

Programmieren mit Python

Eine Einführung

Dr. Aaron Kunert

aaron.kunert@salemkolleg.de

23. November 2022

Zu Beginn ...

Kurze Vorstellungsrunde

Schaffst Du es *in 60 Sekunden* folgende Fragen möglichst knackig und aussagekräftig zu beantworten?

- Wer bist Du?
- Windows, Mac oder Linux?
- Welche Vorkenntnisse hast Du beim Programmieren?
- Warum hast Du Dich zum Python-Kurs angemeldet?
- Wann wäre der Kurs für Dich perfekt gelaufen? (Best Case Szenario)
- Wann würdest Du den Kurs nicht weiter besuchen? (Worst Case Szenario)

Organisation des Kurses

- Pausen: 30 Minuten für den Nachmittagssnack. Bei Bedarf mehr Pausen. Bitte danach pünktlich kommen!
- Skript und alle Unterlagen findet man über MS Teams.
- Gelegentlich gibt es ein Aufgabenblatt (Teams) → Ca. 10 Tage Bearbeitungszeit, Abgabe durch Einladung zum Replit-Projekt → individuelles Kurzfeedback
- Lernleistung durch: Anwesenheit, Mitarbeit und Bearbeitung der Aufgabenblätter
- Wissenschaftliche Arbeit ist möglich
- Kommunikation per E-Mail
- Fragen sind immer und über alle Kanäle willkommen!

Didaktik des Kurses

- Mischung aus Vortrag, Präsenzübungen und Live-Coding
- Lösungen der Präsenzübungen gibt's im Handout in Teams
- Achtung: Präsenzübungen können erstmal frustrierend sein!
- Im Idealfall: Mehr Praxis statt Erklärungen

Ziele des Kurses

- Einblick in die „Denkweise“ eines Computers
- Einige universelle Konzepte von Programmiersprachen kennenlernen
- Schulung des analytischen Denkens
- Verständnis von Python-Syntax
- Programmierung eines rudimentären Quizspiels

Wo findet man Hilfe/Infos?

- Google
- `stackoverflow.com`
- Youtube (z.B. Tutorials)
- `docs.python.org/3`
- Bücher (z.B. *Python Crashkurs* v. Eric Matthes)
- `mailto: aaron.kunert@salemkolleg.de`

Was ist Python?

Wie funktioniert überhaupt die Programmierung in Python?

1. Man **schreibt** eine Abfolge von Befehlen/Anweisungen in eine Text-Datei (nicht Word!)
2. Danach lässt man diese Datei vom Python-Interpreter **ausführen**.

Was wird benötigt?

Am Anfang

- Compiler/Interpreter
- Texteditor (z.B. Mac: Xcode, Windows: Edit)

Später

- Google
- Integrierte Entwicklungsumgebung (IDE)
- Versionskontrolle (VCS)
- Virtueller Maschinen
- Datenbanken
- Grafikbearbeitung

Editor und Compiler müssen nicht auf dem eigenem Computer installiert sein. Es gibt dafür auch cloudbasierte Lösungen.

Warum Python?

- Einfaches Setup
- Einstiegsfreundliche Syntax
- Python ist eine Hochsprache
- Python muss nicht kompiliert, sondern nur interpretiert werden
- Große Community → großes *Ecosystem*
- Python ist extrem vielseitig
- Python ist plattformunabhängig

Typische Einsatzbereiche

- Automatisierung
- Webscraping
- Datenanalyse
- Webentwicklung

Wie Programmierer denken

Wie lernt man analytisches Denken?

Everyone in this country should learn to program a computer, because it teaches you to think.

(Steve Jobs)

Phasen des Lernens einer Programmiersprache

1. Annäherung: Fokus auf dem Begreifen der Grundkonzepte
2. Syntax: Fokus auf der korrekten Anwendung der Syntax
3. Funktionalität: Fokus liegt darauf, Problemstellungen *pragmatisch* zu lösen
4. Design: Fokus auf les-und wartbaren Code
5. Architektur: Fokus auf Strategie, Projekte nachhaltig und erweiterbar umzusetzen

Problem Solving

Sobald man die Syntax korrekt verwenden kann, steht das Lösen von Problemen beim Programmieren im Fokus.

Dabei ist die Kunst nur wenige, klare begrenzte Bausteine (die Befehle der Sprache) *kreativ* so zusammenzusetzen, damit das gegebene Problem gelöst wird.

Problemlösungsstrategien

- **Trial** and Error
- Formuliere laut und möglichst präzise, was eigentlich die Problemstellung ist
- Zerlege das Problem in kleinere Probleme oder mach Dir Zwischenziele
- Gibt es schon eine ähnliches Problem, was Du gelöst hast und von wo aus Du starten kannst?
- Erkläre anderen das Problem und was Du schon bisher geschafft hast
- Ändere Dein Denken: Scheitern ist nicht das Ende des Weges, sondern der Anfang
- To be continued

Die Konsole

Das Sprachrohr zum Computer

Definition: Konsole

Die Konsole ist ein simples Programm, das nur aus einem Eingabefeld besteht, und mit dem man mit einem anderen (in der Regel komplexeren Programm) mittels spezifischer Befehle kommunizieren kann.

Beispiele

- Windows-Eingabeaufforderung (Kommunikation mit Windows)
- Mac-Terminal (Kommunikation mit MacOS)
- Browser-Konsole (Kommunikation mit der Webseite mittels JavaScript)
- Die Python-Konsole

Für Programmierer*innen ist die Konsole der wichtigste Kommunikationsweg zu ihrem Computerprogramm.

Überprüfe, ob Python bei Dir installiert ist

1. Google wie man die Konsole bzw. das Terminal zum Betriebssystem öffnet
2. Öffne die Konsole
3. Prüfe, ob Python installiert ist, indem Du einen der folgenden Befehle ausprobierst
 - `python --version`
 - `python3 --version`
4. Interpretiere die Antwort

Programmieren in der Cloud

Schnell und unkompliziert einsteigen

Browserbasierte IDE verwenden

1. Gehe auf <https://replit.com>
2. Erstelle ein Konto (Sign up)
3. Klicke auf „Create repl“
4. Wähle als Template „Python“ aus

Erste Schritte im REPL

(Read-Evaluate-Print-Loop)

Probier mal folgende Kommandos aus

- `3 + 4`
- `2 - 7`
- `"Hello" + "World"`

Was machen die folgenden *Operatoren*?

- +
- -
- *
- /
- **

Und diese?

- %
- //
- ==
- <=
- <

Operatoren I

Die Operatoren `+` und `-` sind klar. Die Operatoren `*` und `/` bezeichnen Multiplikation und Division. Der Operator `**` berechnet die Potenz (hochnehmen).

Operatoren II

Der Operator `%` ist der Modulo-Operator (vgl. Wikipedia). Der Operator `//` arbeitet analog zur Division, rundet das Ergebnis jedoch auf die nächste ganze Zahl ab. Die Operatoren `==` (Gleichheit), `<=` (Kleiner gleich), `<` (kleiner als) sind Vergleichsoperatoren und geben entweder `True` oder `False` zurück

Wie rechnet Python?

- Wird Punkt-vor-Strich berücksichtigt?
- Kann man mit Klammern die Reihenfolge beeinflussen?
- Was ist der Unterschied zwischen `10/5` und `10//5` ?
- Was bedeutet das Kommando `_`?
- Wie kann man Zwischenergebnisse in Variablen speichern?

Rechenregeln

- Python rechnet Punkt-vor-Strich.
- Python berücksichtigt Klammern.
- Das Ergebnis von `/` ist stets eine Fließkommazahl, das Ergebnis von `//` ist stets eine ganze Zahl.
- Das Kommando `_`, referenziert das vorherige Ergebnis.
- Zwischenergebnisse lassen sich mittels des Zuweisungsoperators `=` in einer Variable speichern.

Variablen

Jeder Wert in Python kann in einer Variable gespeichert werden:

```
my_variable = 3
```

Die Zuweisung darf auch das Ergebnis einer Berechnung sein:

```
my_new_variable = 3 + 5
```

Die Zuweisung darf auch weitere Variablen enthalten:

```
my_brand_new_variable = my_variable + my_new_variable
```

Man darf auch Kettenzuweisungen machen:

```
a = b = c = 100
```

Gültige Variablennamen

- Erlaubt sind Buchstaben (nur ASCII), Ziffern und Unterstriche
- Der Name darf nicht mit einer Ziffer starten
- Beliebige Länge
- Wer's schon kennt als *regulärer Ausdruck*: `[_a-zA-Z][_0-9a-zA-Z]*`
- Schlüsselwörter sind nicht erlaubt

Liste der Schlüsselwörter

False	None	True	and	as
await	break	class	continue	def
else	except	finally	for	from
import	in	is	lambda	nonlocal
pass	raise	return	try	while
assert	global	with	elif	or
del	not	async	if	yield

Style-Guide Variablennamen

- Englische Wörter
- Nur Kleinbuchstaben
- Möglichst ausdrucksstarke Namen verwenden
- Keine Angst vor langen Namen
- Namen, die aus mehreren Worten bestehen, mit Unterstrich trennen (*snake-case*)

z.B. `students_in_this_room`, `number_of_unpaid_bills`

Probier's aus!

- Welchen Wert hat eine Variable, wenn man sie nicht vorher definiert hat?
- Was passiert, wenn man eine Variable definiert, die schonmal verwendet wurde?
- Wie kann man eine Variable mit Wert 3 um 1 vergrößern?

Lösung

- Verwendet man eine undefinierte Variable, wird ein Fehler geworfen
- Ja, man kann eine Variable einfach neu definieren
- Hat beispielsweise `my_variable` den Wert 3, so lässt sich der Wert wie folgt vergrößern:
`my_variable = my_variable + 1`

Datentypen

Jeder Wert in Python hat einen *Datentyp*. Unter anderem gibt es folgende *primitive* Typen in Python.

- `int` Integer (ganze Zahlen)
- `float` Float (Dezimalzahlen)
- `bool` Boolean (Wahrheitswerte)
- `str` String (Zeichenketten)
- `NoneType` (Typ des leeren Werts `None`)

Integer

Ganze Zahlen wie z.B. 1, -1, 0. Nicht aber 2.0 oder 0.0.

Float

Fließkommazahlen, z.B. 3.1415925. Achtung: Bei Float-Berechnungen können schnell „Überraschungen“ auftreten: Was ergibt z.B. 1.2 - 1.0 ?

Boolean

Booleans sind eine Sonderform von `int` und können nur die Werte `True` (entspricht 1) und `False` (entspricht 0) annehmen. Sie entstehen in der Regel, wenn man Fragen im Programm stellt (z.B. `3 < 4` oder `1 == 2`).

String

Strings sind beliebige Zeichenketten und müssen in (ein-, zwei- oder dreifache) Anführungszeichen eingeschlossen werden. Die Ausdrücke `'hello'`, `"Hello"` und `"""Hello"""` sind (fast) äquivalent.

Mehrzeilige Strings

Ein *Stringliteral* kann nur innerhalb einer Zeile definiert werden. Soll ein String mehrere Zeilen umfassen, müssen dreifache Anführungszeichen verwendet werden.

Steuerzeichen

Gewisse Kombinationen mit Backslash sind reservierte Steuerzeichen. So bezeichnet beispielsweise `\n` einen Zeilenumbruch und `\t` ein Tabulatorzeichen.

Beispiel: `"This text\nfills two lines"`

Escaping

Möchte man ein Steuerzeichen nicht ausführen, sondern buchstäblich nehmen. Muss man sie mit einem Backslash *escapen* bzw. maskieren.

Beispiel: `"This text fits in\\n one line"`

Raw-Strings

Möchte man alle Steuerzeichen eines Strings ignorieren, kann man ihn als *Raw-String* definieren.

Beispiel: `r"This \n String \t has no control characters"`

Typecasting (Umwandlung von Typen)

Implizit

Bei manchen Operationen nimmt Python automatisch eine Typumwandlung vor.

Beispiel: `1 + 2.0` ergibt `3.0`

Explizit

Die Funktionen `int()`, `float()`, `str()` und `bool()` führen jeweils eine Typumwandlung durch (sofern möglich). Beispiele:

- `int(2.0)` ergibt `2`
- `float(2)` ergibt `2.0`
- `int("3")` ergibt `3`

Typ einer Variablen ermitteln

Mit der Funktion `type()` lässt sich der Typ bestimmen, z.B. `type(3.2)`.

Versuche die Fragen erst ohne Python zu beantworten, überprüfe Deine Vermutung

- Welchen Datentyp hat das Ergebnis von `3 - 1.0` ?
- Was ist das Ergebnis von `"2" + 1` ?
- Was ist das Ergebnis von `"2" + "2"`?
- Sind die beiden Werte `0` und `"0"` gleich?
- Sind die beiden Werte `2` und `True` gleich?
- Sind die beiden Werte `bool(2)` und `True` gleich?
- Sind die beiden Werte `1` und `True` gleich?

Typaufgaben

- Der Datentyp des Ergebnisses ist `float`
- Fehlermeldung
- Das Ergebnis ist "22"
- Nein
- Nein
- Ja
- Ja

Erkläre mit Deinen eigenen Worten

- Nach welcher Regel wandelt `int()` eine Fließkommazahl in eine ganze Zahl um?
- Nach welchen Regeln wandelt `bool()` Zahlen und Strings in einen Wahrheitswert um?

Operatoren

Die wichtigsten Operatoren

- + (Addition oder Zusammenkleben von Strings)
- - (Subtraktion)
- * (Multiplikation)
- / (Division, ergibt immer ein Wert vom Typ `float`)
- ** (Potenzierung)
- % (*modulo-Operator*: Rest bei ganzzahliger Division)
- // (Division und Abrunden, ergibt immer ein Wert vom Typ `int`)
- == (Vergleichsoperator, ergibt immer ein Wert vom Typ `bool`)
- != (Ungleichheitsoperator, ergibt das Gegenteil von ==)

Operator-Präzedenz

1. Klammern
2. **
3. *, /, //, %
4. +, -

Operatoren gleichen Rangs werden innerhalb eines Ausdrucks von links nach rechts abgearbeitet.

Ausnahmen:

Potenzierung (**) und Zuweisung (=) werden von rechts nach links verarbeitet.

Kombinierte Zuweisung

Oft möchte man eine gegebene Variable neu zuweisen:

```
counter = 1  
counter = counter + 1      # counter = 2
```

Dies lässt sich auch kurz schreiben als

```
counter = 1  
counter += 1      # counter = 2
```

Analog sind die Operatoren `-=`, `*=`, `/=`, etc. definiert.

Was kommt raus?

x = 7

x += 3

x *= 6

x /= 15

Von der REPL zum Quellcode

Script Mode

Sobald man mehrere zusammenhängende Zeilen hat, wird die Eingabekonsole (REPL) sehr unübersichtlich. Daher gibt es auch die Möglichkeit, alle Programmzeilen zunächst aufzuschreiben und diese dann gebündelt von Python ausführen zu lassen. Im Gegensatz zum REPL bzw. interactive Mode von Python wird dies *Script Mode* genannt.

Beispiel

```
name = "Max"  
age = 20  
age = age + 1
```

Ausführung

Um diesen Code auszuführen, muss man bei Replit auf den Run-Button klicken oder alternativ den Shortcut Strg+Enter (Windows) bzw. Cmd+Enter (Mac) verwenden.

Achtung

Im Gegensatz zum REPL werden Ergebnisse von Rechnungen nicht mehr automatisch auf der Konsole ausgegeben.

Input/Output

Kommunikation über die Konsole

Die Konsole

Grafische Benutzeroberflächen sind zu Beginn relativ kompliziert, daher verwenden wir zunächst die *Python-Konsole* für die Kommunikation mit unserem Programm.

Output

Um einen String auf der *Konsole* auszugeben, verwende die Funktion `print()`.

Zum Beispiel: `print("Hello there")`.

Es können auch Variablen eingesetzt werden:

```
message = "Hello there"  
print(message) # Hello there
```

String Interpolation

Um Variablenwerte innerhalb eines Strings auszugeben, verwenden wir die String-Interpolation-Syntax:

```
my_value = 5
print(f"The variable my_value has the value {my_value}")
# The variable my_value has the value 5
```

Das geht auch als *inline expression*:

```
print(f"The sum of 1 and 2 is {1+2}")
# The sum of 1 and 2 is 3
```

Input

Um einen String vom User einzulesen, verwende die Funktion `input()`:

```
age = input("How old are you?")  
print(f"I am {age} years old")
```

Achtung

Das Ergebnis von `input` hat stets den Datentyp `string` auch wenn Zahlen eingelesen werden. Gegebenenfalls muss das Ergebnis mittels `int()` oder `float()` in den gewünschten Typ umgewandelt werden.

Beispiel: Input und Output kombiniert

```
name = input("What is your name?")  
age = input("What is your age?")  
print(f"Hello {name}, you are {age} years old")
```

Adressabfrage

Schreibe ein kurzes Skript, das Dich nach Deinem Namen, Alter und Adresse fragt. Wenn es alles eingelesen hat, soll es diese Infos in folgender Form auf der Konsole ausgeben:

```
Hallo Max, schön, dass Du da bist. Du bist 21 Jahre alt und wohnst in der  
Bismarckstraße 12 in Glücksstadt.
```

Adressabfrage

```
name = input("Dein Name: ")
age = input("Dein Alter: ")
street = input("Deine Adresse: ")
city = input("Deine Stadt: ")

print(f"Hallo {Name}, schön, dass Du da bist. Du bist {age} Jahre alt")
print(f"und wohnst in der {street} in {city}.")
```

Blick in die Zukunft

Schreibe ein kurzes Skript, dass Dich nach Deinem Alter fragt. Daraufhin soll es auf der Konsole ausgeben, wie alt Du in 15 Jahren sein wirst.

Lösung

```
age = input("Wie alt bist Du? ")
age = int(age) + 15
print(f"In 15 Jahren wirst Du {age} sein.")
```

Kommentare

Kommentare

Alle Zeichen einer Zeile, die hinter einem # (Hashtag) kommen, werden von Python ignoriert. So lassen sich Kommentare im Quellcode platzieren.

Beispiel

```
print("This line will be printed")  
# print("This line won't")
```

Conditionals

Ein Programm verzweigen

Problemstellung

Lies eine Zahl x ein. In Abhängigkeit von x soll Folgendes ausgegeben werden:

Die Zahl x ist größer als 0

bzw.

Die Zahl x ist kleiner 0

Wie macht man das?

Lösung (fast)

```
x = input("Gib eine Zahl x an")  
x = int(x)
```

```
if x > 0:  
    print("x ist größer 0")  
else:  
    print("x ist kleiner 0")
```

Struktur if-else Statement

if *Bedingung*:

 ▯▯ *Codezeile A1*

 ▯▯ *Codezeile A2*

 ▯▯ ⋮

else:

 ▯▯ *Codezeile B1*

 ▯▯ *Codezeile B2*

 ▯▯ ⋮

Codezeile C1

 ⋮

Wie funktioniert's?

Ist die `if`-Bedingung `True`, so wird der `if-Block` ausgeführt. Ist sie `False` wird der `else-Block` ausgeführt.

Definition: Block

Aufeinanderfolgende Codezeilen, die alle die gleiche Einrückung besitzen, nennt man *Block*. D.h. Leerzeichen am Zeilenanfang haben in Python eine syntaktische Bedeutung.

Good to know

- Der `else-Block` ist optional.
- Falls die Bedingung nicht vom Typ `bool` ist, so wird sie implizit umgewandelt.

Antwort überprüfen

Schreib ein Programm, dass folgende Frage auf der Konsole ausgibt und die Antwort einliest.

```
Was ist die Hauptstadt von Frankreich?
```

Darauf hin soll entsprechend der Antwort folgendes Feedback auf der Konsole erscheinen:

```
Das war richtig!
```

```
bzw.
```

```
Das war falsch! Die richtige Antwort ist Paris.
```

Antwort überprüfen

```
answer = input("Was ist die Hauptstadt von Frankreich?")
if answer == "Paris"
    print("Das ist richtig!")
else
    print("Das war falsch! Die richtige Antwort ist Paris.")
```

Volljährigkeit prüfen/Zutrittskontrolle

Schreibe ein Skript, dass nach dem Alter eines Users fragt und überprüft, ob der User schon volljährig ist. Dementsprechend soll auf der Konsole folgendes Feedback erscheinen:

```
Willkommen
```

```
bzw.
```

```
Du darfst hier nicht rein
```

Teilbarkeit bestimmen

Schreibe ein Skript, dass eine ganze Zahl einliest. Daraufhin soll auf der Konsole ausgegeben werden, ob die Zahl durch 7 teilbar ist. Beispiel: Ist die Eingabe 12, so ist die Ausgabe:

```
Die Zahl 12 ist nicht durch 7 teilbar.
```

Zutrittskontrolle

```
age = input("Wie alt bist Du? ")
age = int(age)

if age >= 18:
    print("Willkommen!")
else:
    print("Du darfst hier nicht rein!")
```

Teilbarkeit bestimmen

```
x = input("Gib eine Zahl ein: ")
x = int(x)

if x % 7 == 0:
    print(f"Die Zahl {x} ist durch 7 teilbar")
else:
    print(f"Die Zahl {x} ist nicht durch 7 teilbar")
```

Logische Operatoren

Booleans können mittels folgender Operatoren miteinander verknüpft werden:

`and` Ist genau dann `True`, wenn beide Operanden `True` sind.

`or` Ist genau dann `True`, wenn mindestens ein Operand `True` ist.

`not` Kehrt den nachfolgenden Wahrheitswert um.

Beispiel

- `2 > 0 and 3 > 4` ist `False`
- `1 > 0 or 6 > 1` ist `True`
- `not 2 < 1` ist `True`

Was ergeben die folgenden Ausdrücke?

- `not 2 < 3 and 4 < 7`
- `4 not == 8`
- `3 != 4 and not 4 == 8`
- `7 <= 7.0 and not 7 != 7.0`
- `7 > 5 or 4 < 5 and not 9 > 6`
- `not 3 < 6 > 8`
- `not 3`

Präzedenz beachten!

1. `==, !=, <=, <, >, >=`
2. `not`
3. `and`
4. `or`

Das elif-Statement

Mit der reinen if-else-Syntax können nur *binäre* Verzweigungen dargestellt werden. Um mehrer, gleichrangige Verzweigungsäste zu realisieren kann man das elif-Conditional verwenden.

Beispiel

```
if x < 0:
    print("x is < 0")
elif x == 0:
    print("x is 0")
elif x == 1:
    print("x is 1")
else:
    print("x is not negative but neither 0 nor 1")
```

Die Anzahl der elif-Blöcke ist beliebig. Der else-Block ist wie immer optional.

Worin unterscheiden sich die beiden Abschnitte?

Abschnitt 1:

```
if x % 2 == 0:
    # some Code here
if x % 3 == 0:
    # some Code here
else:
    # some Code here
```

Abschnitt 2:

```
if x % 2 == 0:
    # some Code here
elif x % 3 == 0:
    # some Code here
else:
    # some Code here
```

Berechne deinen Urlaubsort:

Anleitung:

- A) Wähle eine Zahl zwischen 1 und 9
- B) Multipliziere die Zahl mit 3
- C) Addiere 3 dazu
- D) Das Ergebnis mit 3 multiplizieren
- E) Zähle die beiden Stellen der Zahl zusammen
- F) Endergebnis = Dein Urlaubsort

Urlaubsort:

1. Italien

2. Spanien

3. Türkei

4. Bali

5. Holland

6. Sylt

7. Kroatien

8. Frankreich

9. Zuhause

10. USA



Lies eine Zahl zwischen 1 und 9 ein und gib auf der Konsole *deinen nächsten Urlaubsort* aus.

Urlaubsort

```
number = input("Gib eine Zahl zwischen 1 und 9 ein: ")
number = int(number)

number = number * 3
number = number + 3
number = number * 3

cross_sum = number // 10 + number % 10
print("Dort verbringst Du Deinen Urlaub: ")
if cross_sum == 1:
    print("Italien")
elif cross_sum == 2:
    print("Spanien")
# ... more elif statements ...
elif cross_sum == 9:
    print("Zu Hause")
else:
    print("USA")
```

Der Ternary Operator

Oftmals möchte man eine Variable in Abhängigkeit eines Wahrheitswertes definieren. Für diesen einfachen Fall, ist das `if-else`-Konstrukt sehr umständlich. Stattdessen kann man für die Kürze den *ternary operator* verwenden.

Beispiel

```
if x < 0:
    sign = "negative"
else:
    sign = "positive"
```

Stattdessen mit Ternary Operator

```
sign = "negative" if x < 0 else "positive"
```

Ternary Operator

Lies eine ganze Zahl ein und gib ihren Betrag auf der Konsole aus. Schaffst Du es, das Ganze mit weniger als 5 Zeilen Code zu programmieren?

Lösung

```
x = input("Gib eine Zahl ein: ")
x = float(x)
abs_value = x if x >= 0 else -x
print(f"Der Betrag von {x} ist {abs_value}")
```

Die For-Schleife

Einen Programmabschnitt x-mal ausführen

Problemstellung

Lies eine ganze Zahl x ein. Gib dann folgende Zeilen auf der Konsole aus

1
2
3
4
:
 x

Wie macht man das?

Lösung (fast)

```
x = input("Gib eine Zahl ein")  
x = int(x)
```

```
for k in range(1, x):  
    print(k)
```

Struktur der `for...in` Schleife

```
for Variable in range(start, end):
```

```
    Codezeile 1
```

```
    Codezeile 2
```

```
    :
```

Code, der nicht mehr Teil der Schleife ist

Wie funktioniert's?

Die Schleifenvariable wird zunächst gleich dem unteren Wert in `range` gesetzt. Dann wird der `for`-Block wiederholt ausgeführt. Bei jedem Durchgang wird die Schleifenvariable um 1 vergrößert und zwar so lange, wie der Wert der Schleifenvariable kleiner als der obere Wert in `range` ist.

Good to know

- Achtung: Die Schleifenvariable erreicht nie das obere Ende der `range`-Funktion, sondern bleibt immer 1 drunter.
- Die `range`-Funktion ist nicht auf 1er-Schrittweite beschränkt. Mit folgendem Ausdruck werden die Zahlen von 0 bis 9 z.B. in 3er-Schritten durchlaufen: `range(0, 10, 3)`.
- For-Schleifen sind flexibel und können alles mögliche durchlaufen, z.B. auch die einzelnen Buchstaben eines Strings (dazu später mehr).

Eingangsbeispiel

Schreibe ein Skript, das alle Zahlen von 1 bis 100 auf der Konsole ausgibt.

Lösung

```
for k in range(1, 101):  
    print(k)
```

Zählen

Zähle auf der Konsole in 7-er Schritten bis 70.

```
7  
14  
:  
70
```

Einmaleins: Die 7er-Reihe

Schreibe ein kleines Skript, was die 7er-Reihe (bis 70) wie folgt auf der Konsole ausgibt:

```
1 mal 7 ist 7  
2 mal 7 ist 14  
:  
:
```

Zählen

```
for k in range(1, 11):  
    print(7*k)
```

Einmaleins

```
for k in range(1, 11):  
    print(f"{k} mal 7 ist {7 * k}")
```

Zählen in krummen Abständen

Zähle auf der Konsole bis 20, allerdings sollen nur Zahlen ausgegeben werden, die durch 3 oder durch 5 teilbar sind:

```
3  
5  
6  
9  
10  
:  
20
```

Anzahl bestimmen

Bestimme die Anzahl der Zahlen zwischen 1 und 20, die durch 3 oder durch 5 teilbar sind.

Zählen in krummen Abständen

```
for k in range(1, 21):  
    if k % 3 == 0 or k % 5 == 0:  
        print(k)
```

Anzahl bestimmen

```
counter = 0  
for k in range(1, 21):  
    if k % 3 == 0 or k % 5 == 0:  
        counter = counter + 1  
print(f"Es gibt {counter} gesuchte Zahlen")
```

Das Gauss-Problem

Berechne die Summe der Zahlen 1 bis 100.

Lösung

```
result = 0
for k in range(1, 101):
    result = result + k
print(f"Das Ergebnis ist {result}.")
```

Schleife über einen String

Lies Deinen Namen auf der Konsole ein und gib die Buchstaben einzeln auf der Konsole auf.

Lösung

```
name = input("Gib Deinen Namen ein: ")
```

```
for letter in name:  
    print(letter)
```

Needle-Haystack-Problem

Lies Deinen Namen auf der Konsole ein und überprüfe, ob er den Buchstaben *a* (groß/klein) enthält.

Lösung

```
name = input("Gib ein Wort ein: ")

# Flag (Schalter) initialisieren
name_contains_letter_a = False

for letter in name:
    if letter == "a" or letter == "A":
        name_contains_letter_a = True

if name_contains_letter_a:
    print("Der Name enthält ein 'a'.")
else:
    print("Der Name enthält kein 'a'.")
```

Übung mit Trick

Quersumme

Lies eine ganze Zahl x ein und bestimme ihre Quersumme.

Tipp: Wandle die Zahl zunächst in einen String um

Lösung

```
number = input("Gib eine Zahl ein: ")
result = 0
# Wir lassen die Zahl als String, damit wir eine Schleife über die Ziffern legen können
for digit in number:
    result = result + int(digit) # Achtung: digit ist ja eigentlich ein String

print(f"Die Quersumme von {number} ist {result}")
```

Brutale Übung

Fibonacci-Zahlen

Die Zahlenfolge 1, 1, 2, 3, 5, 8, 13... nennt man *Fibonacci*-Folge. Dabei entsteht ein Element der Folge, durch die Addition des vorherigen und vorvorherigen Elements.

Berechne die 30. Fibonacci-Zahl.

Lösung

```
last = 1 # letzte Zahl
current = 1 # aktuelle Zahl

for k in range(2, 31):
    old_current = current # Zahl zwischenspeichern
    current = current + last
    last = old_current

print(f"Die {k}-te Fibonacci-Zahl ist {current}")
```

Die While-Schleife

Wie die For-Schleife nur abstrakter und open-end

Problemstellung

Lies immer wieder eine Zahl von der Konsole ein. Höre auf, wenn diese Zahl 7 ist.

Wie macht man das?

Lösung

```
x = 0
```

```
while x != 7:  
    x = input("Gib eine Zahl an: ")  
    x = int(x)
```

```
print("Fertig!")
```

Struktur der while-Schleife

```
while Bedingung:
```

```
    Codezeile 1
```

```
    Codezeile 2
```

```
    ⋮
```

```
Code, der nicht mehr Teil der Schleife ist
```

Wie funktioniert's?

Die Schleife wird solange ausgeführt, solange die *Bedingung* **True** ergibt. Nach jedem Durchgang wird der Ausdruck der *Bedingung* neu ausgewertet. Ist die Bedingung **False** wird der Code unterhalb des Schleifenblocks ausgeführt.

Achtung Endlosschleife

Man sollte immer darauf achten, dass die Bedingung in der `while`-Schleife auch wirklich irgendwann `False` wird. Ansonsten bleibt das Programm in einer *Endlosschleife* gefangen.

Ersetze eine `for`-Schleife durch eine `while`-Schleife

Schreibe ein Skript, das alle Zahlen von 1 bis 100 auf der Konsole ausgibt. Verwende eine `While`-Schleife.

Lösung

```
k = 1
while k <= 100:
    print(k)
    k += 1
```

Quizfrage

Schreibe ein Programm, dass solange nach einer Hauptstadt Deiner Wahl fragt, bis die richtige Antwort eingegeben wird.

Beispiel

Was ist die Hauptstadt von Frankreich?

Darauf hin soll entsprechend der Antwort folgendes Feedback auf der Konsole erscheinen:

Das war richtig!

bzw.

Das war falsch! Versuch's gleich nochmal

Quizfrage

```
answer = input("Was ist die Hauptstadt von Frankreich?")
while answer != "Paris":
    print("Das war leider falsch, versuch es gleich nochmal")
    answer = input("")
print("Das war richtig!")
```

Ratespiel

Definiere eine positive ganze Zahl `number_to_guess`. Der User kann nun wiederholt eine Zahl eingeben. Das Spiel endet, wenn die eingegebene Zahl mit `number_to_guess` übereinstimmt. Andernfalls wird auf der Konsole beispielsweise ausgegeben:

```
Sorry, Deine eingegebene Zahl war zu klein, versuche es nochmal:
```

Zusatz 1:

Am Ende soll die Anzahl der Versuche angegeben werden.

Zusatz 2:

Google, wie Python die Zahl `number_to_guess` zufällig erzeugen kann (das verbessert das Gameplay).

Ratespiel ohne Zusätze

```
number_to_guess = 512
guess = input("Rate meine Zahl: ")
guess = int(guess)

while guess != number_to_guess:
    if guess < number_to_guess:
        print("Deine Zahl war zu klein")
    else:
        print("Deine Zahl war zu groß")
    guess = input("Versuch's nochmal: ")
    guess = int(guess)

print("Du hast gewonnen")
```

Ratespiel mit Zusatz 1

```
number_to_guess = 512
guess = input("Rate meine Zahl: ")
guess = int(guess)
counter = 1

while guess != number_to_guess:
    if guess < number_to_guess:
        print("Deine Zahl war zu klein")
    else:
        print("Deine Zahl war zu groß")
    guess = input("Versuch's nochmal: ")
    guess = int(guess)
    counter = counter + 1

print(f"Du hast nach {counter} Versuchen gewonnen")
```

Ratespiel mit Zusatz 2

```
import random

number_to_guess = random.randint(1, 1000)
guess = input("Rate meine Zahl: ")
guess = int(guess)
# ... usw.
```

Den Schleifenfluss kontrollieren

`break, continue und else`

Das break-Statement

Taucht innerhalb einer Schleife das Schlüsselwort `break` auf, so wird die weitere Abarbeitung der Schleife abgebrochen. Die Ausführung wird mit dem Code *nach* dem Schleifenblock ausgeführt.

Beispiel

```
for k in range(1,100):  
    print(k)  
    if k > 3:  
        break  
print("fertig")  
# 1 2 3 4  
# fertig
```

Das continue-Statement

Taucht innerhalb einer Schleife das Schlüsselwort `continue` auf, so wird der aktuelle Schleifendurchgang abgebrochen. Die Ausführung wird mit der nächsten Schleifeniteration fortgesetzt.

Beispiel

```
for k in range(1,11):  
    if k % 2 == 0:  
        continue  
    print(k)  
# 1 3 5 7 9
```

Der else-Block einer Schleife

Analog zum `if`-Statement, kann auch eine Schleife einen `else`-Block haben. Dieser wird ausgeführt, wenn die Schleife *regulär* (also nicht durch die Verwendung von `break`) beendet wird.

Beispiel

```
name = input("Dein Name: ")

for letter in name:
    if letter == "a" or letter == "A":
        print("Dein Name enthält ein A")
        break
else:
    print("Dein Name enthält kein A")
```

Zählen bis zur nächsten 10er-Zahl

Lies eine Zahl x ein und gib auf der Konsole die Zahlen von x bis zur nächsten 10er-Zahl aus.
Ist die Eingabe $x = 17$, so soll die Ausgabe wie folgt aussehen:

```
17
18
19
20
```

Zählen mit Lücken

Schreibe ein Skript, dass die Zahlen von 1 bis 99 aufzählt, dabei allerdings die 10er-Zahlen weglässt. Verwende dabei ein `continue`-Statement.

Zählen bis zur nächsten 10er-Zahl

```
x = input("Gib eine Zahl an: ")
x = int(x)

for k in range(x, x + 11):
    print(k)
    if k % 10 == 0:
        break
```

Zählen mit Lücken

```
for k in range(1, 100):
    if k % 10 == 0:
        continue
    print(k)
```

Quizfrage mit Ausstiegsmöglichkeit

Schreibe ein Programm, dass solange nach einer Hauptstadt Deiner Wahl fragt, bis die richtige Antwort eingegeben wird. Wird allerdings der Buchstabe q eingegeben, so bricht das Programm ab.

Quizfrage mit Ausstiegsmöglichkeit

```
answer = input("Was ist die Hauptstadt von Frankreich?")
while answer != "Paris":
    print("Das war leider falsch, versuch es gleich nochmal")
    answer = input("")
    if answer == "q":
        break
else:
    print("Das war richtig!")
```

Harte Übung

Primzahltest

Lies eine ganze Zahl x ein und überprüfe, ob diese Zahl eine Primzahl ist. Die Ausgabe des Programms soll etwa wie folgt aussehen:

Die Zahl 28061983 ist eine Primzahl.

Lösung

```
x = input("Gib eine Zahl ein: ")
x = int(x)

for k in range(2, x):
    if x % k == 0:
        print(f"{x} ist keine Primzahl.")
        break
else:
    print(f"{x} ist eine Primzahl.")
```

Listen

Viele Variablen gleichzeitig speichern

Problemstellung

Lies mit Hilfe einer Schleife nach und nach Ländernamen ein. Alle Länder sollen dabei gespeichert werden. Danach sollst Du die Möglichkeit haben, das soundsovielte Land anzeigen lassen zu können.

Wie macht man das?

Lösung (fast)

```
# ...  
# Um das Eingaben der Länder kümmern wir uns noch  
countries = ["Deutschland", "Frankreich", "Italien", "Spanien"]  
  
index = input("Das wievielte Land möchtest Du nocheinmal anschauen?")  
index = int(index)  
  
print(f"Das { index }. Land ist { countries[index] }")
```

Struktur einer *Liste*

```
my_list = [element_0, element_1, ..., element_n]
```

Die Variable `my_list` trägt nicht nur einen Wert, sondern $n + 1$ Werte. Ansonsten verhält sich `my_list` wie eine ganz „normale“ Variable. Als Einträge einer Liste sind beliebige Werte mit beliebigen Datentypen zugelassen.

Frage: Welchen Datentyp hat die Liste `[2, 2.3, "Hello"]` ?

Auf Listenelemente zugreifen

Auf das n -te Element der Liste `my_list` kann man mittels `my_list[n]` zugreifen.

Mit `my_list[-1]`, `my_list[-2]`, etc. kann man auf das letzte, vorletzte, etc. Element der Liste zugreifen.

Achtung

Python fängt bei 0 an zu zählen. D.h. das erste Element in der Liste hat den Index 0.

Beispiel: `my_list[1]` liefert das **2. Element** der Liste.

Schreibzugriff auf Listenelemente

Nach dem gleichen Prinzip lassen sich einzelne Listeneinträge verändern.

Beispiel: `my_list[3] = "Albanien"`.

Achtung

Man kann nur schon existierende Listeneinträge verändern.

Listeneinträge hinzufügen

Mit der *Methode* `.append()` kann ein Eintrag zur Liste hinzugefügt werden.

Bsp: `my_list.append("Russland")` fügt den String "Russland" zu der Liste hinzu.

Listeneinträge entfernen

Mit dem Keyword `del` kann man Einträge an einer bestimmten Position löschen. Dabei verschieben sich die darauffolgenden Einträge um 1 nach vorne.

Beispiel: `del my_list[2]` löscht das dritte Element.

Mit der Methode `.remove()` kann man Einträge mit einem bestimmten Wert löschen.

Beispiel: `my_list.remove("Italien")` entfernt den ersten Eintrag mit dem Wert "Italien". Ist der Wert nicht vorhanden gibt es eine Fehlermeldung.

Eine Liste erstellen

Schreibe ein kleines Programm, dass Dich ca. 4x nach einem Land fragt, das Du besucht hast und Dir am Ende die Liste der besuchten Länder ausgibt.

Lösung

```
countries = []  
for k in range(1, 5):  
    country = input("Wo warst Du schonmal im Urlaub? ")  
    countries.append(country)  
print(countries)
```

Das Eingangsproblem

Schreibe ein kleines Programm, dass solange Namen von Ländern einliest, bis Du **q** drückst. Danach sollst Du die Möglichkeit haben, eine Zahl k einzugeben, so dass das k -te Land angezeigt wird.

Das Eingangsproblem

```
countries = []
while True:
    country = input("Gib ein Land ein: ")
    if country == "q":
        break
    countries.append(country)

index = input("Das wievielte Land möchtest Du nochmal anschauen?")
index = int(index)
print(f"Das { index }. Land ist { countries[index-1] }.")
```

Mutability

Listen sind der erste Datentyp, den wir kennenlernen, der *mutable* (veränderbar) ist. Die bisherigen Datentypen waren *immutable*, d.h. man konnte sie zwar überschreiben, aber nicht verändern.

Call by Reference vs. Call by Value

Enthält die Variable `my_list` eine Liste, so speichert Python eigentlich gar nicht die Liste in dieser Variable, sondern nur die Speicheradresse der Liste. Dieses vorgehen nennt man auch *Call by Reference*. Bei den Datentypen `int` und `str` wird stattdessen tatsächlich der Wert der Variable abgespeichert. Dies nennt man *Call by Value*.

Eine Liste kopieren

Definiere die Variable `my_list` als die Liste `[1,2,3]`. Kopiere die Variable `my_list` in die Variable `my_list_copy`. Füge einen weiteren Eintrag zu `my_list` hinzu. Welchen Wert hat `my_list_copy`?

Lösung

```
my_list = [1, 2, 3]
my_list_copy = my_list
my_list.append(4)
print(my_list_copy) # 1 2 3 4
```

Schleife über Liste

Analog wie über Strings und Ranges kann man Schleifen auch über eine Liste laufen lassen.

Beispiel

```
countries = ["Bulgarien", "Griechenland", "Türkei", "Libanon"]
```

```
for country in countries:  
    print(country)
```

```
# Bulgarien
```

```
# Griechenland
```

```
# Türkei
```

```
# Libanon
```

Schleife über Liste mit Indizes

Möchte man in einer Schleife nicht nur die Listeneinträge, sondern auch die Indizes verwenden, so muss man die Funktion `enumerate()` auf die Liste anwenden.

Beispiel

```
countries = ["Guatemala", "Nicaragua", "Honduras", "Belize"]
```

```
for (index, country) in enumerate(countries):  
    print(f"Das {index + 1}. Land ist {country}")
```

```
# Das 1. Land ist Guatemala
```

```
# Das 2. Land ist Nicaragua
```

```
# Das 3. Land ist Honduras
```

```
# Das 4. Land ist Belize
```

Liste durchsuchen

Prüfe, ob in einer Liste von Ländern das Land "Italien" vorkommt. Gib dazu auf der Konsole entweder

```
Italien ist in der Liste
```

oder

```
Italien ist nicht in der Liste
```

aus.

Liste durchsuchen

Wähle ein Beispiel für countries

```
countries = ["Finnland", "Norwegen", "Schweden", "Dänemark"]
```

```
for country in countries:
    if country == "Italien":
        print("Italien ist in der Liste")
        break
else:
    print("Italien ist nicht in der Liste")
```

Ist ein Element in einer Liste enthalten?

Möchte man prüfen, ob ein Element in einer Liste enthalten ist, so kann man auch das Schlüsselwort *in* verwenden.

Beispiel

```
countries = ["Finnland", "Norwegen", "Schweden", "Dänemark"]
```

```
var_1 = "Finnland" in countries
```

```
var_2 = "Deutschland" in countries
```

```
print(var_1)  # True
```

```
print(var_2)  # False
```

Eine Liste sortieren

Um eine Liste zu sortieren, verwende die Methode `.sort()`. Dies verändert die Liste dauerhaft.

Um eine sortierte Kopie einer Liste zu erstellen, verwende die Funktion `sorted()`.

Mit Hilfe des Parameters `reverse=True` lässt sich eine Liste absteigend ordnen.

Beispiel für `sort`

```
my_list = [1, 5, 2, 7]
my_list.sort()
print(my_list)  # [1, 2, 5, 7]
```

Beispiel für `sorted`

```
my_list = [1, 5, 2, 7]
sorted_list = sorted(my_list)
print(my_list)  # [1, 5, 2, 7]
print(sorted_list)  # [1, 2, 5, 7]
```

Beispiel für absteigende Sortierung

```
my_list = [1, 5, 2, 7]
my_list.sort(reverse=True)
print(my_list)  # [7, 5, 2, 1]
```

```
my_list = [7, 12, 5, 18]
sorted_list = sorted(my_list, reverse=True)
print(sorted_list)  # [18, 12, 7, 5]
```

Beste/Schlechteste Note

Sei `grades` eine Liste der Noten deiner letzten Klausuren (z.B. `grades = [12, 9, 14, 11]`). Gib dann auf der Konsole einmal die beste und einmal die schlechteste Note aus.

Lösung

```
grades = [12, 9, 14, 11]
grades.sort()
min_grade = grades[0]
max_grade = grades[-1]
print(f"Schlechteste Note: {min_grade}")
print(f"Beste Note: {max_grade}")
```

Nützliche Funktionen/Methoden

Für Listen stellt Python viele nützliche Methoden bzw. Funktionen bereit. Wenn Du googlest, findest Du für viele „Alltagsfragen“ eine Lösung.

Zum Beispiel hier: <https://docs.python.org/3/tutorial/datastructures.html>

Beispiele

```
my_list = [2, 4, 8, 1]
```

```
len(my_list)    # = 4   (Gibt die Anzahl der Elemente an)
```

```
sum(my_list)    # = 15  (Berechnet die Summe der Elemente)
```

```
my_list.reverse() # [1, 8, 4, 2] (Dreht die Reihenfolge um)
```

```
my_list.insert(2,-1) # [2, 4, -1, 8, 1] (fügt den Wert -1 an Position 2 ein)
```

```
my_list.pop()    # 1 (Gibt den letzten Eintrag der Liste zurück und entfernt ihn aus der Liste)
```

Durchschnittsnote

Sei `grades` wieder eine Liste mit deinen letzten Noten. Gib auf der Konsole die Durchschnittsnote aus.

Lösung

```
grades = [12, 9, 14, 11]
total_sum = sum(grades)
count = len(grades)
average = total_sum/count
print(f"Die Durchschnittsnote ist {average}")
```

Slicing

Wenn man eine Liste hat, ist es oft nötig, einen Teil der Liste „auszuschneiden“.

Dafür hat Python die *Slice-Notation* eingeführt.

Diese funktioniert nach folgendem Schema:

```
my_list[start:stop:step].
```

Die Einträge (start, stop, step) sind dabei jeweils optional. Wie immer wird der obere Wert (stop) gerade nicht erreicht.

Slicing lässt sich übrigens auch nach dem gleichen Schema auch auf Strings anwenden.

Wichtig

Wenn man Slicing anwendet, erhält man eine Kopie der ausgewählten Elemente zurück. Die ursprüngliche Liste wird *nicht* verändert.

Beispiele

```
my_list = [2, 4, 6, 8, 10]
```

```
my_list[1:3]      # [4, 6]
my_list[0:4]      # [2, 4, 6, 8]
my_list[1:1]      # []
my_list[0:4:2]    # [2, 6]
my_list[:3]       # [2, 4, 6]
my_list[2:]       # [6, 8, 10]
my_list[:]        # [2, 4, 6, 8, 10]
my_list[1:-2]     # [4, 6]
my_list[-3:-1]    # [6, 8]
my_list[::-1]     # [10, 8, 6, 4, 2]
```

Dictionaries

Problemstellung

Eine Variable soll nicht nur die Namen von Ländern enthalten, sondern auch noch deren Hauptstadt.

Wie macht man das?

Lösung

```
capitals = {"Deutschland": "Berlin", "Spanien": "Madrid", "Italien": "Rom"}
```

```
country = input("Von welchem Land möchtest Du die Hauptstadt wissen?")
```

```
print(f"Die Hauptstadt von { country } ist { capitals[country] } Punkte")
```

Struktur eines *Dictionary*s

```
my_dict = {key_1:value_1, key_2:value_2, ..., key_n:value_n}
```

Das *Dictionary* `my_dict` enthält Schlüssel-Wert-Paare (*key-value-pairs*). Die Schlüssel müssen eindeutig und unveränderlich sein (z.B. vom Typ `string` oder `int`). Die Werte dürfen beliebige Datentypen sein.

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
capitals = {  
    "Deutschland": "Berlin",  
    "Spanien": "Madrid",  
    "Italien": "Rom"  
}
```

- Dictionaries sind mutable, können also verändert werden.
- Dictionaries besitzen keine vernünftige Anordnung und können nicht geordnet werden.
- Ein Dictionary kann leer sein.

- Oftmals bietet es sich an, statt einem Dictionary eine Liste von Dictionaries zu verwenden:

```
countries = [  
    {  
        "name": "Deutschland",  
        "capital": "Berlin",  
        "pop": 82000000,  
        "is_eu_member": True  
    },  
    # ...  
    {  
        "name": "Italien",  
        "capital": "Rom",  
        "pop": 65000000,  
        "is_eu_member": True  
    }  
]
```

Auf Dictionary-Elemente zugreifen

Sei `my_dict = {"a": 5, "b": 8}`.

Mit der Syntax `my_dict["a"]` kann man den Wert an der Stelle "a" auslesen.

Mit der Syntax `my_dict["a"] = 12` kann man einzelne Werte des Dictionaries verändern.

Auf diese Weise können auch ganz neue Paare hinzugefügt werden. Zum Beispiel:

`my_dict["c"] = -2`.

Dictionary manipulieren

Gegeben sei das folgende Dictionary:

```
grades = {"Mathe": 8, "Bio": 11, "Sport": 13}
```

Bestimme die Durchschnittsnote dieser drei Fächer. Verbessere danach Deine Mathenote um einen Punkt und füge noch eine weitere Note für Englisch hinzu (Abfrage über Konsole). Gib danach erneut den Durchschnitt an.

Dictionary manipulieren

```
grades = {"Mathe": 8, "Bio": 11, "Sport": 13}

grades_sum = grades["Mathe"] + grades["Bio"] + grades["Sport"]
average = grades_sum/len(grades)
print(f"Der Durchschnitt ist {average} Punkte")

grades["Mathe"] += 1

eng_grade = input("Welche Note hast Du in Englisch? ")
eng_grade = int(eng_grade)
grades["Englisch"] = eng_grade

grades_sum += grades["Englisch"]
average = grades_sum/len(grades)
print(f"Der Durchschnitt ist {average} Punkte")
```

Einen Eintrag aus einem Dictionary entfernen

Wie bei Listen, kann man mittels `del`-Statement einen Eintrag aus einem Dictionary entfernen:

```
del eu_countries["united_kingdom"]
```

Was wird hier passieren?

```
old_capitals = {"Deutschland": "Bonn", "Norwegen": "Oslo"}  
new_capitals = old_capitals  
  
new_capitals["Deutschland"] = "Berlin"  
  
print(old_capitals)  
print(new_capitals)
```

Erklärung

Da Dictionaries mutable sind, findet bei ihnen der Aufruf mittels *Call by Reference* statt. Das heißt, dass in der Variable `old_capitals` bzw. `new_capitals` nicht die Länder gespeichert sind, sondern nur die Speicheradresse, wo die Länder zu finden sind. Ändert man die zugrundeliegenden Daten an einer Stelle, so ändern sie sich daher auch an der anderen Stelle.

Eine Kopie von einem Dictionary erstellen

Mit der Funktion `dict()` kann man eine Kopie von einem Dictionary erstellen.

Beispiel: `dict(my_dict)` erstellt eine Kopie von `my_dict`.

Schleife über Dictionary I

Ähnlich wie bei Listen kann man Schleifen auch über ein Dictionary laufen lassen.

Beispiel

```
capitals = {"Litauen": "Vilnius", "Lettland": "Riga", "Estland": "Tallin"}
```

```
for item in capitals:  
    print(item)
```

```
# Litauen
```

```
# Lettland
```

```
# Estland
```

Schleife über Dictionary II

Möchte man in der Schleife nicht nur die Schlüssel, sondern auch die Werte des Dictionaries zur Verfügung haben, so muss man die Methode `.items()` auf das Dictionary anwenden.

Beispiel

```
capitals = {"Litauen": "Vilnius", "Lettland": "Riga", "Estland": "Tallin"}
```

```
for key, value in capitals.items():  
    print(f"Hauptstadt von {key}: {value}")
```

```
# Hauptstadt von Litauen: Vilnius
```

```
# Hauptstadt von Lettland: Riga
```

```
# Hauptstadt von Estland: Tallin
```

Zwei Dictionaries kombinieren

Gegeben seien zwei Dictionaries, z.B.

```
eu = {"Deutschland": "Berlin", "Frankreich": "Paris" }
```

und

```
non_eu = {"Ukraine": "Kiew", "Norwegen": "Oslo" }
```

Füge die Einträge des zweiten Dictionaries zum ersten Dictionary hinzu.

Ein Dictionary „filtern“

Sei ein beliebiges Dictionary mit Noten gegeben. Entferne alle Einträge, deren Note schlechter als 5 Punkte ist.

Zwei Dictionaries kombinieren

```
eu = {"Deutschland": "Berlin", "Frankreich": "Paris" }
non_eu = {"Ukraine": "Kiew", "Norwegen": "Oslo" }

for key,value in non_eu.items():
    eu[key] = value
print(eu)
```

Ein Dictionary „filtern“

```
grades = {"Deutsch": 11, "Mathe": 3, "Sport": 14, "Geschichte": 1}
# Man darf die Länge eines Dictionaries in einer Schleife nicht verändern, deshalb machen wir eine Kopie
result = dict(grades)
for key, value in grades.items():
    if value < 5:
        del result[key]
print(result)
```

Ein Dictionary zerlegen

Mit der Methode `.keys()` erhält man eine Liste aller Schlüssel eines Dictionaries.

Mit der Methode `.values()` erhält man eine Liste aller Werte eines Dictionaries.

In beiden Fällen, muss das Ergebnis mittels der Funktion `list()` in eine Liste umgewandelt werden.

Beispiel

```
my_dictionary = {"China": "Peking", "Japan": "Tokio", "Korea": "Seoul"}
```

```
countries = my_dictionary.keys()
countries = list(countries)
```

```
capitals = my_dictionary.values()
capitals = list(capitals)
```

```
print(countries)  # ["China", "Japan", "Korea"]
print(capitals)   # ["Peking", "Tokio", "Seoul"]
```

Comprehensions

Typische Manipulationen

Sehr häufig möchte man eine Datenstruktur (d.h. eine Liste oder ein Dictionary) basierend auf den Werten manipulieren. Dabei werden vor allem zwei Aspekte immer wieder gebraucht: Transformationen und Filter.

Transformation

Ersetzt man jedes Element einer Liste durch ein aus dem ursprünglich berechnetem Element, so spricht man von einer *Transformation* (engl. map/mapping).

Beispiel

Gegeben ist die Liste `my_list = [2, 5, 3, 12, 7]`. Die Liste soll so manipuliert werden, dass alle Einträge durch ihren doppelten Wert ersetzt werden.

Traditionelle Lösung

```
my_list = [2, 5, 3, 12, 7]
result = []
for k in my_list:
    result.append(2 * k)
print(result)
```

The Pythonian Way

```
my_list = [2, 5, 3, 12, 7]
result = [2 * k for k in my_list]
print(result)
```

Filter

Streich man Elemente entsprechend ihres Wertes aus einer Liste, so spricht man von einem *Filter*.

Beispiel

Gegeben ist die Liste `my_list = [2, 5, 3, 12, 7]`. Aus der Liste sollen alle ungeraden Einträge gestrichen werden.

Traditionelle Lösung

```
my_list = [2, 5, 3, 12, 7]
result = []
for k in my_list:
    if k % 2 == 0:
        result.append(k)
print(result)
```

The Pythonian Way

```
my_list = [2, 5, 3, 12, 7]
result = [k for k in my_list if k % 2 == 0]
print(result)
```

Kombination aus Transformation und Filter

Selbstverständlich können Transformationen und Filter auch kombiniert werden.

Beispiel

Lösche alle ungeraden Zahlen und verdopple dann alle Zahlen:

```
my_list = [2, 5, 3, 12, 7]
result = [2 * k for k in my_list if k % 2 == 0]
print(result)
```

Struktur List Comprehension

```
result = [ Wunscheintrag(k) for k in my_list if Bedingung(k) ]
```

Dictionary Comprehension

Man kann das gleiche Verfahren auch auf Dictionaries anwenden. Dabei können jeweils key und value für die Transformationen und Filter verwendet werden.

Beispiel

```
my_dict = {"a": 2, "b": 3}
result = {key: value for (key, value) in my_dict.items()}
print(result)
```

Struktur Dictionary Comprehension

```
result = { Wunsch-schlüssel(k,v) : Wunsch-Wert(k,v)  
          for (k,v) in my_dict.items() if Bedingung(k,v) }
```

List Comprehension

Gegeben sei eine beliebige Liste von ganzen Zahlen. Streiche alle Zahlen, die ungerade oder negativ sind. Ersetze die übrigen Zahlen durch ihre Hälfte.

Lösung

```
my_list = [2, -3, 40, 15, 7, 8]
result = [k//2 for k in my_list if k % 2 == 0 and k >= 0]
```

Dictionary Comprehension

Gegeben sei das Dictionary `{"Mathe": 9, "Sport": 13, "Physik": 4, "Bio": 12}`.
Lösche nun daraus alle Noten unter 5 Punkte sowie die Sportnote. Zusätzlich soll das Dictionary danach wie folgt aussehen: `{"In Mathe": "9 Punkte", "In Bio": "12 Punkte"}`.

Lösung

```
grades = {"Mathe": 9, "Sport": 13, "Physik": 4, "Bio": 12}
result = {
    "In " + key: f"{value} Punkte"
    for (key, value) in grades.items()
    if value >= 5 and key != "Sport"
}
```

Funktionen

Wie man Code wiederverwerten kann

Problemstellung

Es sei eine Liste mit Temperaturen gegeben:

```
temperatures = [22, 18, 20, 15, 12, 7, 5, -2, 4]
```

Es sollen zunächst Durchschnitte von jeweils der folgenden Grundgesamtheit genommen werden:

- Alle Werte
- Die ersten drei Werte
- Jeder zweite Wert

Statt durch eine Zahl, soll das Ergebnis jedoch mit den Worten

- "Mild" für Durchschnitte ≥ 15 Grad
- "Zu kalt" für Durchschnitte < 15 Grad

abgespeichert werden. Wie macht man das *elegant*?

Lösung (Hauptsache es funktioniert)

```
average = sum(temperatures) / len(temperatures)
if average >= 5:
    average = "Mild"
else:
    average = "Zu kalt"

average_2 = sum(temperatures[:3]) / len(temperatures[:3])
if average_2 >= 5:
    average_2 = "Mild"
else:
    average_3 = "Zu kalt"

average_3 = sum(temperatures[::2]) / len(temperatures[::2])
if average_3 < 5:
    average_3 = "Zu kalt"
else:
    average_3 = "Mild"

#
```

Nachteile dieser Lösung

- Viel Schreibarbeit, viel Wiederholung
- Der Code ist schwierig zu lesen. Man sieht vor lauter Wiederholungen nicht, was passiert.
- Jedes Mal, wenn man diese „Berechnungslogik“ verwendet, könnte man einen (Tipp-)Fehler machen.
- Wenn man das Anforderungsprofil minimal ändert, muss diese „Logik“ bei *jedem* Auftreten im Code geändert werden (z.B. statt "Mild" soll das Ergebnis "Warm" heißen). In echten Projekten, kann das schnell ein paar Hundert Male sein.

Bessere Lösung

```
def compute_average(temp_list):  
    result = sum(temp_list) / len(temp_list)  
    if result >= 5:  
        result = "Mild"  
    else:  
        result = "Zu kalt"  
    return result  
  
average = compute_average(temperatures)  
average_2 = compute_average(temperatures[:3])  
average_3 = compute_average(temperatures[:2])
```

Definition: Funktion

Eine Funktion ist ein Codeblock, der nur ausgeführt wird, wenn die Funktion *aufgerufen* wird. Man kann der Funktion Werte als *Parameter* übergeben. Sie kann auch einen Wert als Ergebnis *zurückgeben*.

Man kann sich eine Funktion wie eine Maschine vorstellen, wo man oben Dinge (=Parameter) hineinfüllt und unten ein Ergebnis (=Rückgabewert) herausbekommt. Unabhängig von dem Eingabe-Ausgabe-Prinzip, kann solch eine Maschine auch Nebeneffekte (z.B. Krach) produzieren.

Man unterscheidet zwischen *Definition* und *Ausführung* einer Funktion.

Struktur der Funktions-Definition

```
def Funktionsname(Parameter_0, Parameter_1, ..., Parameter_n):  
    Codezeile1  
    Codezeile2  
    :  
    return Ergebnis
```

Struktur eines Funktionsaufrufs

```
result = Funktionsname(Argument_0, Argument_1, ..., Argument_n)
```

Good to know

- Eine Funktion muss schon *vor* dem ersten Aufruf definiert worden sein (das ist nicht in allen Sprachen so).
- Die Eingabwerte nennt man in der Funktionsdefinition *Parameter*, beim Aufruf der Funktion nennt man sie jedoch *Argumente*.
- Nicht jede Funktion braucht Eingangsdaten. Die Liste von Parametern einer Funktion kann daher leer sein.
- Beim Aufruf spielt die Reihenfolge der angegebenen Argumente eine entscheidende Rolle. Sie werden entsprechend der Reihenfolge den Parametern in der Definition zugeordnet.
- Eine Funktion muss nicht unbedingt etwas zurückgeben, d.h. das `return`-Statement ist optional.
- Das `return`-Statement muss nicht unbedingt am Schluss der Funktion stehen. Jedoch wird Code, der nach dem `return`-Statement kommt, nicht mehr ausgeführt.

Funktion ohne Parameter

Schreibe eine Funktion, die Deinen Namen auf der Konsole ausgibt.

Funktion mit einem Parameter

Schreibe eine Funktion, die die übergebene Zahl verdoppelt und das Ergebnis zurückgibt.

Funktion mit zwei Parametern

Schreibe eine Funktion, die die beiden übergebenen Zahlen multipliziert und das Ergebnis zurückgibt.

Funktion ohne Rückgabewert

Was gibt eine Funktion zurück, die kein `return`-Statement enthält?

Funktion ohne Parameter

```
def my_name():  
    print("Aaron Kunert")
```

Funktion mit einem Parameter

```
def double(number):  
    return number * 2
```

Funktion mit zwei Parametern

```
def multiply(number1,number2):  
    return number1 * number2
```

Aggregatzustand von Wasser

Schreibe eine Funktion, die die Temperatur als Parameter erwartet und abhängig von der Temperatur den Aggregatzustand von Wasser ("**fest**", "**flüssig**", "**gasförmig**") als String zurückgibt.

Zusatz: Schaffst Du es ohne die Schlüsselwörter **elif** und **else**?

Aggregatzustand von Wasser

```
def get_state(temp):  
    if temp < 0:  
        return "fest"  
    if temp > 100:  
        return "gasförmig"  
    return "flüssig"
```

Komplexere Übung

Gewichtete Durchschnittsnote

Schreibe eine Funktion, die eine Liste der folgenden Struktur erwartet:

```
grades = [  
    {  
        "subject": "Deutsch",  
        "grade": 14,  
        "is_major": True  
    },  
    # ...  
    {  
        "subject": "Sport",  
        "grade": 11,  
        "is_major": False  
    }  
]
```

Berechne die Durchschnittsnote, wobei Hauptfächer doppelt gewichtet werden sollen.

Gewichtete Durchschnittsnote

```
def weighted_average(grades):  
    weighted_sum = 0  
    weighted_length = 0  
    for grade in grades:  
        if grade["is_major"]:  
            weighted_sum += 2 * grade["grade"]  
            weighted_length += 2  
        else:  
            weighted_sum += grade["grade"]  
            weighted_length += 1  
    result = weighted_sum/weighted_length  
    return result
```

Zinsrechner

Angenommen, Du hast 1000€ so angelegt, dass es darauf jeden Monat 0,5% Zinsen gibt (Haha – als ob!). Schreibe eine Funktion, die einen Geldbetrag erwartet und zurückgibt, nach wie vielen Monaten, dieser Geldbetrag erreicht wurde.

Zinsrechner

```
def months_until_rich(target):  
    month = 0  
    balance = 1000  
    while balance < target:  
        balance *= 1.005  
        month += 1  
    return month
```

Zinsrechner mit Sparrate

Angenommen, Du hast wieder 1000€ so angelegt, dass es darauf jeden Monat 0,5% Zinsen gibt (Haha – als ob!). Zusätzlich gibt es nun jedoch noch eine monatliche Sparrate von 25€, die ebenfalls auf das Konto eingezahlt wird. Schreibe eine Funktion, die einen Geldbetrag erwartet und zurückgibt, nach wie vielen Monaten, dieser Geldbetrag erreicht wurde.

Zinsrechner mit Sparrate

```
def months_until_rich(target):  
    month = 0  
    balance = 1000  
    while balance < target:  
        balance += 25  
        balance *= 1.005  
        month += 1  
    return month
```

Optionale Parameter

Manchmal wirst Du bei Funktionen bemerken, dass einige der Parameter fast immer den gleichen Wert haben. In diesem Fall, möchtest Du diese Parameter nicht bei jedem Aufruf immer hinschreiben, sondern nur dort, wo er vom Standardfall abweicht. Dies ist möglich, wenn man den Standardwert (*default value*) bei der Definition mit angibt.

Wichtig: Bei der Definition müssen die optionalen Parameter immer hinter den Pflichtparametern stehen.

Beispiel

```
def double(number, factor=2):  
    return number * factor
```

Diese Funktion ist sehr vielseitig: Im einfachen Fall verdoppelt sie die eingegebene Zahl. Optional lässt sich der Faktor aber beliebig verändern.

Typischer Einsatzbereich

Oftmals merkt man im Verlauf eines Projektes, dass eine gegebene Funktion nicht flexibel genug ist, dann kann man sie um einen optionalen Parameter erweitern, ohne den bisherigen Code verändern zu müssen.

Fiktives Beispiel

Stell Dir vor, Du baust einen Zinsrechner wie oben, den auch schon einige Deiner Freund*innen, die den gleichen Sparplan haben, ebenfalls verwenden. Eine Deiner Freundinnen hat jedoch eine bessere Bank gefunden, die ihr 0,6% Zinsen gibt. Also erweiterst Du die Funktion, so dass auch die Höhe der Zinsen anpassbar ist.

Jedoch möchtest Du den bisherigen Code nicht verändern. Daher definierst Du den Zinssatz als optionalen Parameter, so dass die Funktion „abwärtskompatibel“ zu ihrer bisherigen Verwendung ist.

Die Definition startet dann mit `def months_until_rich(target, interest=0.5):`

Flexibler Zinsrechner

Erweitere den Zinsrechner, so dass optional der Zinssatz angegeben werden kann.

Flexibler Zinsrechner

```
def months_until_rich(target, interest=0.5):  
    month = 0  
    balance = 1000  
    while balance < target:  
        balance += 25  
        balance *= (1 + interest/100)  
        month += 1  
    return month
```

Flexibler Durchschnittsrechner

Erweitere die Funktion zur Berechnung von gewichteten Durchschnittsnoten so, dass optional der Gewichtungsfaktor angegeben werden kann.

Flexibler Durchschnittsrechner

```
def weighted_average(grades, weight=2):  
    weighted_sum = 0  
    weighted_length = 0  
    for grade in grades:  
        if grade["is_major"]:  
            weighted_sum += weight * grade["grade"]  
            weighted_length += weight  
        else:  
            weighted_sum += grade["grade"]  
            weighted_length += 1  
    result = weighted_sum/weighted_length  
    return result
```

Named Parameters

Hat eine Funktion viele Parameter, von denen etliche optional sind, so kann man einen Parameter statt über die Reihenfolge auch über den Namen übergeben.

Beispiel

```
def my_function(parameter1, parameter2=0, parameter3="x", parameter4=-17):  
    # ...
```

Möchte man jetzt die Funktion mit einem eigenen Wert `parameter1` und `parameter4` aufrufen aber alles andere auf Standard lassen, so geht das wie folgt:

```
my_function(15, parameter4=-20)
```


Ganz flexibler Zinsrechner

Erweitere den Zinsrechner, so dass zusätzlich optional der Startbetrag und die monatliche Sparrate angepasst werden können.

Welchen der Parameter muss man verdoppeln, um am schnellsten 10.000€ zu erreichen?

Ganz flexibler Zinsrechner

```
def months_until_rich(target, interest=0.5, initial_amount=1000, savings_rate=25):  
    month = 0  
    balance = initial_amount  
    while balance < target:  
        balance += savings_rate  
        balance *= (1 + interest/100)  
        month += 1  
    return month
```

Scope

Wo Variablen gültig sind

Problemstellung

Sei `my_variable` eine Variable mit Wert 1. Schreibe eine Funktion, die bei Aufruf die Variable `my_variable` um 1 erhöht. Schreibe eine Funktion, die bei Aufruf die Variable `my_variable` um 1 erhöht.

Wie macht man das?

Das Problem

```
my_variable = 1

def increment():
    my_variable = my_variable + 1

increment()
print(my_variable)
```

Die offensichtliche Lösung funktioniert nicht. Warum nicht?

Experiment I

```
global_variable = 1
```

```
def my_function():  
    local_variable = 5
```

```
my_function()  
print(global_variable)  
print(local_variable)
```

Beobachtung

Eine Variable, die innerhalb einer Funktion definiert wurde, ist auch nur innerhalb der Funktion sichtbar.

Experiment II

```
global_variable = 1
```

```
def my_function():  
    print(global_variable)
```

```
my_function()  
print(global_variable)
```

Beobachtung

Eine *globale* Variable ist auch innerhalb einer Funktion definiert.

Experiment III

```
global_variable = 1

def my_function():
    global_variable = 5
    print(global_variable)

my_function()
print(global_variable)
```

Beobachtung

Eine Variable innerhalb einer Funktion kann den gleichen Namen wie eine Variable außerhalb haben, allerdings ist die innere Variable nur innerhalb der Funktion sichtbar.

Experiment IV

```
global_variable = 1

def my_function():
    print(global_variable)
    global_variable = 5

my_function()
print(global_variable)
```

Beobachtung/Erklärung

Python entscheidet anhand des Kontexts ob `global_variable` eine globale Variable ist, oder eine lokale Variable, die zufällig den gleichen Namen wie eine globale Variable trägt.

Falls Python denkt, dass es sich um eine globale Variable handelt, so kann diese nur gelesen, nicht aber geschrieben (d.h. neu definiert) werden.

Das Eingangsbeispiel

```
my_variable = 1

def increment():
    my_variable = my_variable + 1

increment()
print(my_variable)
```

Erklärung

Da `my_variable` rechts vom Gleichheitszeichen steht, denkt Python, dass es sich um die globale Variable `my_variable` handelt. Da `my_variable` aber auch links vom Gleichheitszeichen steht, wird auch schreibend auf die Variable zugegriffen. Das ist nicht erlaubt.

Mögliche Lösung

```
my_variable = 1
```

```
def increment(var):  
    return var + 1
```

```
my_variable = increment(my_variable)  
print(my_variable)
```

Definition

Der Gültigkeitsbereich einer Variable wird *Scope* genannt.

Scope in Python

In Python unterscheidet man zwischen *global Scope* und *local Scope*. Im local Scope hat man nur Lesezugriff auf den global Scope.

Achtung Ausnahme

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

def append(item):
    my_list.append(item)

append(4)
print(my_list)
```

Erklärung

Da die Variable `my_list` nicht überschrieben wird, sondern nur das referenzierte Objekt verändert wird, erkennt Python dies nicht als Schreibzugriff und erlaubt dieses Vorgehen.

Warum ist der Zugriff auf den Global Scope eingeschränkt?

- Funktionen sollen möglichst wenige Nebeneffekte haben. Wenn eine Funktion den global Scope verändern kann, ist dies ein großer Nebeneffekt.
- Wenn man eine Funktion schreibt, muss man sich keine Gedanken machen, ob ein Variablenname schon vergeben ist.
- Wenn man sich innerhalb einer Funktion den Kontakt zum global Scope reduziert, so ist die Funktion besser zu verstehen, zu warten und zu testen.
- ...