Programmieren mit Python

Eine Einführung

Dr. Aaron Kunert
aaron.kunert@salemkolleg.de

14. April 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 ?

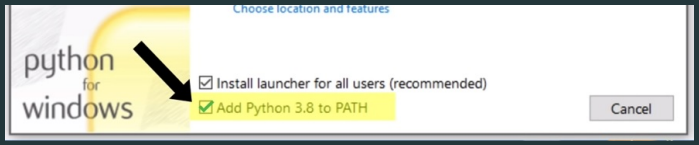
- Öffne ein Terminal/die Eingabeaufforderung
- Gib ein `python --version`
- oder alternativ `python3 --version`
- Erhältst Du die Antwort `Python` und eine Zahl ≥ 3.6, dann ist alles fein
- Falls nicht: Installiere Python!

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"`

Übung

Was machen die folgenden *Operatoren*?

- `+`
- `-`
- `*`
- `/`
- `**`

Übung

Was machen die folgenden *Operatoren*?

- `+`
- `-`
- `*`
- `/`
- `**`

Und diese?

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

Übung

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 aussdrucksstarke 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 aussdrucksstarke 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`

Übung

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`

Übung

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?

Übung

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:

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

Kombinierte Zuweisung

Oft möchte man eine gegebene Variable neu zuweisen:

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

Dies lässt sich auch kurz schreiben als

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

Kombinierte Zuweisung

Oft möchte man eine gegebene Variable neu zuweisen:

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

Dies lässt sich auch kurz schreiben als

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

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

Input/Output

Teil 1

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:

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

String Interpolation

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

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

String Interpolation

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

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

Das geht auch als *inline expression*:

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

Output

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

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

Output

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

```
1 age = input("How old are you?")
2 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.

Script Mode

Script Mode

Sobald man mehrere zusammenhängende Zeilen hat, wird Pythons *interactive mode* sehr unübersichtlich. Daher gibt es auch die Möglichkeit, alle Programmzeilen in eine Text-Datei zu schreiben und diese gebündelt auszuführen.

Ein erstes Beispiel

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

Speichere diesen Code in der Datei `my_script.py` ab.

Führe danach in diesem Ordner das Kommando `python my_script.py` aus.

Übung

Schreibe ein kurzes Skript, dass 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.
```

Kommentare

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

Beispiel

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

Arbeit mit einer IDE

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting
- Code-Inspection

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting
- Code-Inspection
- Autocomplete

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting
- Code-Inspection
- Autocomplete
- Geile Shortcuts

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting
- Code-Inspection
- Autocomplete
- Geile Shortcuts
- Code direkt ausführen

Integrierte Entwicklungsumgebungen (IDE)

Die Arbeit mit gewöhnlichen Texteditoren ist auf Dauer sehr mühsam. Daher empfiehlt es sich eine IDE zu verwenden. Das bringt zum u.a. folgende Vorteile:

- Syntax-Highlighting
- Code-Inspection
- Autocomplete
- Geile Shortcuts
- Code direkt ausführen
- Hilfe bei der Fehlersuche (*Debugging*)

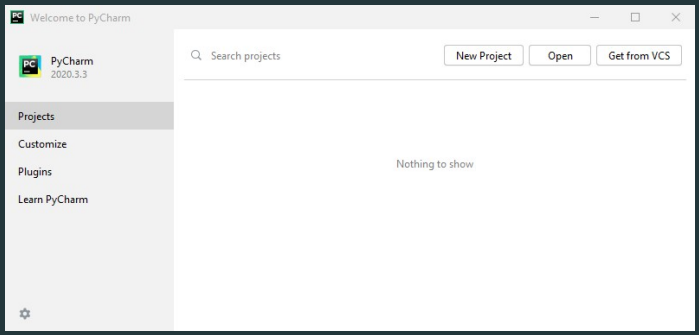
Installation von PyCharm

1. Gehe auf `https://www.jetbrains.com/pycharm/download`
2. Lade die kostenlose *Community Edition* herunter
3. Führe den Installer aus
4. Öffne PyCharm

Installation von PyCharm

1. Gehe auf `https://www.jetbrains.com/pycharm/download`
2. Lade die kostenlose *Community Edition* herunter
3. Führe den Installer aus
4. Öffne PyCharm

Wenn alles passt, sollte es etwa so aussehen:



Installation PyCharm

1. Gehe auf `Customize > All Settings...`
2. Einstellungen synchronisieren
   2.1 `Tools > Settings Repository`
   2.2 Unter *Read-only Sources* auf +
   2.3 `https://github.com/a-kunert/ide-settings.git` eingeben
3. Verknüpfe den Interpreter
   3.1 In den Settings auf `Python Interpreter`
   3.2 Falls möglich unter `Python Interpreter` einen Interpreter wählen. Ansonsten wie folgt:
   3.3 `Zahnrad > Add`
   3.4 `System Interpreter`
   3.5 Dort den Pfad zu Python angeben
4. Mit dem Button *Apply* alles bestätigen

Konfiguration abschließen

1. Lege eine Ordner für den Kurs an
2. Öffne diesen Ordner mit `Projects > Open`
3. `File > Manage IDE Settings > Sync with Settings Repository > Merge` ausführen
4. Bei `File > Settings` unter `Keymap` die Keymap *Salem-Win/Mac* auswählen.
5. Code in die Datei `main.py` schreiben
6. Mittels grünem Pfeil (oben rechts) Code ausführen

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)

```
1  x = int(input("Gib eine Zahl x an"))
2
3  if x < 0:
4          print("x ist größer 0")
5  else:
6          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
          ⋮
```

Struktur if-else-statement

```
if Bedingung:
....Codezeile A1
....Codezeile A2
....          ⋮
else:
....Codezeile B1
....Codezeile B2
....          ⋮
Codezeile C1
        ⋮
```

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

Wie funktioniert's?

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

Good to know

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

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`

Übung

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`

Übung

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.

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

```
1  if x < 0:
2      print("x is < 0")
3  elif x == 0:
4      print("x is 0")
5  elif x == 1:
6      print("x is 1")
7  else:
8      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

```
1  if x < 0:
2      print("x is < 0")
3  elif x == 0:
4      print("x is 0")
5  elif x == 1:
6      print("x is 1")
7  else:
8      print("x is not negative but neither 0 nor 1")
```

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

Für welche `x` 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
```

Der *ternary Operator*