>	<code>x = 1 print(x)</code>	Eine Ausgabeanweisung kann nicht in einer Variablen gespeichert werden.

Üben wir uns im Schreiben von Ausgabeanweisungen!

Bearbeiten Sie bitte **Übung 4.2**.

6 Vergleichsoperatoren

6.1 Lesen: Vergleichsoperatoren

Sie wissen jetzt, welche Arten von Daten es gibt, wie man mit ihnen mathematische Operationen durchführt und wie man die Werte speichert und ausgibt. Eine häufige Aufgabe, die wir mit Werten lösen wollen, ist die Bestimmung ihrer Beziehung zueinander. So wollen wir zum Beispiel nur dann eine Jacke anziehen, wenn die Temperatur unter einem bestimmten Schwellenwert liegt. Oder wir entscheiden uns, einen Regenschirm mitzunehmen, wenn es regnet. Oder wir entscheiden uns dafür, jemanden nur dann in unser Haus zu lassen, wenn er das Passwort kennt. **Vergleichsoperatoren bestimmen die Beziehung zwischen Werten.**

Gängige Vergleichsoperatoren

Operator	Erklärung	Beispiel
<code>==</code> ist gleich	Bestimmt, ob 2 Werte genau gleich sind	<code>1 == 1</code> (wahr) <code>"A" == "A"</code> (wahr) <code>"A" == "a"</code> (falsch) <code>True == False</code> (falsch)
<code>!=</code> nicht gleich	Bestimmt, ob 2 Werte nicht genau gleich sind	<code>1 != 1</code> (falsch) <code>"A" != "A"</code> (falsch) <code>"A" != "a"</code> (wahr) <code>True != False</code> (wahr)
<code><</code> kleiner als	Bestimmt, ob der linke Wert kleiner ist als der rechte Wert	<code>1 < 1</code> (falsch) <code>-1 < 1</code> (wahr) <code>10 < 9</code> (falsch) <code>3.14 < 5</code> (wahr)
<code>></code> größer als	Bestimmt, ob der linke Wert größer ist als der rechte Wert	<code>1 > 1</code> (falsch) <code>-1 > 1</code> (falsch) <code>10 > 9</code> (wahr) <code>3.14 > 5</code> (falsch)
<code><=</code> kleiner als oder ist gleich	Bestimmt, ob der linke Wert kleiner oder gleich dem rechten Wert ist	<code>1 <= 1</code> (wahr) <code>-1 <= 1</code> (wahr) <code>10 <= 9</code> (falsch) <code>3.14 <= 5</code> (wahr)
<code>>=</code> größer als oder ist gleich	Bestimmt, ob der linke Wert größer oder gleich dem rechten Wert ist	<code>1 >= 1</code> (wahr) <code>-1 >= 1</code> (falsch) <code>10 >= 9</code> (wahr) <code>3.14 >= 5</code> (falsch)

Beachten Sie, dass bei Zeichenketten die Groß- und Kleinschreibung relevant ist, sodass "a" und "A" nicht gleich sind.

6.1.1 and-Operator: Zur Verknüpfung von Vergleichsoperatoren

Manchmal wollen wir mehrere Beziehungen gleichzeitig überprüfen. Zum Beispiel wollen wir nur spazieren gehen, wenn es wärmer als 15 Grad und kühler als 35 Grad ist. In Python würde diese zusammengesetzte Beziehung wie folgt aussehen: `temperature > 15 and temperature < 35`.

Der **and**-Operator gibt nur dann wahr zurück, wenn die Beziehungen auf der linken **und** rechten Seite wahr sind. Hier sind einige Beispiele:

Code	Ausgabe	Erklärung
<pre>temp = 60 print(temp > 50 and temp < 80)</pre>	True	Die Beziehungen auf beiden Seiten von and sind wahr, sodass die gesamte Bezeichnung wahr ist.
<pre>temp = 90 print(temp > 50) print(temp < 80) print(temp > 50 and temp < 80)</pre>	True False False	Da temp nicht < 80 ist, wird die 3. Druckanweisung als Falsch ausgewertet, da die Beziehung rechts von and falsch ist.
<pre>x = 1 y = 2 z = 3 print(y > x and y < z) print(x > y and x < z)</pre>	True False	Die erste Ausgabe ist wahr für die Beziehungen auf beiden Seiten von and . Die zweite Ausgabe ist False, weil die Beziehung auf der linken Seite von and False ist.

Üben wir uns im Lesen von Vergleichsoperatoren!

Bearbeiten Sie bitte **Übung 5.1**.

6.2 Schreiben von Vergleichsoperatoren

Ähnlich wie arithmetische Operatoren müssen auch Vergleichsoperatoren zwischen zwei Werten oder Variablen stehen. Die Werte oder Variablen, die Sie mit einem Vergleichsoperator vergleichen, sollten in der Regel vom gleichen Typ sein.

Beim Schreiben von Code, der Vergleichsoperatoren verwendet, sind einige Dinge zu beachten:

1. **= steht für die Zuweisung von Variablen;**
== steht für die Überprüfung der Gleichheit. Ein einfaches Gleichheitszeichen dient dazu, einer Variablen einen Wert zuzuweisen (z. B. `x = 1` weist der Variablen `x` den Wert 1 zu). Ein doppeltes Gleichheitszeichen dient zur Überprüfung der Gleichheit zweier Werte (z. B. `x == 1` bestimmt, ob der in der Variable `x` gespeicherte Wert gleich 1 ist).
2. **Bei der Gleichheit von Strings wird die Groß- und Kleinschreibung berücksichtigt.** Strings sind gleich, wenn sie genau die gleichen Zeichen enthalten. Groß- und Kleinbuchstaben werden als unterschiedliche Zeichen angesehen, daher müssen Strings die gleiche Groß-/Kleinschreibung aufweisen, um gleich zu sein.
3. **Es ist am besten, wenn die Werte vom gleichen Typ sind.** Der Vergleich von Werten unterschiedlichen Typs (z. B. der Vergleich eines Strings mit einem Integer) ist selten sinnvoll und kann oft zu unerwarteten Ergebnissen führen. Dieses seltsame Verhalten gilt insbesondere für den Vergleich von Integern und Floats.
4. **and** kann verwendet werden, um mehrere Beziehungen zu prüfen. Wenn wir nur True zurückgeben wollen, wenn mehrere Beziehungen alle zu True ausgewertet werden, können wir den Operator `and` zwischen die Beziehungen setzen.

Lassen Sie uns üben, wie man Code mit Vergleichsoperatoren schreibt.

Bearbeiten Sie bitte **Übung 5.2**.

6.3 Template: Lesen des Float-Gleichheits-Templates

Computer führen sehr genaue Berechnungen durch. Das ist oft eine gute Sache, denn wir wollen Präzision, wenn wir Dinge wie zum Beispiel die Flugbahn eines Satelliten um die Erde berechnen, bei der kleine Änderungen zu großen Unterschieden führen können. Diese Präzision kann aber auch zu einem seltsamen Verhalten führen. Sehen wir uns ein Beispiel dafür an:

Ist also 1.0 gleich 1.0000000000000001? Wären wir ein Kassierer, der Wechselgeld herausgibt, würden wir vielleicht denken, dass es sich um dieselbe Zahl handelt. Aber unser Mathelehrer würde uns wahrscheinlich sagen, dass diese Zahlen nicht gleich sind. Schauen wir mal, was Python denkt:

```
>>> 1.0 == 1.0000000000000001
False
```

Python ist sich einig, dass diese Floats nicht gleich sind. Aber ist 1.0 gleich 1.00000000000000001? (1 zusätzliche Null hinzugefügt). Unser Mathelehrer würde wahrscheinlich immer noch denken, dass sie nicht gleich sind. Mal sehen, ob Python das anders sieht:

```
>>> 1.0 == 1.00000000000000001
True
```

Interessant! Python glaubt, dass diese Werte gleich sind. Der Grund dafür ist die Art und Weise, wie der Computer Float-Werte speichert, und ein willkürlicher Grenzwert, den die meisten (aber nicht alle) Computer haben, um die Gleichheit zu bestimmen. Das ist nicht ideal, also sollten wir **NICHT den Gleichheitsoperator verwenden, um die Gleichheit von Floats zu prüfen.**

Ein besserer Weg, die Gleichheit von Floats zu prüfen, ist, festzulegen, dass Zahlen, die "nahe genug" beieinander liegen, gleich sind. Dabei müssen wir den Schwellenwert für "nahe genug" definieren. Dann prüfen wir, ob der Abstand oder die absolute Differenz zwischen zwei Floats kleiner ist als der von uns definierte Schwellenwert.

Float-Gleichheit bestimmt, ob Float-Werte nahe genug beieinander liegen, um als gleich zu gelten.

Um die Gleichheit der Floats zu bestimmen:

1. Definieren Sie einen Schwellenwert (kleiner positiver Float) und setzen Sie ihn auf eine Variable.
2. Berechnen Sie die absolute Differenz zwischen zwei Zahlen (Ganzzahlen oder Gleitkommazahlen) mithilfe der Funktion `abs()`.
3. Bestimmen Sie, ob die absolute Differenz kleiner als der Schwellenwert ist.

Sehen wir uns ein Beispiel für die Berechnung der Float-Gleichheit an:

In diesem Beispiel haben wir 3 Floats (gespeichert in den Variablen `f1`, `f2`, `f3`) und wollen feststellen, ob sie gleich sind.

```
f1 = 1.111
f2 = 1.112
f3 = 1.1112
```

Zu diesem Zweck legen wir zunächst einen Schwellenwert fest. Floats, die weniger als diesen Schwellenwert voneinander entfernt sind, werden als gleich angesehen. Wir deklarieren eine Variable `threshold` und setzen sie auf `0.001`.

```
f1 = 1.111
f2 = 1.112
f3 = 1.1112
```

```
threshold = 0.001
```

Nun prüfen wir, ob die Floats gleich sind. Dazu prüfen wir, ob der absolute Abstand zwischen den Floats kleiner ist als der von uns festgelegte Schwellenwert. Um den absoluten Wert zu erhalten, können wir die Funktion `abs()` verwenden, die den absoluten Wert der Eingabe zurückgibt. So wäre `abs(1)` gleich 1 und `abs(1.1)` gleich 1.1. Wir nehmen also den absoluten Wert der Differenz von 2 Floats und prüfen, ob er kleiner als der Schwellenwert ist. Wir prüfen die Gleichheit zwischen `f1` und `f2` sowie zwischen `f1` und `f3` und geben das Ergebnis aus.

6 Vergleichsoperatoren

```
f1 = 1.111
f2 = 1.112
f3 = 1.1112

threshold = 0.001

print(abs(f2 - f1) < threshold)
print(abs(f3 - f1) < threshold)
```

Die Ausgabe lautet:

False
True

Bei dem von uns festgelegten Schwellenwert sind `f1` und `f2` also nicht gleich und `f1` und `f3` sind gleich.

Üben wir uns nun im Lesen von Float-Vergleichen.

Bearbeiten Sie bitte **Übung 5.3**.

6.4 Schreiben des Float-Gleichheits-Templates

Wir verwenden das Float-Gleichheits-Template immer dann, wenn wir die Beziehung zwischen Zahlen vergleichen wollen, die Floats sein könnten. Das bedeutet umgekehrt, dass eine oder beide dieser Zahlen eine ganze Zahl sein könnten und diese Gleichheitsprüfung immer noch gültig wäre. Das heißt, `3 == 3` wäre das gleiche Ergebnis wie `abs(3, 3) < 0.0001`.

Die Schritte zur Überprüfung der Gleichheit von float und die häufigsten Fehler bei jedem Schritt:

Schritt	Häufiger Fehler	Beispiel eines Fehlers
2 Werte vergleichen, die Zahlen sind (Floats oder Integer).	Vergleich von Werten, die keine Zahlen sind. Dies wäre ein Fehler und Ihr Code würde nicht laufen.	<code>1.0003 == "1.0004"</code>
Eine kleine positive Zahl als Schwellenwert definieren.	Definieren von 0 oder einer negativen Zahl als Schwellenwert. Ihr Code würde trotzdem laufen, und die Prüfung der Gleichheit von Floats würde immer zu einem falschen Ergebnis führen.	<pre>x = 1.0003 y = 1.0004 threshold = 0.0 abs(x-y) < threshold</pre>
Bestimmen Sie, ob die absolute Differenz kleiner als der Schwellenwert ist.	<code>abs()</code> nicht zur Prüfung der absoluten Differenz nutzen. Ohne diese Funktion wäre es möglich, eine negative Zahl zu erhalten, was dazu führen würde, dass Ihr Code läuft und <code>True</code> ausgibt, obwohl er das nicht soll!	<pre>x = 1.0003 y = 1.0004 threshold = 0.0001 x-y < threshold</pre>

Lassen Sie uns das Schreiben von Code mit Float-Gleichheits-Templates üben!

Bearbeiten Sie bitte **Übung 5.4**.

7 Bedingte Anweisungen

7.1 Lesen von Bedingten Anweisungen

Da wir nun wissen, wie wir die Beziehung zwischen Werten mit relationalen Operatoren bestimmen können, wollen wir verschiedene Dinge auf der Grundlage verschiedener Beziehungen tun. Ein Beispiel: Wenn es draußen kalt ist, nehme ich meinen Mantel mit. Oder wenn ich heute Abend keine Hausaufgaben habe, dann werde ich mich mit meinen Freunden treffen.

Bedingte Anweisungen (auch *if-Anweisungen* genannt) ermöglichen die Ausführung unterschiedlichen Codes in Abhängigkeit von einer bestimmten Beziehung.

Angenommen, Sie möchten eine Cola kaufen, würden dies aber nur tun, wenn sie höchstens 1 € kostet. Sie können eine bedingte Anweisung verwenden, um diese Entscheidung zu treffen. Nehmen wir an, die Variable `cost` ist ein Float und wurde zuvor deklariert. Um festzustellen, ob wir die Cola kaufen sollen, schreiben wir folgenden Code:

```
if cost <= 1.00:
    print("Kaufe die Cola!")
```

Wir haben also das Wort "if", gefolgt von einem Vergleichsoperator, bekannt als **Bedingte Anweisung** (`cost <= 1.00`). Wenn diese Bedingung als wahr bewertet wird, wird der gesamte eingerückte Code unter der if-Anweisung (in diesem Fall die Ausgabeanweisung) ausgeführt. Ergibt die Bedingung nicht den Wert "wahr" (also "falsch"), wird der eingerückte Code nicht ausgeführt.

Wenn wir einen anderen Code ausführen wollten, wenn die Bedingung falsch ist, würden wir eine else-Bedingung hinzufügen:

```
if cost <= 1.00:
    print("Kaufe die Cola!")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

7 Bedingte Anweisungen

Wenn die Bedingung als falsch ausgewertet wird (die Kosten für die Cola sind größer als 1.00), dann wird der gesamte Code unter der else-Anweisung (2 Ausgabeanweisungen) ausgeführt.

Zusammenfassend lässt sich also sagen, dass bedingte Anweisungen den eingerückten Code unter der if-Anweisung ausführen, wenn die Bedingung erfüllt ist. Wenn die Bedingung nicht erfüllt ist, überspringt der Computer den eingerückten Code unter der if-Anweisung und führt stattdessen den eingerückten Code unter der else-Anweisung aus.

Gehen wir diesen Code einmal durch:

```
cost = 1.25
if cost <= 1.00:
    print("Kaufe die Cola!")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

Hier wird die Variable cost auf 1.25 gesetzt. Wir überprüfen dann die if-Anweisung und stellen fest, dass sie falsch ist. Also überspringen wir den eingerückten Code unter der if-Anweisung und springen zur else-Anweisung. Wir führen dann den eingerückten Code unter der else-Anweisung aus.

Die Ausgabe der Ausführung dieses Codes:

Kaufe die Cola nicht!
Ich wiederhole, kaufe sie nicht!

Hier ist derselbe Codeblock mit gelb unterstrichenen Codezeilen, um die Codezeilen anzuzeigen, die ausgeführt werden, und einer nicht markierten Zeile, um die Codezeile anzuzeigen, die nicht ausgeführt wird.

```
cost = 1.25
if cost ≤ 1.00 :
    print("Kaufe die Cola!")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

Ändern wir die Kosten für die Cola und betrachten wir das Beispiel erneut:

```
cost = 0.75
if cost <= 1.00:
    print("Kaufe die Cola!")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

Hier sind die Kosten so niedrig, dass die Bedingung in der if-Anweisung erfüllt ist. Daher wird die Ausgabeanweisung, die unter der if-Anweisung eingerückt ist, ausgeführt und die else-Anweisung übersprungen.

Die Ausgabe der Ausführung dieses Codes: Kaufe die Cola!

Der kommentierte Codeblock zeigt, dass der eingerückte Code unter der else-Bedingung übersprungen wurde.

```
cost = 0.75
if cost ≤ 1.00 :
    print("Kaufe die Cola!")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

7.1.1 Else-if-Anweisungen: wenn es mehr als 2 Optionen gibt

Einfache Bedingte Anweisungen mit einer if- und else-Bedingung helfen uns, Entscheidungen zwischen zwei Optionen zu treffen. Was aber, wenn wir eine 3. Option einführen wollen? Dafür brauchen wir else if-Anweisungen.

Nehmen wir an, Sie haben einen Freund, der Ihnen 0.50 € leihen kann. Wenn die Cola also bis zu 1.50 € kostet, kannst du sie dir immer noch leisten. Aber wenn die Cola mehr als 1 € und weniger als oder maximal 1.50 € kostet, müssen Sie Ihren Freund um das Geld bitten. Damit wird eine dritte Möglichkeit eingeführt. Um diese dritte Option zu berücksichtigen, können wir eine else-if-Anweisung einführen.

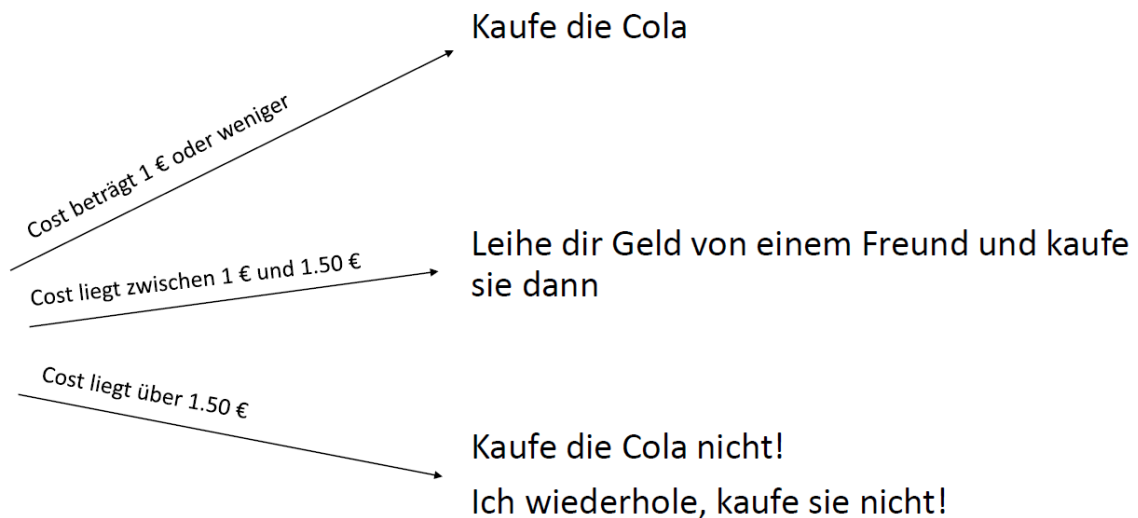
Else-if-Anweisungen stehen nach der if-Anweisung und vor der else-Anweisung. Wenn die if-Bedingung als falsch bewertet wird, wird die else-if-Bedingung geprüft. Wenn sie wahr ist, wird der eingerückte Code unter der elseif-Bedingung ausgewertet

und die else-Bedingung wird übersprungen. Wenn die elseif-Bedingung ebenfalls als falsch bewertet wird, wird die else-Bedingung ausgeführt.

Sehen wir uns ein Beispiel an:

```
cost = 1.50
if cost <= 1.00:
    print("Kaufe die Cola!")
elif cost <= 1.50:
    print("Leihe dir Geld von einem Freund und kaufe sie dann.")
else:
    print("Kaufe die Cola nicht!")
    print("Ich wiederhole, kaufe sie nicht!")
```

Dieser Code entspricht dem nachstehenden Diagramm, in dem die Kosten des Sodas bestimmen, welcher eingerückte Codeblock ausgeführt wird. Beachten Sie, dass in jedem Fall nur 1 Bedingung ausgeführt wird. Wenn also eine Cola 0.75 € kostet, wird nur Kaufe die Cola! ausgegeben, obwohl bei einem Preis von 0.75 € auch die Bedingung else if wahr wäre. Das liegt daran, dass in jeder bedingten Anweisung höchstens 1 Bedingung ausgeführt wird.



Bearbeiten Sie bitte **Übung 6.1.**

7.2 Schreiben von Bedingten Anweisungen

Beim Schreiben Bedingter Anweisungen sollten wir darauf achten, dass wir den Code korrekt schreiben, aber auch darauf, dass die Logik in den Bedingten Anweisungen korrekt ist.

Um eine if-Anweisung korrekt zu schreiben, sind einige Dinge zu beachten:

Regel	Fehlerhafter Code	Korreakter Code
Die Bedingung in einer if-Anweisung hat einen Doppelpunkt am Ende der Zeile.	<pre>if x < 2 print("yes")</pre>	<pre>if x < 2 : print("yes")</pre>
Einrücken des Codes, der ausgeführt werden soll, wenn eine Bedingung erfüllt ist.	<pre>if x < 2: print("yes")</pre>	<pre>if x < 2: print("yes")</pre>
Andere Bedingungen (z. B. <code>elif</code> , <code>else</code>) für eine bestimmte if-Anweisung sollten auf der gleichen Einrückungsebene stehen.	<pre>if x < 2: print("yes") elif x < 5: print("ok") else: print("too big")</pre>	<pre>if x < 2: print("yes") elif(x < 5): print("ok") else: print("too big")</pre>
Else-if-Anweisungen beginnen mit <code>elif</code> .	<pre>if x < 2: print("yes") else if x < 5: print("ok")</pre>	<pre>if x < 2: print("yes") elif x < 5: print("ok")</pre>

7 Bedingte Anweisungen

Beim Schreiben von bedingten Anweisungen müssen wir vorsichtig sein, denn kleine Änderungen können die Funktionsweise unserer bedingten Anweisungen beeinflussen. Hier sind einige Dinge zu beachten:

Regel	Gefährlicher Code	Erklärung
Jede if-Anweisung wird ausgeführt.	<pre>cost = 0.45 if cost < 0.5: print("Kauf es!") if cost < 0.75: print("Leih dir Geld, um es zu kaufen!") else: print("Kauf es nicht.")</pre>	In diesem Beispiel werden 2 Zeichenketten ausgegeben, weil die bedingte Anweisung für 2 if-Anweisungen wahr ist. Das Ersetzen der zweiten if-Anweisung durch eine elif-Anweisung würde diesen logischen Fehler beheben.
If-Anweisungen sind erforderlich. Elseif- und else-Anweisungen sind nicht erforderlich.	<pre>name = "sue" elif name == "bob": print("Bob ist da!") else: print("Bob ist nicht da!")</pre>	Dieser Code würde nicht laufen, denn es gibt keine if-Anweisung zum Starten der Bedingung. Ersetzt man das elif durch ein if, wird der Code ausgeführt.
Wir können den and-Operator mit Bedingten Anweisungen verwenden.	<pre>will_rain = True going_outside = False if(will_rain and going_outside): print("Zieh eine Regenjacke an!") else: print("Keine Sorge.")</pre>	In diesem Beispiel wird "keine Sorge" ausgegeben, weil going_outside falsch ist, also wird die else-Anweisung ausgeführt.

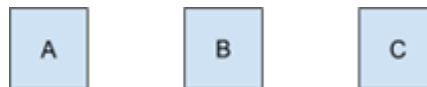
Üben wir uns im Schreiben von Bedingten Anweisungen!

Bearbeiten Sie bitte **Übung 6.2.**

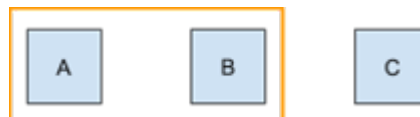
7.3 Template-Lesen: Minimum oder Maximum finden

Angenommen, wir haben Zahlen und wollen die kleinste Zahl ausgeben. Wir können ein einfaches if/else verwenden, um dies für 2 Zahlen zu tun. Was aber, wenn wir mehr als 2 Zahlen haben? Hierfür können wir das, was wir über den Operator and und die if-Anweisungen und else if-Anweisungen wissen, verwenden. Schauen wir uns ein konzeptionelles Beispiel an:

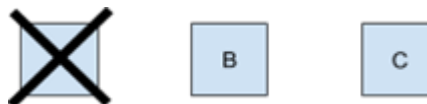
Angenommen, ich hätte 3 Variablen, in denen Zahlen gespeichert sind, und möchte den größten Wert herausfinden, aber ich könnte nur höchstens 2 Variablen auf einmal betrachten. Wie könnte ich das tun?



Nun, ich könnte damit beginnen, A und B zu vergleichen.



Wenn ich sehe, dass A größer ist als B, dann sollte ich auch sehen, ob A größer ist als C. Wenn A größer ist als B und C, dann bin ich fertig. Wenn A nicht größer ist als B und C, dann weiß ich, dass der größte Wert nicht A ist und kann A ignorieren.



Mein nächster Schritt ist der Vergleich von B und C.



Wenn B größer ist als C, dann weiß ich, dass B die größte Zahl ist. Ich brauche A und B nicht noch einmal zu vergleichen, weil ich A schon vorher überprüft habe! Wenn wir feststellen, dass B nicht größer ist als C, können wir sagen, dass B nicht die größte Zahl ist, und es entfernen.



Wenn wir herausgefunden haben, dass die größte Zahl nicht A oder B ist, dann muss es C sein! Wir haben mit 3 Zahlen begonnen und mussten 1 Zahl mit den anderen 2 vergleichen. Wenn sie nicht der gesuchten Zahl entsprach, haben wir sie entfernt. Dann haben wir 1 der verbleibenden Zahlen mit der anderen 1 verglichen. Wenn es nicht die gesuchte Zahl war, haben wir sie entfernt. Dann blieb nur noch 1 Zahl übrig und 0 Zahlen, mit denen wir vergleichen konnten. Wenn das der Fall war, dann muss diese letzte Zahl die gesuchte sein.

Durch dieses Verfahren, bei dem ein Wert geprüft und dann von künftigen Prüfungen ausgeschlossen wird, und durch Wiederholung dieses Vorgangs mit den verbleibenden Zahlen wird die maximale oder minimale Zahl aus mehreren Optionen ermittelt. Um dies im Code zu sehen:

```
if a > b and a > c:
    print(a)
elif b > c:
    print(b)
else:
    print(c)
```

Ähnlich wie im Beispiel auf der vorigen Seite haben wir den Operator **and** verwendet, um festzustellen, ob a die größte Zahl ist. Ist dies nicht der Fall, können wir es ignorieren und mit der else if-Bedingung weitermachen, um zu sehen, ob b die größte Zahl ist. Wenn dies nicht der Fall ist, dann wissen wir in der else-Bedingung, dass c die größte Zahl ist.

Wenn wir alle > Zeichen durch < Zeichen ersetzen, können wir dasselbe Template verwenden, um den Mindestwert zu finden!

7 Bedingte Anweisungen

Um den maximalen (größten) oder minimalen (kleinsten) Wert zu finden, gehen wir folgendermaßen vor:

1. Verwenden Sie if-Anweisungen, um 1 Wert mit allen anderen verbleibenden Werten zu vergleichen.
 - 1.1 Möglicherweise benötigen wir eine zusammengesetzte bedingte Anweisung (mit **and**).
2. Ignorieren Sie den gerade geprüften Wert und wiederholen Sie Schritt 1, wenn es mindestens 2 verbleibende Werte gibt.
3. Wenn es keine weiteren Werte zum Vergleich gibt, erreichen wir unsere else-Bedingung.

Üben wir das Lesen des Min- oder Maxtemplates!

Bearbeiten Sie bitte **Übung 6.3**.

7.4 Schreiben: Min- oder Maxtemplate

Beim Schreiben von Code zur Ermittlung des Höchst- oder Mindestwerts sind einige Dinge zu beachten:

Regel	Schlechter Code	Erklärung
Vergleichen Sie einen Wert mit allen Werten, die NICHT eliminiert wurden.	<pre>if a > b and a > c: print(a) elif b > c and b > a: print(b) else: print(c)</pre>	Die Variable a wurde bereits eliminiert und muss nicht erneut verglichen werden. Dies führt zu unnötiger Verwirrung.
Wenn mehr als 2 Werte zu vergleichen sind, müssen Sie if-Anweisungen mit and kombinieren.	<pre>if a > b: if a > c: print(a) elif b > c: print(b) else: print(c)</pre>	Der fettgedruckte Code sollte als zusammengesetzte Bedingung in die darüber stehende if-Anweisung aufgenommen werden: if a > b and a > c:
Stellen Sie sicher, dass die bedingte Anweisung mit der auszugebenden Variable übereinstimmt.	<pre>if a > b and a > c: print(c) elif b > c: print(b) else: print(c)</pre>	Die erste if-Anweisung prüft, ob die Variable a das Maximum ist, aber es wird die Variable c ausgegeben.

Bearbeiten Sie bitte **Übung 6.4**.

8 for-Schleifen

8.1 Einführung: Lesen

Nachdem wir Bedingte Anweisungen kennengelernt und verstanden haben, wollen wir nun Codestücke mehrmals ausführen lassen. Dafür gibt es verschiedene Arten von Schleifen. Im folgenden Kapitel betrachten wir for-Schleifen.

for-Schleifen ermöglichen, die mehrmalige Ausführung eines Codestücks.

Angenommen, Sie möchten 10x "meow" ausgeben, dann könnten Sie 10 Mal den Code `print("meow")` schreiben. Dies wäre aber ineffizient. Außerdem würde es ab einer gewissen Menge unmöglich werden, so viele Ausgaben zu schreiben. Deswegen nutzen wir für die mehrmalige Ausführung eines Codestücks eine Schleife. Anhand des Flussdiagramms 10.1 auf der nächsten Seite sehen Sie den Ablauf einer for-Schleife. Zu Beginn wird unser Iterator, also die Variable, mithilfe der wir durch unsere Schleife laufen, auf 0 gesetzt. Dies geschieht in Python implizit.

In dem Beispiel verwenden wir als Variable den Buchstaben `i`, der eine geläufige Abkürzung für den Iterator einer Schleife darstellt. Da wir 10x "meow" ausgeben möchten, haben wir die Abbruchbedingung

`i < 10`. Dies wird in Python nur implizit angegeben, durch die Abbruchbedingung der for-Schleife (`range(10)`). Falls diese Bedingung noch nicht erreicht und demnach wahr ist, erhöhen wir `i` um 1, was in Python wieder nur implizit stattfindet. Falls die Abbruchbedingung erreicht worden ist, stoppt unsere for-Schleife und der Code, der innerhalb der for-Schleife eingerückt worden ist, wird nicht mehr ausgeführt. Ohne diese Abbruchbedingung würde die Schleife unendlich lang laufen und sich das Programm schlussendlich aufhängen.

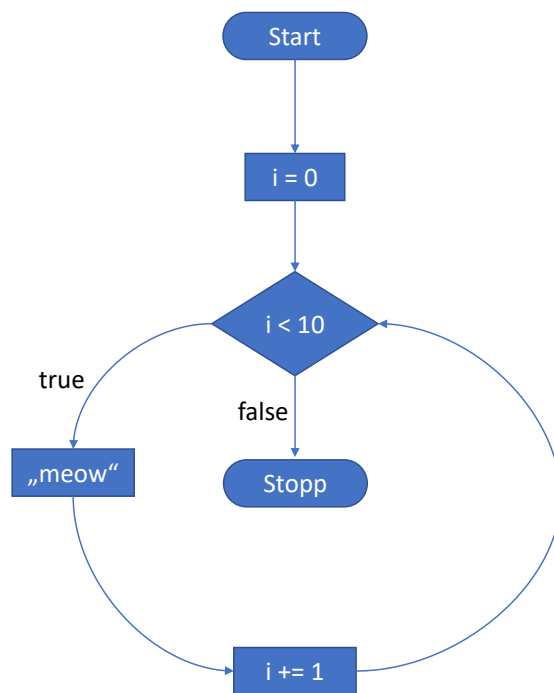


Abbildung 8.1: Flussdiagramm zur Funktionsweise einer for-Schleife.

Der dazugehörige Code sieht wie folgt aus:

```
for i in range(10):
    print("meow")
```

Wir benötigen also das Schlüsselwort `for`, gefolgt von einem Variablennamen, der als Iterator genutzt wird und `in range`. Standardmäßig startet `in range` bei 0 und mit einem Iterationsschritt von 1 und wird beendet, wenn unser Iterator im nächsten Schritt dem Endwert (in diesem Fall 10) entsprechen würde.

Wir können aber nicht nur Strings mehrfach ausgeben, sondern auch unseren Iterator. Die Ausgabe der Zahlen von 0 bis einschließlich 5 würde mithilfe einer for-Schleife folgendermaßen aussehen:

```
for number in range(6):
    print(number)
```

In der Einführung zu diesem Kapitel haben Sie bereits gelernt, dass der Startwert als auch der Iterationsschritt implizit auf 0 bzw. 1 gesetzt werden. Wenn wir die Ausgabe mit der Zahl 1 starten wollen würden, dann sieht der Code so aus:

```
for number in range(1, 6):  
    print(number)
```

Falls wir die Zahlen von 1 bis 5 addieren möchten, dann würde unser Code so aussehen:

```
result = 0  
for number in range(1, 6):  
    result += number  
print(result)
```

Die Ausgabe der Ausführung dieses Codes: 15

Ändern wir den Wert, um den iteriert werden soll und betrachten wir das Beispiel erneut. Hierbei ist der Startwert 1, der Endwert 6 und der Iterationsschritt entspricht 2:

```
result = 0  
for number in range(1, 6, 2):  
    result += number  
print(result)
```

Die Ausgabe der Ausführung dieses Codes: 9

Zusammenfassend lässt sich also sagen, dass for-Schleifen genutzt werden sollten, wenn etwas definiert oft stattfinden soll.

Lassen Sie uns das Lesen von for-Schleifen üben!

Bearbeiten Sie bitte **Übung 7.1** und **7.2**.

8.2 Einführung: Schreiben

Beim Schreiben von for-Schleifen sollten wir darauf achten, dass wir syntaktisch korrekten Code schreiben, aber auch darauf, dass die Logik der for-Schleife korrekt ist.

Um eine for-Schleife syntaktisch korrekt zu schreiben, sind einige Dinge zu beachten:

Regel	Kaputter Code	Korrekter Code
Die Länge der range-Funktion wird durch zwei runde Klammern eingeschlossen	<pre>for i in range[5]: print(i)</pre>	<pre>for i in range (5) print(i)</pre>
Der Schleifenkopf hat einen Doppelpunkt am Ende der Zeile	<pre>for i in range(5) print(i)</pre>	<pre>for i in range(5) : print(i)</pre>
Einrücken des Codes, der ausgeführt werden soll, wenn eine Schleife ausgeführt werden soll	<pre>for i in range(5): print(i)</pre>	<pre>for i in range(5): print(i)</pre>

Üben wir uns im Schreiben von for-Schleifen!

Bearbeiten Sie bitte **Übung 7.3, 7.4 und 7.5.**

8.3 Lesen: Primzahlprüfung

Nachdem Sie gelernt haben, wie for-Schleifen grundlegend funktionieren, beschäftigen wir uns nun mit einem Template, welches prüft, ob es sich bei einer Zahl um eine Primzahl handelt. In der Informatik werden Primzahlen beispielsweise für die Verschlüsselung eingesetzt. Das RSA-Verfahren ist ein asymmetrisches Verfahren, das für die Verschlüsselung einen privaten und einen öffentlichen Schlüssel benutzt. Der private Schlüssel wird zum Entschlüsseln genutzt und der öffentliche zum Verschlüsseln. Beide Schlüssel bestehen dabei unter anderem aus dem Ergebnis der Multiplikation zweier Primzahlen. Die Berechnung dieser Primzahlen ist nicht aufwendig. Aber die Bestimmung des Multiplikators und des Multiplikanden dauert selbst mit den leistungstärksten Computern so lange, dass das Verfahren als sicher gilt. Dafür würde eine Primfaktorzerlegung bzw. der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen eingesetzt werden.

Nach der Einführung in ein theoretisches Beispiel aus der Kryptographie, befassen wir uns jetzt mit unserem Template zur Primzahlprüfung: Angenommen, wir möchten überprüfen, ob eine Zahl eine Primzahl ist. Dann müssen wir uns zuerst überlegen, wie eine Primzahl definiert ist. Eine Primzahl ist eine natürliche Zahl, die größer als 1 und ausschließlich durch sich selbst und durch 1 teilbar ist. Zur Bestimmung benötigen wir einen boolean, der zu Beginn auf wahr gesetzt wird. Danach laufen wir für jede einzelne Zahl alle Werte von 2 bis zum Wert der Zahl durch und überprüfen, ob dieser Iterator durch die Zahl teilbar ist. Falls die Zahl durch den Iterator teilbar ist, dann handelt es sich nicht um eine Primzahl. Falls es sich bei der Zahl um eine Primzahl handelt, dann geben wir diese nach der Schleife aus.

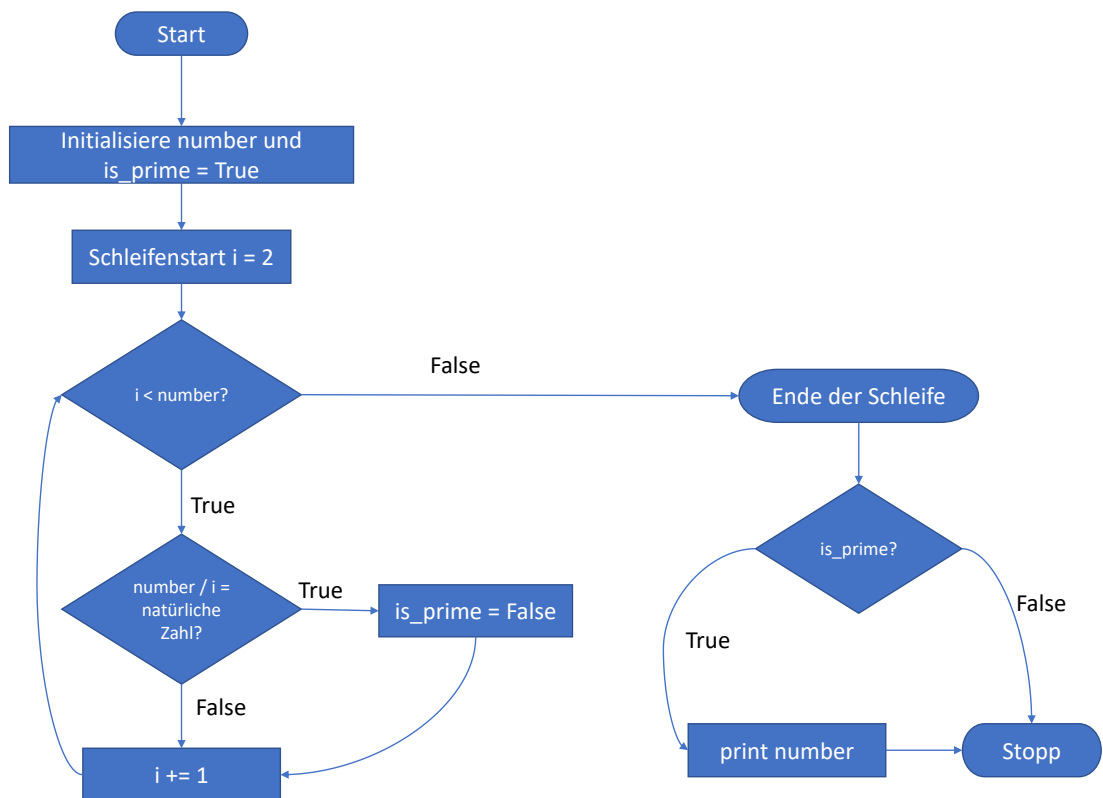


Abbildung 8.2: Flussdiagramm für die Primzahlprüfung.

Nach der Betrachtung des Flussdiagramms, sehen Sie hier den Code, der überprüft, ob es sich bei einer Zahl um eine Primzahl handelt und falls dies wahr ist, diese ausgibt:

```

number = 7
is_prime = True

for i in range(2, number):
    if number % i == 0:
        is_prime = False

if is_prime:
    print(number)
  
```

Bearbeiten Sie bitte **Übung 7.6**.

8.4 Schreiben: Primzahlprüfung

Beim Schreiben von Code zur Bestimmung, ob eine Zahl einer Primzahl entspricht, sind einige Dinge zu beachten:

Regel	Nicht korrekter Code	Erklärung
Wir benötigen einen boolean, um zu überprüfen, ob es sich bei der Nummer um eine Primzahl handelt.	<pre>is_prime = False number = 7 for i in range(2, 7): if number%i == 0: is_prime = False if is_prime: print(number)</pre>	In dem Beispiel ist der boolean zu Beginn auf False gesetzt und es wird nur geprüft, wann eine Zahl keiner Primzahl entspricht. Demnach ist der Boolean permanent auf False gesetzt, auch wenn es sich um eine Primzahl handelt.
Wir müssen abfangen, ob die Zahl geteilt durch i keinen Rest enthält.	<pre>is_prime = True number = 7 for i in range(2, 7): if number%i != 0: is_prime = False if is_prime: print(number)</pre>	In dem Fall fangen wir nicht ab, ob es keinen Rest gibt, sondern ob es einen Rest gibt. Demnach ist unsere Bedingte Anweisung nicht korrekt und wir erhalten ein falsches Ergebnis.
Wir möchten nach der Prüfung die geprüfte Zahl ausgeben.	<pre>is_prime = True number = 7 for i in range(2, 7): if number%i == 0: is_prime = False if is_prime: print(i)</pre>	Wir geben nach dem Durchlauf eine falsche Zahl aus.

Bearbeiten Sie bitte **Übung 7.7**.

9 while-Schleifen

9.1 Einführung: Lesen

In diesem Kapitel werden Sie den zweiten Schleifentypen in Python kennenlernen, und zwar while-Schleifen.

while-Schleifen sind ein Programmierkonstrukt, das erlaubt eine Frage mehrmals zu stellen, bis eine bestimmte Bedingung abgearbeitet worden ist. Angenommen, Sie möchten die Zahlen von 3 bis 0 ausgeben, dann könnten Sie 3 Mal eine Ausgabe mit den jeweiligen Zahlen schreiben. Dies wäre aber ineffizient. Außerdem würde es ab einer gewissen Zahlengröße unmöglich werden, so viele Ausgaben zu schreiben. Wenn wir eine while-Schleife nutzen, dann würde die Ausgabe der Zahlen von 3 bis 0 folgendermaßen aussehen.

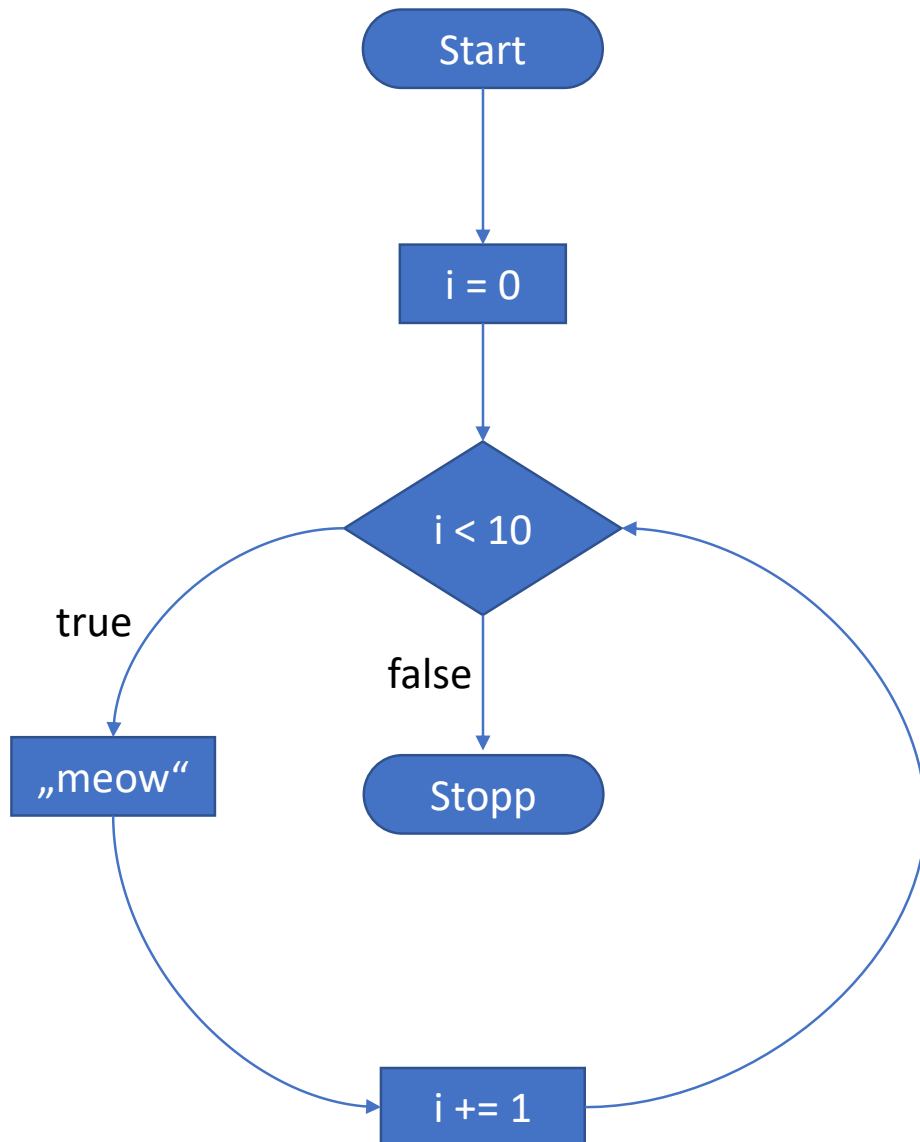
```
i = 3
while i >= 0:
    print(i)
    i += 1
```

Wir müssen eine Variable definieren, die innerhalb der while-Schleife geändert wird. Diese Variable dient also als Iterator. Außerdem ist es notwendig die Variable zu dekrementieren oder zu inkrementieren. Wir müssen unbedingt darauf achten, dass die Variable verändert wird, denn ansonsten läuft die while-Schleife unendlich lange. Wenn wir die Ausgabe von 0 bis 3 haben möchten, dann müssen wir den Code folgendermaßen ändern:

```
i = 0
while i < 3:
    print(i)
    i += 1
```

Wenn wir nun aber nicht die jeweilige Zahl ausgeben möchten, sondern beispielsweise 10x "meow", dann können wir das folgende Flussdiagramm betrachten. In dem Beispiel ist der Iterator **i**. Vor dem Start der while-Schleife wird der Variable **i** der Wert 0 zugewiesen. Da wir 10x "meow" ausgeben möchten, lautet unsere Abbruchbedingung $i < 10$. Diese Bedingung steht im Kopf der while-Schleife. Falls die Bedingung noch nicht erreicht und demnach true ist, geben wir "meow" aus. Außerdem müssen wir hier, anders als bei der for-Schleife, explizit **i** um 1 erhöhen:

$i += 1$. Falls die Abbruchbedingung erreicht worden ist, stoppt unsere while-Schleife und der Code, der innerhalb der while-Schleife eingerückt worden ist, wird nicht mehr ausgeführt. Ohne diese Abbruchbedingung würde die Schleife unendlich lang laufen und sich das Programm aufhängen.



Der dazugehörige Code sieht wie folgt aus:

```
i = 0
while i < 10:
    print("meow")
    i += 1
```

While-Schleifen führen einen Codeblock aus, solange eine Kondition wahr ist. Wenn wir nicht wie bei den for-Schleifen die Zahlen von 1 bis einschließlich 10 addieren möchten, sondern Zahlen so lange addieren, bis die Summe größer gleich einem bestimmten Wert ist, dann fällt unsere Wahl idealerweise auf eine while-Schleife. Dafür benötigen wir eine Variable, welche die Summe der einzelnen Zahlen speichert (**result**) sowie einen Iterator für unsere Zahlen (**number**).

```
result = 0
number = 1

while result <= 10:
    result += number
    number += 1
```

result: 15 und number: 6

Im nächsten Beispiel möchten wir so lange Süßigkeiten kaufen, bis wir kein Geld mehr haben. Dafür durchlaufen wir die while-Schleife so lange, bis wir kein **money** mehr haben. Für jede Einheit von **money** erhalten wir 2 **sweets**.

```
money = 5
sweets = 0

while money > 0:
    sweets += 2
    money -= 1
```

money: 0 und sweets: 10

Wenn Sie den Durchschnitt der Zahlen zwischen 5 und einschließlich 10 ausgeben möchten, dann benötigen Sie eine Variable für die Startzahl (**number**), für die Anzahl der Zahlen **count**, für die Summe (**result**) und eine while-Schleife, die so lange läuft, bis die Zahl dem Wert 10 entspricht. Innerhalb dieser Schleife fügen Sie **result** die aktuelle **number** hinzu und erhöhen danach **number** und **count** um jeweils 1. Nach der Schleife können Sie dann den Durchschnitt (**average**) durch eine Teilung von **result** und **count** berechnen.

```
number = 5
count = 0
result = 0

while number <= 10:
    result += number
    number += 1
    count += 1

average = result / count
```

Average beträgt:

Bearbeiten Sie bitte **Übung 8.1.**

9.2 Einführung: Schreiben

Beim Schreiben von while-Schleifen sollten wir darauf achten, dass wir korrekten Code schreiben, aber auch darauf, dass die Logik der while-Schleife korrekt ist.

Regel	Fehlerhafter Code	Korreakter Code
Das Schlüsselwort while wird klein geschrieben	<pre>count = 0 While count < 0: print(count) count += 1</pre>	<pre>count = 0 while count < 0: print(count) count += 1</pre>
Der Schleifenkopf hat einen Doppelpunkt am Ende der Zeile	<pre>count = 0 while count < 0 print(count)</pre>	<pre>count = 0 while count < 0 : print(count) count += 1</pre>
Einrücken des Codes, der ausgeführt werden soll, wenn eine Schleife ausgeführt werden soll	<pre>count = 0 while count < 0: print(count) count += 1</pre>	<pre>count = 0 while count < 0: print(count) count += 1</pre>

9 while-Schleifen

Beim Schreiben von bedingten Anweisungen müssen wir vorsichtig sein, denn kleine Änderungen können die Funktionsweise unserer bedingten Anweisungen beeinflussen. Hier sind einige Dinge zu beachten:

Regel	Fehlerhafter Code	Erklärung
Eine Variable muss definiert werden, damit sie genutzt werden kann.	<pre>while count < 10: print(count) count += 1</pre>	Der Code wird nicht ausgeführt, weil dem Compiler die Variable <code>count</code> nicht bekannt ist.
Die Zählervariable muss hochgezählt werden.	<pre>count = 0 while count < 10: print(count)</pre>	Ohne das Hochzählen läuft die while-Schleife unendlich lange. Dies führt zu einem Absturz des Programms.
Die Bedingung der while-Schleife muss sinnvoll sein.	<pre>count = 5 while count < 5: print(count) count += 1</pre>	Durch die inkorrekte while-Schleifen-Bedingung würde diese Schleife unendlich lang laufen. Anstatt die Zählervariable zu verringern, wird diese unendlich lange hochgezählt.

Bearbeiten Sie bitte **Übung 8.2.**

9.3 Template: Quersummenberechnung

Im folgenden Kapitel lernen Sie ein Template zur Berechnung der Quersumme einer Zahl kennen. Die Quersumme stellt dabei die Summe der Ziffernwerte einer natürlichen Zahl dar. Dafür addieren wir alle Ziffern einer Zahl.

In der Praxis werden Quersummen z.B. für das Prüfen der Datenintegrität genutzt. Beim (veralteten) ISBN-10 (Internationale Standardbuchnummer) wurde mithilfe der Quersumme % 11 überprüft, ob die Nummer valide ist. Weitere Beispiele wären das Speichern eines Passwortes, das aus der Quersumme der einzelnen Buchstabenreihenfolgen (sowie eines Schlüssels) besteht, anstelle dem Speichern von Klartextpasswörtern.

9.4 Template-Lesen: Quersumme

Nachdem wir in Kapitel 4.7 betrachtet haben, wie wir die einzelnen Ziffern einer Zahl auslesen können, bauen wir nun auf diesem Template auf. Sie sollten aus den beiden Schleifenkapiteln mitgenommen haben, dass man mit Schleifen eine quasi unendlich lange Anzahl an Operationen ausführen kann. In Kapitel 4.7 haben wir mithilfe von 3 Variablen die Ziffern einer dreistelligen Zahl gespeichert. Anstelle der Speicherung der einzelnen Ziffern, lernen Sie in diesem Kapitel, wie Sie die Quersumme einer "unendlich" langen Zahl bestimmen. Dafür benötigen wir eine Zahl, deren Quersumme wir berechnen möchten: **number**, eine Variable, welche die Quersumme speichert: **checksum** und eine while-Schleife. Die Abbruchbedingung der while-Schleife lautet **number > 0**. Innerhalb der while-Schleife fügen wir unserer **checksum** die letzte Ziffer der **number** hinzu. Außerdem führen wir eine Integerdivision der **number** durch 10 durch.

```
number = 123
checksum = 0

while number > 0:
    checksum += number % 10
    number = number // 10
```

Checksum beträgt:

6

Bearbeiten Sie bitte **Übung 8.3**.

9.5 Template-Schreiben: Quersumme

Beim Schreiben von Code zur Bestimmung der Quersumme einer Zahl sind einige Dinge zu beachten:

Regel	Fehlerhafter Code	Erklärung
Integerdivision zur Anpassung der ursprünglichen Zahl	<pre>number = 123 checksum = 0 while number > 0: checksum += number * 10 number = number / 10</pre>	Nutzen der normalen Division anstatt der Integer-Division. Dadurch kommt es zu einer unendlich langen Schleife, weil number niemals 0 entspricht.
Die Integerdivison der ursprünglichen Zahl erfolgt als letzter Schritt.	<pre>number = 123 checksum = 0 while number > 0: number = number // 10 checksum += number % 10</pre>	Dadurch wird die letzte Ziffer der ursprünglichen Zahl übersprungen.
Die umgedrehte Zahl wird nicht mit 10 multipliziert, wodurch die ursprünglichen Ziffern nur addiert werden.	<pre>number = 123 checksum = 0 while number > 0: checksum = number % 10 number = number // 10</pre>	Die umgedrehte Zahl wird nicht mit 10 multipliziert, wodurch die ursprünglichen Ziffern nur addiert werden.

Bearbeiten Sie bitte **Übung 8.4.**

10 Listen

10.1 Einführung: Lesen

Nachdem wir die beiden Schleifenarten in Python kennengelernt haben, werden wir in diesem Kapitel mit Listen arbeiten. Eine Liste ist eine veränderbare Datenstruktur in Python, die aus einer Reihe von Elementen besteht. Demnach kann diese, anders als eine Variable, mehrere Werte speichern. Listen werden eingesetzt, wenn wir mit einer Vielzahl von Werten arbeiten, die alle zusammen gehören.

Listen ermöglichen das Speichern mehrerer Werte. Außerdem können Listen auch unterschiedliche Datentypen enthalten, wie bspw. Integer, Floats und Strings.

```
result_list = ["Hello", 1, "World", 2, 2.1]
```

Wenn wir nun alle Werte zwischen 0 und einschließlich 10 in einer Liste speichern möchte, dann würden wir folgenden Code schreiben:

```
result_list = []  
  
for number in range(11):  
    result_list.append(number)
```

result_list besteht aus:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Wir benötigen also eckige Klammern, um eine Liste zu erstellen. Außerdem wird durch den Punkt die Methode `append` der Listenklasse aufgerufen¹. Außerdem bekommt diese Methode als Parameter unseren Iterator `number` übergeben.

Außerdem könnten wir auch zwei Listen erstellen und in einer Liste alle geraden natürlichen Zahlen hinzufügen und in der anderen alle ungeraden natürlichen Zahlen. Dafür müssten wir nur eine zweite leere Liste erzeugen und ein `else` in unserer `for`-Schleife hinzufügen:

```
even_list = []
odd_list = []

for number in range(11):
    if number%2 == 0:
        even_list.append(number)
    else:
        odd_list.append(number)
```

even_list besteht aus:	<table border="1"><tr><td>0, 2, 4, 6, 8, 10</td></tr></table>	0, 2, 4, 6, 8, 10
0, 2, 4, 6, 8, 10		
odd_list besteht aus:	<table border="1"><tr><td>1, 3, 5, 7, 9</td></tr></table>	1, 3, 5, 7, 9
1, 3, 5, 7, 9		

¹Die Rolle von Funktionen, Methoden und Klassen werden Sie in AuP und Datenstrukturen genauer kennenlernen.

Neben dem Hinzufügen von Elementen zu einer Liste, können wir auch Elemente aus dieser Liste löschen. Eine Möglichkeit ist das Nutzen der **remove**-Methode der Listen-Klasse. Wenn wir eine Liste mit den Zahlen 1, 2, 3, 4, 5 haben und aus dieser den Wert 2 löschen möchten, dann benötigen wir dafür folgenden Code:

```
number_list = [1, 2, 3, 4, 5]
number_list.remove(2)
```

number_list besteht aus: 1, 3, 4, 5

Mit einer Bedingten Anweisung können wir auch überprüfen, ob ein Element in einer Liste vorhanden ist.

```
number = 3
number_list = [1, 2, 3, 4, 5]
```

```
if number in number_list:
    print(number)
```

Ausgabe: 3

Bearbeiten Sie bitte **Übung 9.1.**

10.2 Einführung: Schreiben

Beim Schreiben von Listen sollten wir darauf achten, dass wir den Code korrekt schreiben, aber auch darauf, dass die Logik in den Listen korrekt ist.

Um eine Liste korrekt zu schreiben, sind einige Dinge zu beachten:

Regel	Kaputter Code	Korreakter Code
Eine Liste wird mit <code>[]</code> deklariert	<code>my_list = ()</code>	<code>my_list = []</code>
Um auf die einzelnen Listenelemente zuzugreifen, müssen <code>[]</code> genutzt werden	<code>my_list = [1, 2, 3]</code> <code>print(my_list(2))</code>	<code>my_list = [1, 2, 3]</code> <code>print(my_list[2])</code>

In der unteren Tabelle finden Sie typische logische Fehler, die auftreten, wenn ein Index außerhalb der Liste aufgerufen wird.

Regel	Fehlerhafter Code	Erklärung
Zugriff auf einen Index der Liste, der größer ist als die Liste selber	<code>my_list = [1, 2, 3]</code> <code>print(my_list[3])</code>	Dieser Zugriff ist nicht möglich, da die Liste nur 3 Elemente lang ist. Die Zählung beginnt dabei mit 0. Dies führt zu einem klassischen off-by-one-Fehler.
Zugriff auf einen Index der Liste, der größer ist als die Liste selber	<code>my_list = [1, 2, 3]</code> <code>for i in range(len(my_list)+1):</code> <code>print(my_list)</code>	” ”

Bearbeiten Sie bitte **Übung 9.2**.

10.3 Template-Lesen: Maximales adjazentes Produkt berechnen

Nach der Betrachtung der syntaktischen und semantischen Regeln von Listen in Python beschäftigen wir uns nun mit einer etwas komplexeren Aufgaben. Wir möchten die Produkte aller benachbarten/adjazenten Elemente einer Liste miteinander vergleichen.

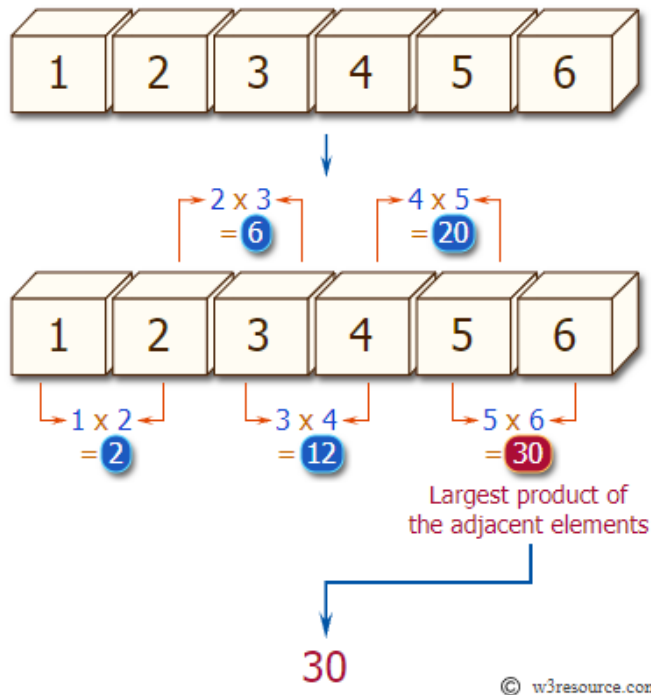


Abbildung 10.1: Abbildung Bilden von Produkten benachbarter Elemente.

Dafür müssen wir jeweils auf zwei Elemente zugreifen. Wir können eine for-Schleife nutzen und greifen nicht nur auf den Index des aktuellen Iterators hinzu, sondern auch auf das davor liegende oder das danach liegende Element. Dabei müssen wir beachten, nicht auf einen Index außerhalb der Liste zuzugreifen.

1. Weisen Sie dem maximalen Produkt den Wert aus dem Produkt zwischen dem ersten Element und dem zweiten Element der Liste zu.
2. Durchlaufen Sie mit einer for-Schleife alle Elemente der Liste, bis zum vorletzten Element

- a) Weisen Sie einer temporären Variable das Produkt aus dem aktuellen Element und dem nachfolgenden zu
- b) Überprüfen Sie, ob das aktuelle Produkt größer ist als das maximale Produkt, falls ja:
 - Setzen Sie das maximale Produkt auf das temporäre Produkt

Üben wir das Lesen des Templates zur Bestimmung des maximalen adjazenten Produkts!

Bearbeiten Sie bitte **Übung 9.3.**

10.4 Template-Schreiben: Maximales adjazentes Produkt berechnen

Im vorherigen Kapitel und in der dazugehörigen Übung haben Sie gesehen, dass es verschiedene Fehler gibt, die beim Schreiben des Codes zur Bestimmung des maximal möglichen adjazenten Produkts auftreten können.

Regel: Wir vergleichen nur die adjazenten Elemente in unserer Liste. Das erste und letzte Element sind nicht adjazent.

Fehlerhafter Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = number_list[0] * number_list[1]

for i in range(len(number_list)):
    temp = number_list[i] * number_list[i+1]
    if max_product < temp:
        max_product = temp
```

Erklärung: Im ersten Beispiel greifen wir auf den aktuellen Iterator als auch auf das davor liegende Element zu. Damit wir auf das davorliegende Element zugreifen können, beginnt die Iterator der Liste nicht bei 0, sondern bei 1. In den meisten gängigen Programmiersprachen führt diese zu einem Fehler, da Sie versuchen würden auf den Index -1 zuzugreifen. In Python greifen Sie aber auf den letzten Index zu, also den ersten Index vom Ende der Liste aus gesehen. Sie erhalten also keinen Laufzeitfehler, aber dieses Vorgehen würde zu einem semantischen Fehler führen. Da der Anfang und das Ende der Liste nicht miteinander verbunden sind.

Korreakter Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = number_list[0] * number_list[1]

for i in range(1, len(number_list)):
    temp = number_list[i-1] * number_list[i]

    if temp > max_product:
        max_product = temp
```

Regel: Die Variable für das maximale Produkt muss korrekt initialisiert werden.

Fehlerhafter Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = number_list[0] * number_list[1]

for i in range(len(number_list)):
    temp = number_list[i] * number_list[i+1]
    if max_product < temp:
        max_product = temp
```

Erklärung: Das zweite Beispiel stellt den Zugriff auf den aktuellen Iterator als auch auf das nachfolgende Element zu. Hierbei muss beachtet werden, nicht auf einen Index außerhalb der Liste zuzugreifen. Wenn die Abbruchbedingung der for-Schleife die Länge der Liste ist, dann würden wir durch Iterator + 1 auf einen Index zugreifen, der außerhalb der Liste liegt. Unser Compiler gibt uns dann einen "IndexError: list index out of range" und das Programm kann nicht ausgeführt werden.

Korreakter Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = number_list[0] * number_list[1]

for i in range(len(number_list)):
    temp = number_list[i] * number_list[i+1]
    if max_product < temp:
        max_product = temp
```

Regel: Die Variable für das maximale Produkt muss korrekt initialisiert werden.

Fehlerhafter Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = 0

for i in range(len(number_list)-1):
    temp = number_list[i] * number_list[i+1]
    if max_product < temp:
        max_product = temp
```

Erklärung: Nach der Betrachtung der Problematiken, die bei einem Zugriff auf Indizes außerhalb der Liste führen können, betrachten wir nun ein Problem mit der Berechnung des maximalen Produkts. Dafür berechnen wir erstmal das Produkt

des ersten und zweiten Elements der Liste. Wenn wir das Produkt bspw. auf 0 setzen würden, aber jedes zweite Element negativ wäre, dann würden wir am Ende ein falsches Produkt erhalten. Zudem durchlaufen wir die Liste bis zum vorletzten Element, damit wir innerhalb der Schleife auf das jeweilige nachfolgende Element zugreifen zu können, ohne auf einen Index außerhalb der Liste zuzugreifen. Außerdem müssen wir vergleichen, ob das temporäre Produkt größer als das aktuell größte Produkt ist. Falls ja, dann aktualisieren wir das größte Produkt und setzen das größte Produkt auf das temporäre Produkt.

Korrekt Code:

```
number_list = [1, 2, 3, 4, 5, 6]
max_product = number_list[0] * number_list[1]

for i in range(len(number_list)-1):
    temp = number_list[i] * number_list[i+1]
    if max_product < temp:
        max_product = temp
```

Bearbeiten Sie bitte **Übung 9.4.**

Vielen Dank für Ihre Teilnahme. Wir wünschen Ihnen einen erfolgreichen Start ins Wintersemester! Wir sehen uns spätestens im Sommersemester wieder.