

Programmieren mit Python

Eine Einführung

Dr. Aaron Kunert

aaron.kunert@salemkolleg.de

5. Mai 2021

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)

Ablauf des Kurses

- Mischung aus Vortrag, Live-Coding und Präsenzübungen
- Im Idealfall: Mehr Praxis statt Erklärungen
- Jede Woche gibt's ein Aufgabenblatt → Besprechung in der nächsten Woche
- Kommunikation über Slack: <https://bit.ly/3a5W9fE> (freiwillig)

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

Phasen des Lernens einer Programmiersprache

Phasen des Lernens einer Programmiersprache

1. Annäherung: Fokus auf dem Begreifen der Grundkonzepte

Phasen des Lernens einer Programmiersprache

1. Annäherung: Fokus auf dem Begreifen der Grundkonzepte
2. Syntax: Fokus auf der korrekten Anwendung der Syntax

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

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

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

Was wird benötigt?

Was wird benötigt?

Am Anfang

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

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

Wo findet man Hilfe/Infos?

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

Installation von Python

Ist Python schon installiert ?

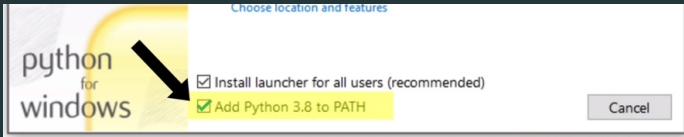
- Öffne ein Terminal/die Eingabeaufforderung
- Gib ein `python --version`
- oder alternativ `python3 --version`
- Erhältst Du die Antwort Python und eine Zahl ≥ 3.6 , dann ist alles fein
- Falls nicht: Installiere Python!

Installation

1. Gehe auf <https://www.python.org/downloads/>
2. Klicke den Button "Download Python 3.9.3."
3. Führe die Installationsdatei aus
4. Falls Du gefragt wirst, bestätige, dass Python zum PATH hinzugefügt wird
5. Eventuell muss der Rechner neu gestartet werden

Achtung bei Windows

Python muss zum PATH hinzugefügt werden.



Cross-Check

Gib `python` (Win) oder `python3` (Mac) im *Terminal* ein. Du solltest etwa folgendes sehen:

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64  
bit (AMD64)] on win32 Type "help", "copyright", "credits" or "license" for  
more information.
```

```
>>>
```

Jetzt bist Du im *interactive mode* (REPL) von Python. Hier kannst Du einzelne Codezeilen eingeben und mittels `Enter` ausführen. Um den interactive mode zu verlassen, gib `exit()` ein und bestätige mit der `Enter`-Taste.

Erste Schritte im REPL

(Read-Evaluate-Print-Loop)

Probier mal folgende Kommandos aus

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

Was machen die folgenden *Operatoren*?

- +
- -
- *
- /
- **

Was machen die folgenden *Operatoren*?

- +
- -
- *
- /
- **

Und diese?

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

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?

Variablen

Jeder Wert in Python kann in einer Variable gespeichert werden:

```
my_variable = 3
```

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
```

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
```

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

Gültige Variablennamen

- Erlaubt sind Buchstaben (nur ASCII), Ziffern und Unterstriche

Gültige Variablennamen

- Erlaubt sind Buchstaben (nur ASCII), Ziffern und Unterstriche
- Der Name darf nicht mit einer Ziffer starten

Gültige Variablennamen

- Erlaubt sind Buchstaben (nur ASCII), Ziffern und Unterstriche
- Der Name darf nicht mit einer Ziffer starten
- Beliebige Länge

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]*`

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

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*)

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?

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.

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. $1.2 - 1.0$?

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.

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"`

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"`

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"`

Typ einer Variablen ermitteln

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

Typ einer Variablen ermitteln

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

Typumwandlung (*Typecasting*)

Typ einer Variablen ermitteln

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

Typumwandlung (*Typecasting*)

Implizit

Bei manchen Operationen nimmt Python automatisch eine Typumwandlung vor.

Beispiel: `1 + 2.0` ergibt `3.0`

Typ einer Variablen ermitteln

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

Typumwandlung (*Typecasting*)

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`

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?

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

Operator-Präzedenz

1. Klammern

Operator-Präzedenz

1. Klammern
2. **

Operator-Präzedenz

1. Klammern
2. **
3. *, /, //, %

Operator-Präzedenz

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

Operator-Präzedenz

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

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

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
```

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
```

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.

Script Mode

Script Mode

Sobald man mehrere zusammenhängende Zeilen hat, wird die 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  
print(f"Hello, I'm {name} and I'm {age} years old")
```

Beispiel

```
name = "Max"  
age = 20  
print(f"Hello, I'm {name} and I'm {age} years old")
```

Problem:

Wie kann man Python erklären, diese 3 Zeilen auf einmal auszuführen?

Old-School-Lösung

- Erstelle eine neue Datei (z.B. `my_script.py`)
- Öffne die Datei mit einem Texteditor und speichere den Beispiel-Code darin ab.
- Öffne den Ordner mit der Datei `my_script.py` mit dem Terminal bzw. der Eingabeaufforderung
- Führe das Kommando `python my_script.py` aus.

Optimallösung: Verwende eine IDE

Eine IDE (integrierte Entwicklungsumgebung) hilft Dir beim Programmieren und unterstützt Dich wo immer möglich. Dadurch lassen sich auch große Projekte schnell umsetzen.

Optimallösung: Verwende eine IDE

Eine IDE (integrierte Entwicklungsumgebung) hilft Dir beim Programmieren und unterstützt Dich wo immer möglich. Dadurch lassen sich auch große Projekte schnell umsetzen.

Nachteile

Die anfängliche Einrichtung kann schnell kompliziert werden. Aufgrund der vielen Features fühlt man sich schnell mal überfordert.

Optimallösung: Verwende eine IDE

Eine IDE (integrierte Entwicklungsumgebung) hilft Dir beim Programmieren und unterstützt Dich wo immer möglich. Dadurch lassen sich auch große Projekte schnell umsetzen.

Nachteile

Die anfängliche Einrichtung kann schnell kompliziert werden. Aufgrund der vielen Features fühlt man sich schnell mal überfordert.

→ Das machen wir etwas später.

Kompromiss für den Anfang: Browserbasierte Editoren

Um schnell einzusteigen, kann zu Beginn auch ein browsergestützter Editor/Interpreter verwendet werden. Zum Beispiel:

Kompromiss für den Anfang: Browserbasierte Editoren

Um schnell einzusteigen, kann zu Beginn auch ein browsergestützter Editor/Interpreter verwendet werden. Zum Beispiel:

- Programiz (<https://www.programiz.com/python-programming/online-compiler>)
 - einfacher Einstieg
 - schnell und unkompliziert
 - geringer Funktionsumfang

Kompromiss für den Anfang: Browserbasierte Editoren

Um schnell einzusteigen, kann zu Beginn auch ein browsergestützter Editor/Interpreter verwendet werden. Zum Beispiel:

- Programiz (<https://www.programiz.com/python-programming/online-compiler>)
 - einfacher Einstieg
 - schnell und unkompliziert
 - geringer Funktionsumfang
- Repl.it (<https://repl.it.com/languages/python3>)
 - Auch für viele andere Sprachen geeignet
 - Manchmal etwas langsam
 - Man kann mehrere Dateien und Projekte verwalten (braucht Account)
 - Hat fast alle IDE-Features (braucht Account)

Input/Output

Kommunikation über die Konsole

Die Konsole

Da wir zu Beginn noch über keine grafische Benutzeroberfläche verfügen, verwenden wir für die Kommunikation mit unserem Programm die *Konsole*. Dabei handelt es sich um ein einfaches Textfenster, auf dem Dein Programm Informationen ausgeben kann (*Output*) und Text einlesen kann (*Input*).

Output

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

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

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
```

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")
```

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.
```

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.

Kommentare

Kommentare

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

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

Struktur if-else Statement

```
if
```

Struktur if-else Statement

if *Bedingung*

Struktur if-else Statement

```
if Bedingung:
```

Struktur if-else Statement

```
if Bedingung:
```

```
    □ □
```

Struktur if-else Statement

if *Bedingung*:

 □ □ *Codezeile A1*

Struktur if-else Statement

if *Bedingung*:

 □□ *Codezeile A1*

 □□ *Codezeile A2*

 □□ ⋮

Struktur if-else Statement

if *Bedingung*:

 □□ *Codezeile A1*

 □□ *Codezeile A2*

 □□ ⋮

else:

Struktur if-else Statement

if *Bedingung*:

 □□ *Codezeile A1*

 □□ *Codezeile A2*

 □□ ⋮

else:

 □□ *Codezeile B1*

 □□ *Codezeile B2*

 □□ ⋮

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.

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.

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.

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 entweder

`Willkommen`

oder

`Du darfst hier nicht rein
erscheinen.`

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 entweder

`Willkommen`

oder

`Du darfst hier nicht rein
erscheinen.`

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.`

Logische Operatoren

Booleans können mittels folgender Operatoren miteinander verknüpft werden:

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.

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`

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 mehrerer, gleichrangiger Verzweigungsäste zu realisieren kann man das `elif`-Conditional verwenden.

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")
```

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
```

Baue einen Bestätigungsdialog

Schreibe ein Skript was einen typischen Bestätigungsdialog simuliert. Zum Beispiel:

```
Are you sure to continue? (y)es/(n)o.
```

Mögliche Antworten sind yes, no bzw. y, n. Daraufhin soll auf der Konsole `confirmed` oder `aborted` erscheinen.

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.

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.

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"
```

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?

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("Enter a number")  
x = int(x)
```

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

Struktur der `for...in` Schleife

Struktur der for...in Schleife

```
for
```

Struktur der `for...in` Schleife

`for` *Variable*

Struktur der `for...in` Schleife

```
for Variable in
```

Struktur der `for...in` Schleife

```
for Variable in range(min, max)
```

Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

```
    □ □ Codezeile 1
```


Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

```
    □ □ Codezeile 1
```

```
    □ □ Codezeile 2
```

Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

```
    Codezeile 1
```

```
    Codezeile 2
```

```
    :
```

Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

```
    Codezeile 1
```

```
    Codezeile 2
```

```
    :
```

Code, der nicht mehr Teil der Schleife ist

Struktur der `for...in` Schleife

```
for Variable in range(min, max):
```

```
    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

Good to know

- Achtung: Die Schleifenvariable erreicht nie das obere Ende der `range`-Funktion, sondern bleibt immer 1 drunter.

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)`.

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).

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
  ⋮
```

7er-Reihe mit beliebigem oberen Ende

Lies eine positive ganze Zahl x ein und gib die 7er-Reihe von 7 bis mindestens x wie oben auf der Konsole aus.

Schleife über einen String

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

Das Gauss-Problem

Berechne die Summe der Zahlen 1 bis 100.

Harte Übung I

Quersumme

Lies eine ganze Zahl x ein und bestimme ihre Quersumme.

Tipp 1: Die Anzahl der Stellen einer Zahl bekommt man mittels `len(str(x))` heraus.

Tipp 2: Man benötigt Tipp 1 gar nicht.

Harte Übung II

Zahlenmuster

Gib folgendes Muster auf der Konsole aus:

```
1
1 2
1 2 3
1 2 3 4
  ⋮
1 2 ... 20
```

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.

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("Enter a number")
```

```
    x = int(x)
```

```
print("Yeah, you picked the right number.")
```

Struktur der while-Schleife

Struktur der while-Schleife

```
while
```

Struktur der while-Schleife

```
while Bedingung
```

Struktur der while-Schleife

`while` *Bedingung*:

Struktur der while-Schleife

`while` *Bedingung*:

□□ Codezeile 1

Struktur der while-Schleife

`while` *Bedingung*:

- □ Codezeile 1

- □ Codezeile 2

Struktur der while-Schleife

```
while Bedingung:
```

```
    □□ Codezeile 1
```

```
    □□ Codezeile 2
```

```
    □□     ⋮
```

Struktur der while-Schleife

`while` *Bedingung*:

 □□ Codezeile 1

 □□ Codezeile 2

 □□ ⋮

Code, der nicht mehr Teil der Schleife ist

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

Schreib ein Programm, dass alle 7er-Zahlen von 7 bis 700 auf der Konsole ausgibt.

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:

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.

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:

Das Spiel soll mit der Eingabe von `q` abgebrochen werden können.

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:

Das Spiel soll mit der Eingabe von `q` abgebrochen werden können.

Zusatz 3:

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

break, continue **und** else

Den Fluss einer Schleife kontrollieren

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.

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
```

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  
# prints 1 2 3 4
```

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.

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)
```

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)  
# prints 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.

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("Your name: ")

for letter in name:
    if letter == "a":
        print("Your name contains an a")
        break
else:
    print("Your name contains no 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 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. Versuche dabei, ein `continue`-Statement zu verwenden.

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.

Listen

Viele Variablen gleichzeitig speichern

Problemstellung

Lies mit Hilfe einer Schleife nach und nach Schulnoten von Dir ein. Alle Noten sollen gespeichert werden. Danach sollst Du die Wahl haben, die soundsovielte Note anzeigen lassen zu können.

Wie macht man das?

Lösung (fast)

```
# ...  
# Um das Eingeben der Noten kümmern wir uns noch  
grades = [12, 10, 7, 14, 13, 13, 6, 4, 15, 14] # Noten in Notenpunkten  
  
index = input("Die wievielte Note möchtest Du nochmal anschauen?")  
index = int(index)  
  
print(f"Deine { index }. Note ist { grades[index] } Punkte")
```

Struktur einer *Liste*

```
my_list =
```

Struktur einer *Liste*

```
my_list = [
```

Struktur einer *Liste*

```
my_list = [element_0
```

Struktur einer *Liste*

```
my_list = [element_0,
```

Struktur einer *Liste*

```
my_list = [element_0, element_1,
```

Struktur einer *Liste*

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

Struktur einer *Liste*

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

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.

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.

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.

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] = -23.`

Schreibzugriff auf Listenelemente

Nach dem gleichen Prinzip lassen sich einzelne Listeneinträge verändern.

Beispiel: `my_list[3] = -23`.

Achtung

Man kann nur schon existierende Listeneinträge verändern.

Schreibzugriff auf Listenelemente

Nach dem gleichen Prinzip lassen sich einzelne Listeneinträge verändern.

Beispiel: `my_list[3] = -23`.

Achtung

Man kann nur schon existierende Listeneinträge verändern.

Neues Konzept

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.

Listeneinträge hinzufügen

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

Bsp: `my_list.append(12)` fügt einen weiteren Eintrag mit Wert 12 hinzu.

Listeneinträge hinzufügen

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

Bsp: `my_list.append(12)` fügt einen weiteren Eintrag mit Wert 12 hinzu.

Listeneinträge entfernen

Listeneinträge hinzufügen

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

Bsp: `my_list.append(12)` fügt einen weiteren Eintrag mit Wert 12 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.

Listeneinträge hinzufügen

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

Bsp: `my_list.append(12)` fügt einen weiteren Eintrag mit Wert 12 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(-23)` entfernt den ersten Eintrag mit dem Wert -23. Ist der Wert nicht vorhanden gibt es eine Fehlermeldung.

Eine Liste erstellen

Schreibe ein kleines Programm, dass Dich ca. 4x nach dem Namen einer Freund*in fragt und Dir am Schluss die Liste der eingegebenen Freund*innen ausgibt.

Das Eingangsproblem

Schreibe ein kleines Programm, dass solange Deine Noten einliest, bis Du **q** drückst. Danach sollst Du die Möglichkeit haben, eine Zahl k einzugeben, so dass Dir die k -te Note angezeigt wird.

Schleife über Liste

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

Schleife über Liste

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

Beispiel

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
for hobby in my_hobbies:  
    print(hobby)
```

Schleife über Liste

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

Beispiel

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
for hobby in my_hobbies:  
    print(hobby)
```

```
# prints:  
# Segeln  
# Tennis  
# Schwimmen  
# Lesen
```

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.

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

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
for index, hobby in enumerate(my_hobbies):  
    print(f"Mein {index + 1}.Hobby ist {hobby}")
```

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

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
for index, hobby in enumerate(my_hobbies):  
    print(f"Mein {index + 1}. Hobby ist {hobby}")
```

```
# prints:  
# Mein 1. Hobby ist Segeln  
# Mein 2. Hobby ist Tennis  
# Mein 3. Hobby ist Schwimmen  
# Mein 4. Hobby ist Lesen
```

Liste durchsuchen

Lies wieder eine Liste Deiner Noten ein. Prüfe, ob Du mindestens einmal unterpunktet hast (d.h. 0 Punkte hattest). Auf der Konsole soll dann entweder

Du hast irgendwo unterpunktet

oder

Du hast nirgendwo unterpunktet

ausgegeben werden.

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.

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

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
sailing_in_list = "Segeln" in my_hobbies
```

```
climbing_in_list = "Klettern" in my_hobbies
```

```
print(sailing_in_list)
```

```
print(climbing_in_list)
```

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

```
my_hobbies = ["Segeln", "Tennis", "Schwimmen", "Lesen"]
```

```
sailing_in_list = "Segeln" in my_hobbies
```

```
climbing_in_list = "Klettern" in my_hobbies
```

```
print(sailing_in_list)    # True
```

```
print(climbing_in_list)  # False
```

Eine Liste sortieren

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

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()`.

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.

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)
```

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]
```

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)
print(sorted_list)
```

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)
```

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

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

Lies wieder ein paar Noten ein. Gib dann auf der Konsole einmal die beste und einmal die schlechteste Note aus.

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>

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

Lies wieder ein paar Noten ein. Gib auf der Konsole die Durchschnittsnote aus.

Slicing

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

Slicing

Wenn man eine Liste hat, ist es oft nötig, einen Teil der Liste „auszuschneiden“. Dafür hat Python die *Slice-Notation* eingeführt.

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:

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].
```


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

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.

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]
```

```
my_list[0:4]
```

```
my_list[1:1]
```

```
my_list[0:4:2]
```

```
my_list[:3]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

Beispiele

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

```
my_list[1:3]      # [4, 6]
```

```
my_list[0:4]
```

```
my_list[1:1]
```

```
my_list[0:4:2]
```

```
my_list[:3]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

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]
```

```
my_list[:3]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

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]
```

```
my_list[:3]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

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]
```

```
my_list[:3]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

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]
```

```
my_list[2:]
```

```
my_list[:]
```

```
my_list[1:-2]
```

```
my_list[-3:-1]
```

```
my_list[::-1]
```

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:]
my_list[:]
my_list[1:-2]
my_list[-3:-1]
my_list[::-1]
```

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[:]
my_list[1:-2]
my_list[-3:-1]
my_list[::-1]
```

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]
my_list[-3:-1]
my_list[::-1]
```

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

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

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

Dictionaries

Problemstellung

In einem Notenrechner wie eben, sollen nicht nur die Noten gespeichert werden, sondern auch das Fach, in dem die Note erreicht wurde.

Wie macht man das?

Lösung

```
## ...
```

```
grades = { "Deutsch": 14, "Mathematik": 8, "Biologie": 11, "Sport": 13}
```

```
key = input("Welche Note möchtest Du wissen?")
```

```
print(f"Deine Note in { key } ist { grades[key] } Punkte")
```

Struktur eines *Dictionary*

```
my_dict =
```

Struktur eines *Dictionary*s

```
my_dict = {
```

Struktur eines *Dictionary*s

```
my_dict = {key_1
```

Struktur eines *Dictionary*s

```
my_dict = {key_1:
```

Struktur eines *Dictionary*s

```
my_dict = {key_1:value_1
```

Struktur eines *Dictionary*s

```
my_dict = {key_1:value_1,
```


Struktur eines *Dictionaries*

```
my_dict = {key_1:value_1, key_2:value_2,
```

Struktur eines *Dictionaries*

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

Struktur eines *Dictionaries*

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

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

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
grades = {  
    "Deutsch": 14,  
    "Mathematik": 8,  
    "Biologie": 11,  
    "Sport": 13  
}
```

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
grades = {  
    "Deutsch": 14,  
    "Mathematik": 8,  
    "Biologie": 11,  
    "Sport": 13  
}
```

- Dictionaries sind mutable, können also verändert werden.

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
grades = {  
    "Deutsch": 14,  
    "Mathematik": 8,  
    "Biologie": 11,  
    "Sport": 13  
}
```

- Dictionaries sind mutable, können also verändert werden.
- Dictionaries besitzen keine vernünftige Anordnung und können nicht geordnet werden.

Good to know

- Zur besseren Übersichtlichkeit werden Dictionaries oftmals wie folgt formatiert:

```
grades = {  
    "Deutsch": 14,  
    "Mathematik": 8,  
    "Biologie": 11,  
    "Sport": 13  
}
```

- 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:

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

Auf Dictionary-Elemente zugreifen

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

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.

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 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 vier 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.

Einen Eintrag aus einem Dictionary entfernen

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

```
del my_dict["a"]
```


Einen Eintrag aus einem Dictionary entfernen

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

```
del my_dict["a"]
```

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.

Schleife über Dictionary I

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

Beispiel

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

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

Schleife über Dictionary I

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

Beispiel

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

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

```
# Mathe
```

```
# Bio
```

```
# Sport
```

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.

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

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

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

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

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

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

```
# Mathe:12
```

```
# Bio:11
```

```
# Sport:13
```

Zwei Dictionaries kombinieren

Gegeben seien zwei Dictionaries, z.B.

```
majors = {"Deutsch": 11, "Mathe": 7}
```

und

```
minors = {"Sport": 14, "Bio": 8 }
```

Füge die Einträge des zweiten Dictionaries zum ersten Dictionary hinzu.

Zwei Dictionaries kombinieren

Gegeben seien zwei Dictionaries, z.B.

```
majors = {"Deutsch": 11, "Mathe": 7}
```

und

```
minors = {"Sport": 14, "Bio": 8 }
```

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.

Ein Dictionary zerlegen

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

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.

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.

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

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

```
grade_subjects = grades.keys()  
grade_subjects = list(grade_subjects)
```

```
grade_numbers = grades.values()  
grade_numbers = list(grade_numbers)
```

```
print(grade_subjects)  
print(grade_numbers)
```

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

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

```
grade_subjects = grades.keys()  
grade_subjects = list(grade_subjects)
```

```
grade_numbers = grades.values()  
grade_numbers = list(grade_numbers)
```

```
print(grade_subjects)  # ["Mathe", "Bio", "Sport"]  
print(grade_numbers)  # [8, 11, 13]
```
