

# Einstieg in die Programmierung mit Python

Kapitel 0 | [Vorbereitung](#)

Kapitel 1 | [Vorwort](#)

Kapitel 2 | [Der interaktive Modus](#)

Kapitel 3 | [Variablen](#)

Kapitel 4 | [Datentypen](#)

Kapitel 5 | [Operatoren](#)

Kapitel 6 | [print\(\), Module & Overloading](#)

Kapitel 7 | [input\(\) & Call-by](#)

Kapitel 8 | [Verzweigungen](#)

Kapitel 9 | [Schleifen & Rekursion](#)

Kapitel 1 | [Übungsaufgaben](#)

Kapitel 4 | [Übungsaufgaben](#)

Kapitel 6 | [Übungsaufgaben](#)

Kapitel 8 | [Übungsaufgaben](#)

Kapitel 9 | [Übungsaufgaben\\_Schleifen](#), [Übungsaufgabe\\_Rekursion](#)

# Einstieg in die Programmierung mit Python

Kapitel 10 | [Listen & Sortieralgorithmen](#)

Kapitel 11 | [Dictionaries](#)

Kapitel 12 | [RegEx & List Comprehension](#)

Kapitel 13 | [Lesen aus Datei](#)

Kapitel 14 | [Schreiben in Dateien](#)

Kapitel 15 | [OOP - Klassen, Objekte & Konstruktoren](#)

Kapitel 16 | [OOP - Getter & Setter](#)

Kapitel 17 | [OOP - Vererbung](#)

Kapitel 18 | [OOP - ABC \(Abstract Base Classes\)](#)

 Kapitel 19 | [OOP - Polymorphie](#)

Kapitel 20 | [Fehlerbehandlung](#)

Kapitel 10 | [Übungsaufgaben](#) , [Übungsaufgabe2](#)

Kapitel 11 | [Übungsaufgaben](#)

Kapitel 12 | [Übungsaufgaben](#)

Kapitel 13 | [Übungsaufgaben](#)

Kapitel 14 | [Übungsaufgaben](#)

Kapitel 15 | [Übungsaufgaben](#)

Kapitel 16 | [Übungsaufgaben](#)

Kapitel 17 | [Übungsaufgaben](#)

 Kapitel 18 | [Übungsaufgaben](#)

 Kapitel 20 | [Übungsaufgaben](#)

# Einstieg in die Programmierung mit Python

Kapitel 21 | [Testen](#)

Kapitel 22 | [Tkinter](#)

Kapitel 23 | [Diverse Übungsaufgaben](#)

[Feedback](#)

[Projekt - Hinweise](#)

[Projekt 1 | Seriennummerngenerator](#)

[Projekt 2 | RPG-Fighting Game](#) (baut auf Aufgabe K9 | A12 auf)

[Projekt 3 | Spielverwaltung](#)

Kapitel 21 | [Übungsaufgaben](#)


Kapitel 22 | [Übungsaufgaben](#)

Umfang | Spaßfaktor



# Einstieg in die Programmierung mit Python

Hinweis(e):

1. Im Skript können Fehler enthalten sein. Wenn ihr einen Fehler findet, schreibt mir bitte eine E-Mail an:  
[joshua.schumacher@bbs1-mainz.de](mailto:joshua.schumacher@bbs1-mainz.de)
2. Kapitel, die mit einem  versehen sind, existieren gegenwärtig noch nicht oder bedürfen einer Überarbeitung!
3. Gerade in der Informatik werden viele englischsprachige Begriffe verwendet. Je nach Kontext solltest du diese lernen, da auch die **Schlüsselwörter** entsprechend englisch sind (z. B. if, for, try, ...).
4. Das Skript ersetzt kein Buch oder Internetseite, die das Thema „vollständig“ behandelt. Es dient lediglich als roter Faden für den Unterricht. Weiterführende Literatur **muss in jedem Fall** gelesen/angesehen werden!
5. Lösungen zu den Aufgaben können über folgendes Repository bezogen werden:  
[https://github.com/JoshuaSchumacherGER/Python\\_Loesungen.git](https://github.com/JoshuaSchumacherGER/Python_Loesungen.git)

*Hinweis: Lösungen werden erst im Schuljahr 2021/2022 von den Schülern erstellt und hochgeladen!*

Ein wichtiges Vorwort: Python bietet häufig eine Vielzahl von Möglichkeiten, Problemstellungen zu lösen. Dieses Skript bzw. der Unterricht soll Anfängern das Programmieren näherbringen. Daher werden teilweise Fachwörter vereinfacht oder Strategien verwendet, die man in der Praxis so nicht einsetzen würde.

Für Anfänger sind allerdings komplexe RegEx-Ausdrücke, Polymorphie, Exceptionhandling, Patterns usw. schwierig zu begreifen. Daher sind die Beispiele bzw. der Text auf dieser einfache Art und Weise formuliert.

Jeder fortgeschrittene Leser möge mir daher verzeihen 😊. Zudem spielen gewisse (fortgeschrittene) Thematiken keine Rolle im Kurs und werden separat (fall erforderlich) von mir vermittelt.

Dazu gehören:

GIT, CI/CD-Ansätze, Entwicklungsmuster, UML bzw. das Themengebiet SE allgemein, Testverfahren im Detail, WebStack, API's, DevOps-Aufgaben.

# Kapitel 0 | Vorbereitungen

Für den Unterricht werden folgende Programme benötigt:

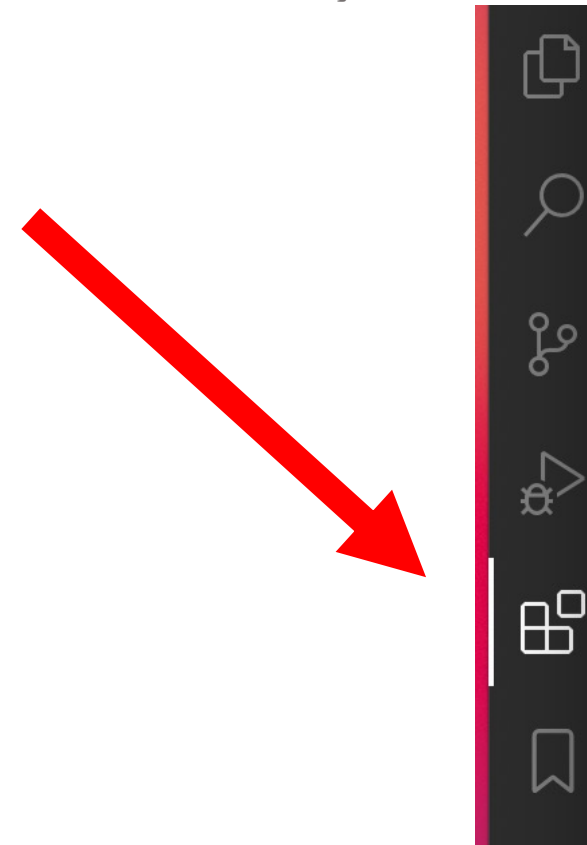
- Python 3 (<https://www.python.org/downloads/>)
  - Python 3 **vor** VSCode installieren!
- Visual Studio Code (<https://code.visualstudio.com/download>)
  - IDE für den Unterricht

## Visual Studio Code

Plugins/Extensions erleichtern die Arbeit mit VSCode und können bei Bedarf installiert werden:

*Hinweis: Leistungsfeststellungen werden unter Umständen ohne Computer geschrieben. „Autoformat“ & „Codevervollständigungs“-Plugins sollten erst eingesetzt werden, wenn man die Syntax & Semantik beherrscht!!!*

Plugins werden über folgendes Symbol in VSCode installiert



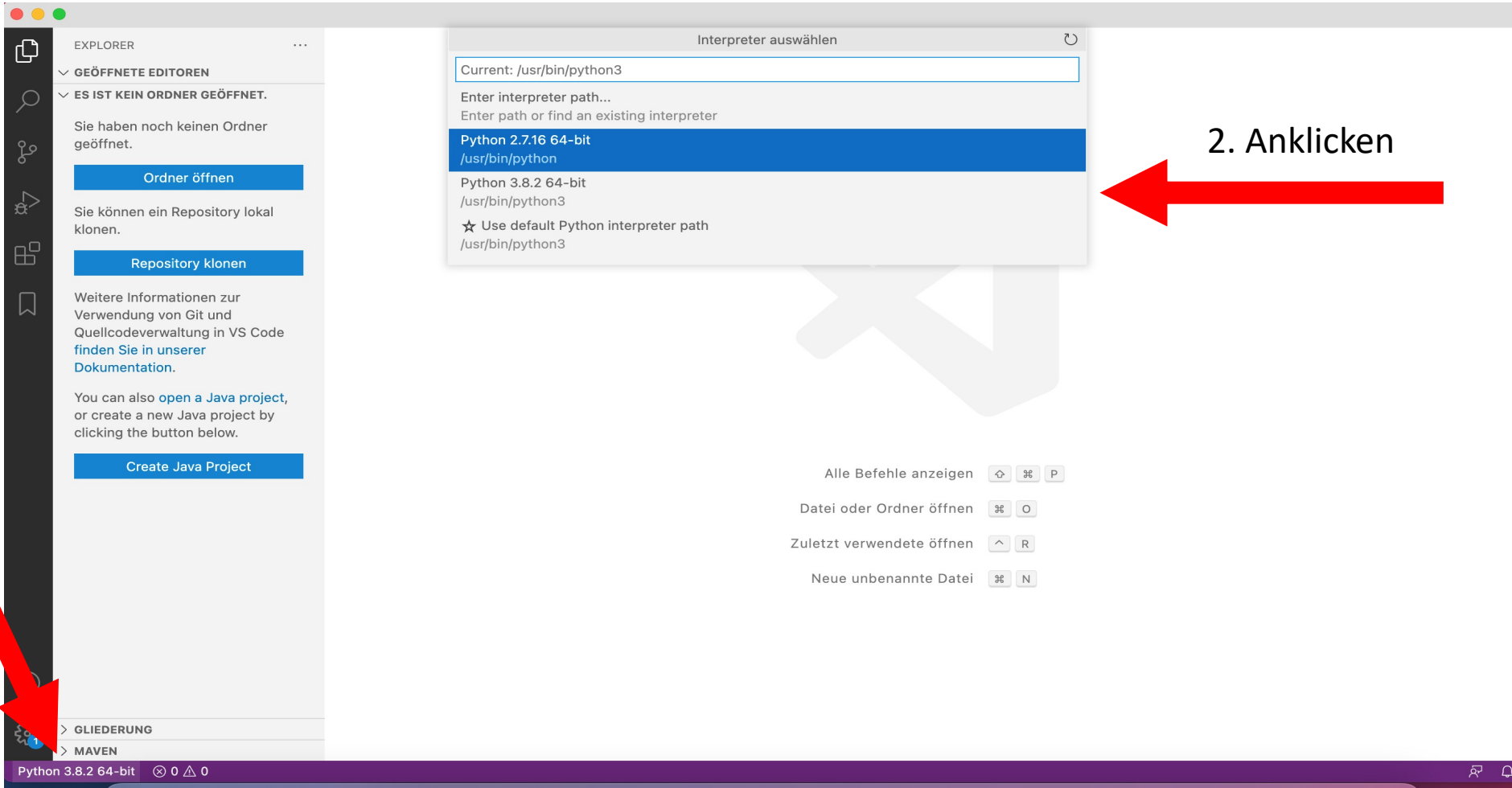
Plugin-Liste (zwingend sollte lediglich 1. installiert werden! Der Rest ist optional):

1. **Python** !!! (sehr wichtig, da VSCode mehrere Programmiersprachen unterstützt. Ansonsten fehlen wichtige Eigenschaften)
2. Python Indent (rückt den Code entsprechend ein)
3. Bracket Pair Colorizer (buntes Einfärben der Klammern für bessere Sichtbarkeit)
4. Bookmarks (markiert Zeilen für eine bessere Übersicht bei größeren Projekten)
5. TODO Highlight (siehe. 4)
6. Git Graph (zeigt einen Graph vom aktuellen Repository)
7. GitLens (für bessere Verwendung von Git)
8. Ayu (Theme)
  
9. Flake 8 (keine Extension, wird über die JSON.Settings aktiviert. Sorgt für qualitativ besseren Code)
10. Autoformat (keine Extension, wird über die Settings aktiviert „Format on Save“)



# Kapitel 0 | Vorbereitungen

Python existiert in der Version 2.X und 3.X. Diese sind **nicht** miteinander kompatibel. Wir verwenden Python 3.X. Je nach Betriebssystem sind mehrere Interpreter installiert. Daher muss darauf geachtet werden, dass der richtige Interpreter ausgewählt wird! Im nachfolgenden Screenshot ist zu sehen, wie man den richtigen Interpreter auswählt.



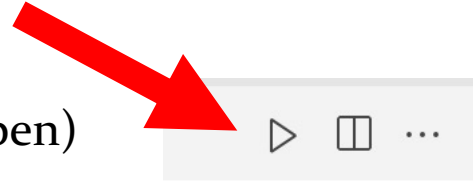
1. Anklicken

2. Anklicken

Geschriebene Programme können auf vielfältige Weise gestartet werden.

1. Möglichkeit:

Klicke auf den „Play“-Button (rechts oben)



2. Möglichkeit:

Im Terminal (VSCode): `python dateiname.py` eintippen.

3. Möglichkeit:

Das Plugin Coderunner installieren und danach mit dem Shortcut `Ctrl+Opt+N` (Mac) starten.

Am Anfang stellt sich häufig die Frage, mit welcher Programmiersprache wir programmieren lernen sollten. Googeln wir diese Frage, werden wir von einer schier unbegrenzten Anzahl an Möglichkeiten erschlagen. Hier eine kleine

Auswahl: Java, Python, C++, C, VisualBasic, Delphi, Pascal, Haskell, Kotlin, Ruby, Javascript, PHP, ...

Hier müssen wir uns das Ziel klar machen, was wir erreichen wollen. Je nach Szenario bieten sich einige Programmiersprachen mehr oder weniger an. Unser Ziel ist es, eine möglichst einfache Programmiersprache zu wählen, die in der Praxis **häufig** verwendet wird und die möglichst vielseitig ist.

**Python** erfüllt diese Punkte. Daher werden wir uns in den kommenden Monaten mit der Programmiersprache Python beschäftigen.

Randnotiz: Python ist aktuell (2022) die beliebteste Programmiersprache (Pypl.github.io)

- Python (31,02%) – Top 1
- Java (16,38%) – Top 2
- Javascript (8,41%) – Top 3

Ein direkter Vergleich zwischen Python & Java. Bei komplexeren Programmen wird der Unterschied immer extremer (**Boilerplate-Code**). Um das Java Programm vollständig zu begreifen, sind Kenntnisse in OOP, Datentypen, Funktionen, Methoden usw. nötig. Das ist einer (von vielen Gründen), warum Python als Einsteigersprache sinnvoll ist. Java ist nicht mächtiger, was der erste Gedanken sein könnte. Im Gegenteil, es gibt Bereiche (Machine Learning, Datenanalyse, ...) bei denen Python die Sprache Nummer 1 ist und Java keine nennenswerte Rolle spielt.

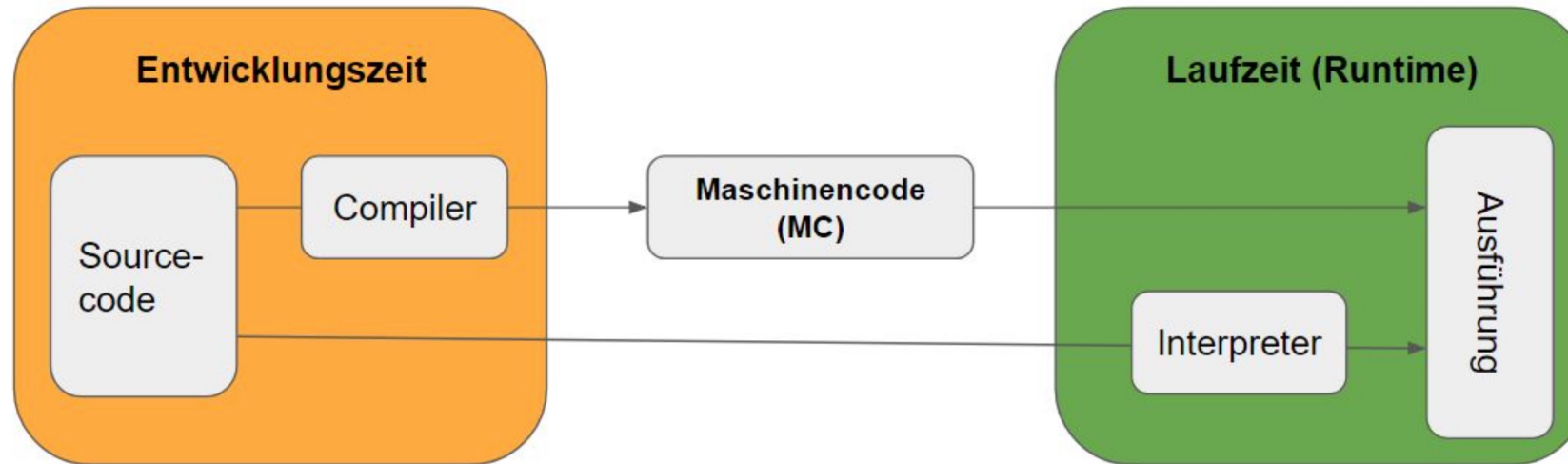
hello\_world.py

```
1 print("Hello World")
```

Hello\_world.java

```
1 public class Hello_World{
2     public static void main(String[] args){
3         System.out.println("Hello World");
4     }
5 }
```

Programmiersprachen können grob in zwei Klassen unterteilt werden. **Kompilierende** und **interpretierende** Sprachen.



[3]: <https://www.data-science-architect.de/python-kompiliert-interpretiert/>

Bei kompilierenden Sprachen wird der Quellcode (Sourcecode) mithilfe eines Compilers in Maschinencode übersetzt. Dieser Maschinencode kann vom Computer ausgeführt werden.

Vorteil: - Maschinencode kann sehr schnell ausgeführt werden.

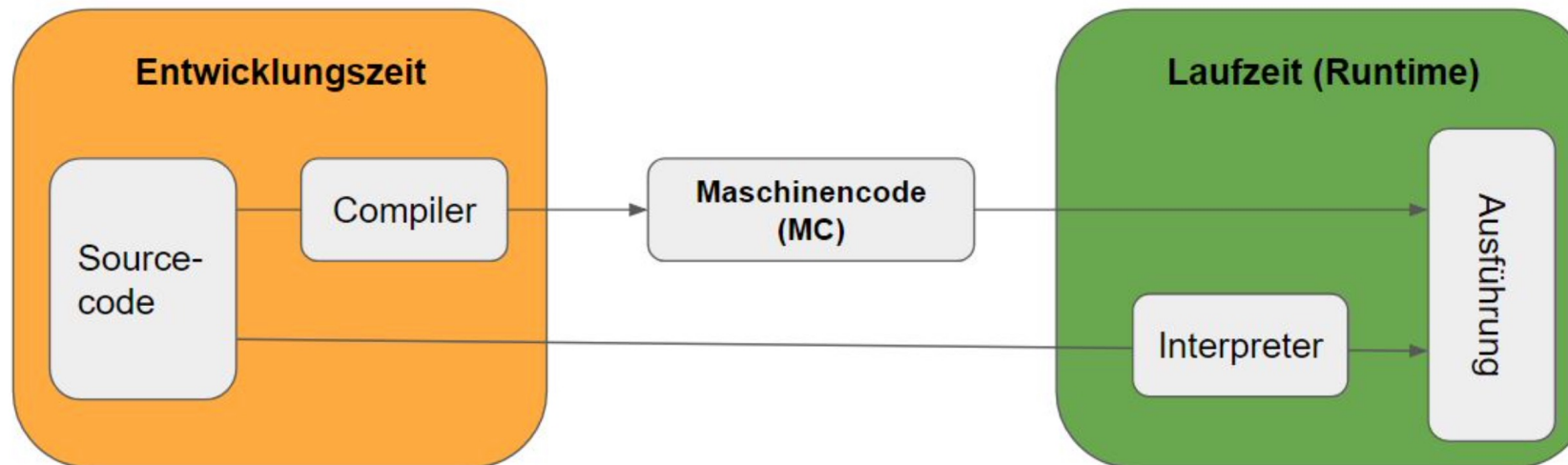
Nachteil: - Plattformabhängigkeit! (Der Quellcode muss also auf den einzelnen Plattformen (Windows, MacOS, Linux, ...) separat kompiliert werden!)

# Kapitel 1 | Vorwort

Bei interpretierenden Sprachen wird der Quellcode (Sourcecode) mithilfe eines Interpreters direkt ausgeführt (siehe Grafik).

Vorteil: - Plattformunabhängigkeit, User-Daten-Interaktion vereinfacht!

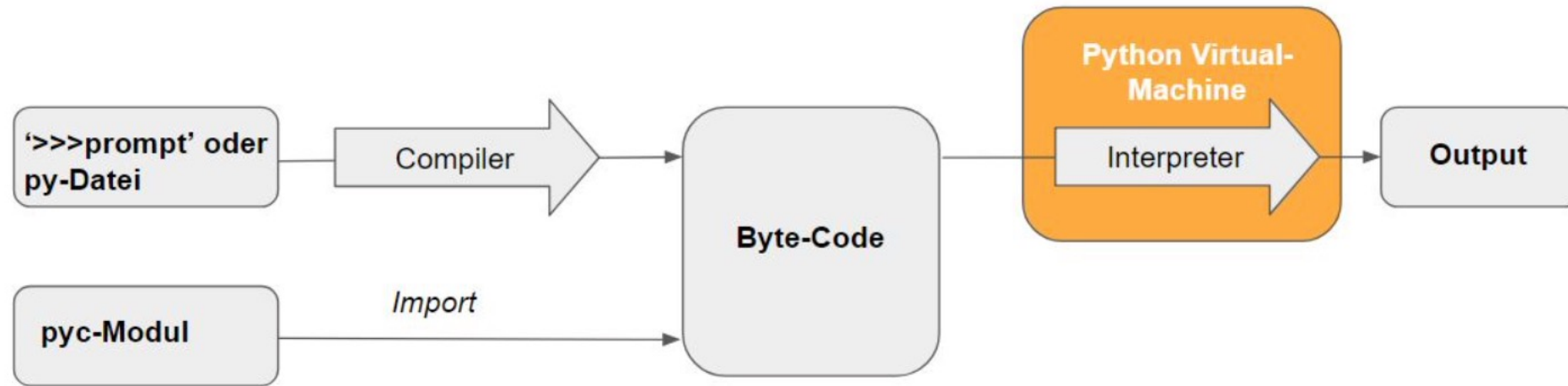
Nachteil: - Im vgl. zu kompilierenden Sprachen sehr langsam!



[3]: <https://www.data-science-architect.de/python-kompiliert-interpretiert/>

# Kapitel 1 | Vorwort

Python nimmt hier eine Sonderstellung ein. Es nutzt beide Technologien.



[3]: <https://www.data-science-architect.de/python-compiliert-interpretiert/>

.py Dateien werden mithilfe eines Compilers in einen Bytecode übersetzt (ähnlich zu Java). Dieser läuft im zweiten Schritt über einen Interpreter (im Vgl. wird in Java der Bytecode von der JRE-Umgebung ausgeführt).

Vorteil: - Plattformunabhängigkeit, Sicherheitstechnische Aspekte

Nachteil: - Langsamer als rein kompilierende Sprachen (z. B. C++)

# Kapitel 1 | Aufgaben

**Aufgabe K1 | A1:** Bevor wir loslegen können benötigst du die beiden Programme aus Folie 5. Installiere zuerst Python in der Version 3.x (***WICHTIG: Python 3.X, nicht 2.x!***)

**Aufgabe K1 | A2:** Installiere danach VSCode mit dem entsprechenden Python-Plugin!



In der Shell können beispielsweise Rechnungen schnell und einfach ausgeführt werden. Gebe dort als Test 3+5 ein und bestätige die Eingabe mit der Entertaste. Daraufhin erscheint das Ergebnis 8. Dieser Modus dient vor allem dazu, schnell Änderungen testen zu können ohne das Programm kompileieren zu müssen.

Kompilieren bedeutet das **Übersetzen** von dem **Quellcode** in (den für den Computer lesbaren) **Maschinencode**. Wir könnten theoretisch auch direkt Maschinencode schreiben und damit den oben genannten Vorgang sparen. Allerdings wäre das umständlich und nervig. Programmiersprachen wie Python machen uns diesen Schritt wesentlich einfacher!

Für komplexere Programme eignet sich der interaktive Modus nicht. Hierfür müssen wir unseren Quellcode in das Hauptfenster schreiben und entsprechend starten.

*Hinweis: Pythondateien sollten immer unter der Endung .py gespeichert werden.*

`i = 42`

Hier ist eine Variable (ein Behälter) mit dem Namen „i“ erstellt (in der Fachsprache auch „**deklariert**“). Daraufhin wird dieser Variablen der Wert 42 zugewiesen. Sie wird mit diesem Wert „**intialisiert**“.

In einer Variablen können nicht nur Zahlen, sondern auch Texte gespeichert werden.

Bsp: `name = "Patrick"`

Wichtig zu wissen: In Python ist es nicht notwendig anzugeben, was in der Variablen gespeichert werden möchte (bspw. Text oder Zahlen). In anderen Sprachen kann es notwendig sein, diesen sogenannten „Datentyp“ anzugeben (bspw. Java).

Java: `int i = 42;`

Python: `i = 42`

Außerdem ist es in Python durchaus möglich, den Datentyp einer Variable zu wechseln (in Java bspw. ist das nicht möglich!). Folgender Code wäre in Python valide:

`name = "Patrick"`

`name = 42`

Variablen dienen uns also als Datenspeicher. Python unterscheidet hier ganze Zahlen, Kommazahlen, Wahrheitswerte, Zeichenketten und Tupel (*weitere Typen wie komplexe Zahlen spielen für uns keine Rolle und sind hier nicht aufgeführt*).

Hier ist jeweils ein Beispiel:

```
i = 42    # ganze Zahl  
Name = "Hugo" # Zeichenkette  
a = 4.2   # Kommazahl  
b = true  # Wahrheitswert  
Tupel = ("Apfel","Birne",5)
```

Übrigens: Mit **#** kannst du Kommentare in Python schreiben. Kommentare dienen dazu, die Programmfunktionalität zu erläutern oder andere Informationen festzuhalten. Kommentare werden von Python ignoriert. Du musst dort also nicht auf die Syntax achten!

Wie bereits beschrieben ist es in Python nicht notwendig, einen expliziten Datentyp anzugeben. Dennoch ist es wichtig, Kenntnisse über Datentypen zu haben. Es gibt Situationen (z. B. beim Einlesen von Daten (siehe Kapitel Funktion – Input())), bei denen der angenommene Datentyp nicht dem entspricht, den wir haben möchte.

Um herauszufinden, welcher Datentyp einer Variable zugeordnet wurde, können wir die Funktion `type()` verwenden. Hierfür schreiben wir den Namen der Variable innerhalb der runden Klammern.

Beispiel:

```
>>> x = 5
>>> type(x)
<class 'int'>
```

Die Variable x wird (in diesem Fall) dem Datentyp „int“ (Integer = ganze Zahl) zugeordnet (Hinweis: Das Schlüsselwort class kannst du hier erst einmal ignorieren). In vielen Fällen stimmt der angenommene Datentyp. Manchmal möchten wir diesen allerdings ändern. Wie das funktioniert, erfährst du später. Zu diesem Zeitpunkt reicht es, wenn dir klar ist, dass es verschiedene Datentypen gibt und welche die geläufigsten sind.

**Aufgabe K4 | A1:** Versuche mit den verschiedenen Datentypen zu arbeiten und lasse dir die jeweiligen Datentypen mit der Funktion `type()` anzeigen. Was passiert, wenn du `5 + 5` eingibst? Was passiert, wenn man verschiedene Datentypen versucht miteinander zu mischen (z. B. `5 + 4.2` . In diesem Fall wird ein Integer mit einem Float addiert. Funktioniert das)?

**Aufgabe K4 | A2:** Gebe nacheinander in der Shell folgendes ein:

```
x = 50
```

```
x = "Patrick"
```

Was ist hier passiert? Halte deine Antwort schriftlich fest und argumentiere mit den Fachbegriffen, die im Skript genannt werden.

In Python existieren eine Vielzahl an Operatoren, von denen du die meisten bereits aus der Schule kennst.

## Kapitel 5 | Arithmetische - Operatoren

Addition ( + )	Subtraktion ( - )	Multiplikation ( * )
Division ( / )	Modulo ( % )	Inkrement ( ++ )*
Dekrement ( -- )*		

*\*in Python via += 1 realisiert*

Für einige ist der Modulo-Operator, das Inkrement und Dekrement neu. Daher jeweils ein Beispiel:

Das Inkrement erhöht eine Variable immer um +1. Damit ist eine verkürzte Schreibweise möglich (Programmierer lieben Abkürzungen!).

Beispiel ohne Inkrement:

```
a = 5  
a = a + 1
```

Mit Inkrement-Operator:

```
a = 5  
a += 1 # hier wird der Inhalt der Variable um +1 erhöht! Der Dekrement-Operator  
funktioniert entsprechend analog.
```

# Kapitel 5 | Operatoren

Der Modulo-Operator ( % ) liefert den Rest einer Division.

z. B. liefert  $9 \% 2 = 1$  # 9 / 2=4 mit Rest 1

Häufig fragen sich Beginner, wofür man Modulo in der Praxis verwenden sollte?  
Fällt dir ein Beispiel ein?

## Kapitel 5 | Relations - Operatoren

Unabhängig davon, welche Art von Software wir programmieren möchten, benötigen wir die Möglichkeit, Werte miteinander vergleichen zu können. Ob beispielsweise Spieler A oder B gewonnen hat, können wir nur herausfinden, wenn wir die Punktzahl von A und B miteinander in Relation setzen. Hierfür stehen uns in Python folgende Operatoren zur Verfügung.

Gleich ( == )	Ungleich ( != )	Größer ( > )
Größer gleich ( >= )	Kleiner ( < )	Kleiner gleich ( <= )

Hier gibt es den ersten Unterschied zur Mathematik. In der Mathematik bedeutet der Ausdruck  $a = 5$ , dass die Variable  $a$  gleich der Zahl 5 ist. In Python hingegen bedeutet  $a = 5$ , dass der Wert 5 auf der rechten Seite der Variable  $a$  auf der linken Seite [zugewiesen](#) wird. Das heißt, wir können mit einem einzelnen  $=$  leider nichts „gleich“ setzen, da die Bedeutung bereits für die Zuweisung vergeben wurde. Daher verwenden wir in Python ein doppeltes Gleichheitszeichen ( $==$ ), um einen Vergleich durchzuführen. Wollen wir also vergleichen, ob in der Variable  $a$  die Zahl 5 steht, schreiben wir  $a == 5$ .

Eine weitere wichtige Unterscheidung ist der Unterschied zwischen  $==$  und **is**.  
Der Ausdruck „ $a == b$ “ vergleicht den Wert zwischen zwischen beiden Variablen.  
Der Ausdruck „ $a$  **is**  $b$ “ vergleicht die Objektidentität.

Der Unterschied wird im Kapitel „Listen“ deutlich!



## Kapitel 5 | Logische - Operatoren

and		or		not
-----	--	----	--	-----

Jeden Tag treffen wir unbewusst eine Vielzahl von Entscheidungen. Wenn es z. B. regnet, werden wir unsere Kleidung anpassen. Für diese Art von Entscheidungen benötigen wir die logischen Operatoren. Logische Operatoren liefern einen **Wahrheitswert** als Ergebnis. Entweder ist die Prüfung wahr oder falsch (in englisch **True** oder **False**).

**and**: Dieser Operator überprüft, ob die Variablen/Methoden links und rechts von der Bedingung wahr sind. Falls beide wahr sind, wird der gesamte Ausdruck wahr. Ansonsten ist der Ausdruck falsch.

Beispiel: a = True

b = False

c = (a **and** b) ““““ Das Ergebnis hier wäre False, da die Variable b False ist. Damit kann der Gesamtausdruck nur False sein!. ““““

**or**: Funktioniert analog! Allerdings genügt es hier, dass **einer** der beiden Variablen/Funktionen True ist.

## Kapitel 5 | Logische - Operatoren

Der **not** – Operator negiert die Aussage. Aus True wird False und aus False wird True.

Beispiel:

```
a = True  
a = not a  
print(a) # Liefert false als Ergebnis!
```

Frage: Welches Ergebnis würde erscheinen, wenn wir `a = not (not a)` eintippen würden?

# Kapitel 6 | Die Funktion print()

Eine Funktion besteht aus Code, der über ihren Namen aufrufbar ist. Python bietet eine Vielzahl vorgefertigter Funktionen, die wir verwenden können. Wir müssen nicht wissen, wie die Funktion intern funktioniert. Wir benötigen lediglich den Namen und den validen Parameterbereich (Argumente), um mit der Funktion arbeiten zu können.

Die bekannteste Pythonfunktion ist print(). Diese Funktion besitzt einige spezielle Eigenschaften. Zum Beispiel können verschiedene Datentypen übergeben werden. In anderen Programmiersprachen wird dieses Konzept als „überladen“ bezeichnet.

Überladen bedeutet, dass diese Funktion mehrfach existiert und sich in ihrer Argumentenliste unterscheidet.  
*Hinweis: In Python existiert dieses Konzept nicht als solches, sondern wird intern anders implementiert*

Beispiel:

```
print("Hallo") # Eingabeparameter ist ein String!  
print(5) # Eingabeparameter ist ein Integer
```

So lange wir wissen, welche Eingabe zu welcher Ausgabe führt, ist die interne Implementierung für uns nicht relevant. Folgendes Schaubild verdeutlicht die Situation:



# Kapitel 6 | Die Funktion print()

Innerhalb des Python-Unterrichts wirst du nur eine kleine Anzahl von (vorgefertigten) Funktionen kennenlernen. Wenn du an weiteren interessiert bist, kannst du diese in der pythondoc nachschlagen.

Vielleicht fragst du dich, ob du auch selbst Funktionen schreiben kannst. Der Vorteil liegt schnell auf der Hand. Mit eigenen Funktionen kannst du einen großen Teil des Codes auslagern und in deinem „Hauptprogramm“ diese nur noch über den Namen aufrufen.

In Python ist das problemlos möglich. Die Syntax hierfür lautet:

```
def funktionsname(Parameterliste):  
    Anweisungen...
```

Nehmen wir im folgenden Beispiel an, dass wir den BMI berechnen wollen. Der BMI liefert einen Wert, der aussagen soll, ob wir zu schwer/leicht sind oder nicht.

*Hinweis: Der BMI gilt als veraltet, da er z. B. die Muskelmasse nicht berücksichtigt! Dennoch wird er als Referenz gerne angeführt.*

# Kapitel 6 | Die Funktion print()

Die Formel für den BMI lautet:  $\frac{\text{Körpergewicht}}{\text{Größe in m}^2}$

Die Funktion hierfür würde wie folgt aussehen:

```
def bmi(a,b):  
    return a/(b*b) # a = Körpergewicht, b = Körpergröße in Meter
```

a und b sind die Eingabeparameter. Die Formel benötigt das Körpergewicht (a) und die Größe (b). Danach erfolgt die Berechnung. Da wir etwas zurückgeben möchten (das ist bei Funktionen optional!), müssen wir das Schlüsselwort **return** verwenden. Das Ergebnis (Float) wird in diesem Fall zurückgeliefert.

Wenn wir die Funktion einmal definiert haben, können wir sie zukünftig einfach einsetzen. Das fertige Programm könnte wie folgt aussehen:

# Kapitel 6 | Die Funktion print()

```
def bmi(a,b):  
    return a/(b*b) # a = Körpergewicht, b = Körpergröße in Meter  
  
print(bmi(80,180)) # Das Ergebnis wäre 24,69
```

In diesem Beispiel sieht man zudem, dass Funktionen ineinander geschrieben werden können. Das spart uns einige Zeilen Code. Natürlich könnte man `bmi(80,180)` auch in einer eigenen Zeile aufrufen und das Ergebnis in eine Variable speichern. Das ganze würde dadurch allerdings umständlicher. Hier der Vergleich:

```
def bmi(gewicht,groesse):  
    return gewicht/(groesse*groesse)  
  
a = 80  
b = 180  
ergebnis = bmi(a,b)  
print(ergebnis) # Das Ergebnis wäre 24,69
```

An dem Beispiel wird deutlich, dass Variablennamen aussagekräftig sein sollten. Dann kannst du dir in diesem Fall die Kommentare sparen. Anstatt `a` und `b` wäre `koerpergewicht` und `groesse` aussagekräftiger und besser!!!

# Kapitel 6 | Die Funktion print()

Würde man alle Funktionen in das Hauptprogramm schreiben, würde die Datei sehr schnell unübersichtlich werden. Daher ist es in Python möglich, diese Programmteile auszulagern (in eigene .py Dateien). Du könntest dir z. B. eine .py Datei mit dem Namen Gesundheitsberechnungen.py anlegen und die Funktion(en) dort hineinspeichern.

*Möchte man diese Datei in sein Hauptprogramm verwenden, muss man die Datei mit dem Befehl **import** Gesundheitsberechnungen „hineinladen“. Dadurch wird die Datei (Modul) in den Namensraum aufgenommen.*

Damit Python unterscheiden kann, welche Funktion (aus welcher .py Datei) wir schließlich verwenden möchten, müssen wir für unsere Funktion den Dateinamen voranstellen. Hier ein Beispiel:

```
import Gesundheitsberechnungen  
  
print(Gesundheitsberechnungen.bmi(80,180))
```

Das würde funktionieren, sieht aber augenscheinlich etwas umständlich aus. Daher können wir das importierte Modul für den Aufruf der Funktionen umbenennen. Hierfür verwenden wir das Schlüsselwort **as**. Das Ergebnis wäre dann:

```
import Gesundheitsberechnungen as Gesundheit  
  
print(Gesundheit.bmi(80,180))
```

Es gibt eine Besonderheit zu beachten. Ein Modul wird bei dem Import ausgeführt!!! Das ist in unserem Fall oft unerwünscht. Python bietet diese Möglichkeit, da es Situationen geben kann, bei denen man einzelne Module als „standalone“-Skript ausführen möchte. In unserem Fall ist das i.d.R. nicht so. Möchten wir verhindern, dass der Code des Moduls beim Import direkt ausgeführt wird, müssen wir in das betreffende Modul, Anweisungen außerhalb von Klassen/Funktionen in folgende Anweisung packen:

```
If __name__ == "__main__":  
    anweisungen....
```

Du musst zum gegenwärtigen Zeitpunkt noch nicht verstehen, was eine Klasse ist oder wofür `__main__`, `if` usw. steht. Wichtig ist zu diesem Zeitpunkt lediglich, dass Anweisungen innerhalb des IF-Statements gepackt werden, wenn diese beim Import nicht ausgeführt werden sollen.

Du kannst dieses Verhalten direkt selbst nachvollziehen, in dem du im Modul „Gesundheitsberechnungen.py“ die Zeile `print(„TEST“)` schreibst und danach das Modul in deinem Hauptprogramm importierst. Nach dem Start wird „TEST“ auf der Konsole ausgegeben, obwohl du lediglich das Modul importiert hast.

Genau das wollen wir nicht! Daher die obige Anweisung ☺.



Es existieren Funktionen, die man mit einer unterschiedlichen Anzahl von Parametern aufrufen kann. Generell nennt man dieses Konzept/Eigenschaft „**Overloading**“. Also das Überladen einer Funktion. Hierfür wird die Funktion mehrfach implementiert, mit unterschiedlicher Signatur. Allerdings existiert dieses Konzept in Python nicht. Schreibt man mehrere Funktionen mit gleichem Namen untereinander, wird lediglich die letzte Funktion als gültig wahrgenommen. Möchte man dennoch “Overloading” implementieren, kann man dies auf unterschiedliche Art und Weise bewerkstelligen.

Entweder man schreibt eine Funktion mit der maximalen Anzahl an Parametern die man möchte und unterscheidet dann im Funktionsrumpf, ob diese Variablen überhaupt gesetzt wurden, oder man arbeitet mit `*args`, bzw `**kwargs`. Das sprengt allerdings diesen Einsteigerkurs und kann bei Bedarf nachgelesen werden.

**Aufgabe K6 | A1:** Erstelle eine Funktion `quadrat(a)`. Diese Funktion nimmt einen Parameter und liefert als Ergebnis das quadrierte Ergebnis. Teste diese Funktion nach der Erstellung mit Beispielwerten!

**Aufgabe K6 | A2:** Erstelle eine Funktion `spritkosten(a,b,c)`. Die Funktion benötigt die gefahrenen Kilometer (a), den Spritverbrauch (b) und die Kosten pro Liter (c). Für die Kosten pro Liter kannst du das Internet aufsuchen. Als Ergebnis liefert die Funktion die Kosten für die gefahrene Strecke. Lagere die Funktion in eine eigene .py Datei aus und importiere sie anschließend in dein Hauptprogramm. Der Rechner kann entsprechend der Bedürfnisse beliebig erweitert werden.

**Aufgabe K6 | A3:** Viele Spritkostenrechner aus dem Internet funktionieren nach dem obigen Prinzip. Welche Variablen sind hier nicht genannt, die allerdings einen Einfluss auf die Kosten haben werden? Notiere diese Punkte als Kommentar in deine .py Datei.

**Aufgabe K6 | A4:** Erläutere in der .py Datei, wofür wir die Anweisung `if __name__ == "__main__":` benötigen.

# Kapitel 7 | Die Funktion input()

Bisher können wir Inhalte auf der Konsole ausgeben. Allerdings können wir noch nicht mit dem Benutzer interagieren. Ohne Interaktion sind unsere Programme ziemlich langweilig. Daher benötigen wir eine Möglichkeit, Daten in das Programm nach dem Start eingeben zu können.

Hierfür verwenden wir die `input()` Funktion!

Der Aufruf der Funktion sorgt dafür, dass der Benutzer eine Eingabe via Konsole tätigen kann. Damit diese Eingabe nicht verloren geht, sollten wir das Ergebnis in eine Variable speichern. Beispiel:

```
print("Bitte gebe deinen Namen ein: ")  
name = input()
```

Die Inputfunktion ermöglicht es uns zudem, eine Ausgabe über den Eingabeparameter zu erzeugen. Damit wird unser Beispiel etwas einfacher:

```
name = input("Bitte gebe deinen Namen ein: ")
```

# Kapitel 7 | Die Funktion input()

Irgendwann wird sich die Frage stellen, was passiert, wenn man innerhalb einer Funktion einen Wert ändert. Wird dieser dann auch außerhalb der Funktion aktualisiert? Hier spielen verschiedene Konzepte eine Rolle, die sich je nach Programmiersprache unterscheiden können!

Die zwei bekanntesten Konzepte sind **Call-By-Value** und **Call-By-Reference**

Schauen wir uns ein Beispiel zu Call-By-Value an:

```
def test(b):  
    b = "Coden ist doof"  
    print("In der Funktion",b)  
  
a = "Coden ist cool"  
test(a)  
print("Außerhalb der Funktion",a)
```

Die Ausgabe wäre in diesem Fall:  
In der Funktion Coden ist doof  
Außerhalb der Funktion Coden ist cool

# Kapitel 7 | Die Funktion input()

Im Fall **Call-by-Reference** wirkt sich die Änderung auch außerhalb der Funktion aus.  
Schauen wir uns ein Beispiel zu Call-by-Reference an:

```
def more(list):  
    list.append(30)  
    print("In der Funktion", list)  
  
mylist = [10, 20]  
more(mylist)  
print("Außerhalb der Funktion", mylist)
```

Die Ausgabe wäre in diesem Fall:

In der Funktion [10,20,30]

Außerhalb der Funktion [10,20,30]

In der Praxis müssen wir häufig Entscheidungen treffen. Nehmen wir folgendes Beispiel:  
„Sollte ich mehr als 2€ in meinem Geldbeutel haben, kaufe ich mir in der großen Pause eine Cola am Automaten.  
Ansonsten bleibe ich im Klassenraum“.

Übersetzen können wir diesen Satz in „WENN geld\_im\_geldbeutel > 2, DANN cola, ANSONSTEN klassenraum“.

Es gibt Programmiersprachen die das Schlüsselwort WENN kennen. Python hingegen verwendet hier **If** als Schlüsselwort. In Python valide Syntax würden wir den obigen Satz wie folgt übersetzen:

```
if geld_im_geldbeutel > 2:  
    print("Cola kaufen")  
else:  
    print("Im Klassenraum bleiben")
```

Notiz: if = Wenn; Ansonsten = Else.  
Wichtig sind hier die Einrückungen!!!

Ist dir aufgefallen, dass in den letzten Folien der Text der Fälle leicht variiert? „Cola“, „Cola kaufen“? Das war kein Versehen, sondern zeigt, welche Interpretationsmöglichkeiten bereits ein einziger Satz haben kann. Auch Datentypen und Lösungsmöglichkeiten können sich bereits bei einem einzigen Satz stark unterscheiden .....

Normalerweise sind eigene Interpretationen gewünscht. In der Auftragsprogrammierung hingegen absolut nicht. Hier gilt die Regel:

- Halte regelmäßig Rücksprache mit deinem Auftraggeber!

Nichts ist ärgerlicher, als monatelang etwas zu entwickeln, was am Ende verworfen werden muss. Dadurch entstehen nicht nur hohe Kosten und Frustration, sondern kann im Extremfall existenzbedrohend sein.

# Kapitel 8 | Verzweigungen

Sind mehrere Fälle denkbar, kann das Schlüsselwort **elif** verwendet werden.  
Die Syntax wäre folgende:

```
if Bedingung1:  
    Anweisung(en)  
elif Bedingung2:  
    Anweisung(en)  
else:  
    Anweisung(en)
```

Else wird ausgeführt, falls alle If-Bedingungen nicht erfüllt werden!!!

```
if Bedingung1:  
    Anweisung(en)  
if Bedingung2:  
    Anweisung(en)  
else:  
    Anweisung(en)
```

Wo liegt hier der semantische Unterschied?



## Sonderfall: Switch-Case

Je nach Anwendungsszenario kann es mehrere Fälle geben, die betrachtet werden müssen. Mit IF-Bedingungen ist es zwar möglich, mehrere Fälle zu betrachten, allerdings wird das Programm sehr unübersichtlich. Hierfür gibt es die Möglichkeit, mit dem **Switch-Case**-Statement zu arbeiten.

Hier ein Beispiel:

```
def switch_test(parameter):  
    test = {  
        1: "Mathematik",  
        2: "Deutsch",  
        3: "Informatik",  
        4: "Sport"  
    }  
    print(test.get(parameter, "invalid"))
```

"""

Hier wird ein Dictionary verwendet, um die einzelnen Optionen prüfen zu können, da Python das Konzept Switch-Case als solches nicht bietet!

"""

**Aufgabe K8 | A1:** Erstelle ein Programm welches zwei Integer nimmt und überprüft welche der beiden Zahlen größer ist. Diese soll im Terminal ausgegeben werden!

**Aufgabe K8 | A2:** Das nachfolgende Programm soll drei Zahlen beinhalten, die in aufsteigender Reihenfolge im Terminal ausgegeben werden!

**Aufgabe K8 | A3:** Du möchtest dir ein neues Auto kaufen. Als Bedingung ist dir wichtig, dass das Auto unter 20.000€ kostet und eine Klimaanlage besitzt. Schreibe ein Programm, welches diese beiden Bedingungen prüft und entweder „Auto wird gekauft“ oder „Auto wird wegen XY nicht gekauft“ ausgibt.

**Aufgabe K8 | A4:** Nach dem Auto würdest du dir gerne ein neues Smartphone kaufen. Hier sind deine Ansprüche allerdings etwas höher. Der Preis sollte unter 1000€ liegen; das Smartphone sollte mindestens 3 Rückkameras besitzen; 2 Tage ohne zu laden durchhalten und von der Marke Apple oder Samsung sein.

**Aufgabe K8 | A5:** Wenn es regnet und du mindestens 20€ übrig hast oder dein/e Freund/in zahlt, geht ihr ins Kino. Ansonsten bleibt ihr zuhause ...  
Erstelle hierzu ebenfalls ein Programm mit entsprechender Ausgabe.

**Aufgabe K8 | A6:** Erstelle einen Switch-Case Block, der als case entweder den String „Stein“, „Schere“ oder „Papier“ nimmt und entsprechend auf der Konsole ausgibt, welcher Inhalt in der Variable vorher eingetragen wurde. Als default-Fall soll „Weder noch ...“ ausgegeben werden!

Mit der IF-Anweisung können wir bereits komplexere Programme schreiben. Leider können wir Programmteile noch nicht **wiederholen**. Gerade diese Wiederholung ist allerdings äußerst wichtig. Wenn du z. B. ein Spiel programmieren möchtest in dem die Spieler **so lange** gegeneinander kämpfen, **bis** eine HP (Health Point) Anzeige auf 0 sinkt, benötigst du eine Möglichkeit, Programmteile wiederholen zu können.

In Python stehen dir hierfür verschiedene Schleifen-Typen zur Verfügung, die je nach Kontext eingesetzt werden können.

Wir unterscheiden in Python die **for und while** Schleife (engl. Loop).

**While**-Schleife: Schauen wir uns direkt ein Beispiel mithilfe der while-Schleife an:

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

Nach dem Schlüsselwort while wird die Bedingung formuliert. Der Schleifenrumpf wird solange ausgeführt, bis die Bedingung im Schleifenkopf **nicht** mehr gültig ist. In unserem Fall würden die Zahlen von 0 bis 9 ausgegeben werden!

For-Schleife. Ein Beispiel:

```
for x in range(0,10):  
    print(x)
```

Hier werden ebenfalls die Zahlen von 0 bis 9 ausgegeben. Die For-Schleife bedient sich hier der Hilfsfunktion `range()`, die bei jedem Schleifendurchlauf die Variable `x` um +1 erhöht.

Merke dir hier folgendes:

- Die Abbruchbedingung muss (i.d.R) irgendwann erreicht werden. Ansonsten hast du eine **Endlosschleife** produziert.  
Frage: Gibt es Fälle, bei denen eine Endlosschleife sinnvoll ist (das Programm also nie **terminiert**)?
- Die Schrittweite kann beliebig angepasst werden!
- Möchtest du eine Schleife abbrechen, bevor die Abbruchbedingung erreicht ist, kannst du hierfür **break** verwenden.
- Um lediglich die aktuelle Iteration zu beenden, verwende **continue**.

**Kontrollstrukturen** (Schleifen & IF-Anweisung) gehören zu den wichtigsten Basics einer Sprache. Es ist sehr wichtig, dass du diese beherrscht!

Gerade die Schleifen sind häufig am Anfang verwirrend und schwierig. Mit einiger Übung wirst du feststellen, dass diese nicht so schwierig sind wie sie vielleicht scheinen.

Die kommenden Übungsaufgaben variieren im Schwierigkeitsgrad. Versuche zuerst alleine die Aufgabe zu lösen bevor du deinen Kollegen um Hilfe fragst oder die Lösung nachschlägst! Ohne die Kontrollstrukturen wirst du keine komplexeren Programme schreiben können.

Tipp: Alleine eine Stunde an einem Programmierprojekt zu verbringen ist immer sinnvoller als die Lösung nachzuschlagen.

**Goldene Regel 1**: Du lernst ja auch keine Gitarre zu spielen, indem du andere beim spielen zuschaust!

**Goldene Regel 2**: In der Programmierung schlägt Fleiß immer das Talent!



Einzelne Programmsegmente kannst du mit Schleifen wiederholen. Rekursion ermöglicht es, durch mehrmaligen Aufruf derselben Funktion den gleichen Effekt zu erzielen. Hier ein Beispiel:

```
# Die Fakultät soll berechnet werden. Bei der Fakultät
# multipliziert man die Zahlen von n, n-1 bis zur 1. Also
# fak(3) = 3*2*1 = 6
# fak(4) = 4*3*2*1 = 24

fak(n):
    if n == 1:
        return 1
    else:
        return n * fak(n-1)
```

Dieses Beispiel wäre auch mit einer Schleife relativ simpel zu lösen. Die Verwendung einer Rekursion ist vom Gedankengang allerdings naheliegender.

Hier wird mit dem zweiten Return die Funktion wieder aufgerufen, allerdings mit n-1. Durch das erste Return ist zudem sichergestellt, dass die Rekursion irgendwann endet (**Rekursionsanker**), ansonsten würde diese sich so oft aufrufen, bis es zu einer Fehlermeldung kommt bzw. (je nach Sprache und Kontext), das System einfriert!

# Kapitel 9 | Rekursion



[3]: <https://i.redd.it/0wap3cp4kkm01.jpg>

Rekursion vereinfacht einige Aufgaben enorm. Daher solltest du in der Lage sein, Aufgaben rekursiv lösen zu können.

Folgender Klassiker zeigt bereits eindrucksvoll, wie einfach bzw. kompliziert eine Aufgabe sein kann, je nachdem ob man diese rekursiv oder iterativ löst.

Fibonacci-Folge:

$F(n) = F(n-1) + F(n-2)$ , für die gilt, dass  $F(0) = 0$  und  $F(1) = 1$ .

**Aufgabe K9 | A1:** Implementiere diese Aufgabe rekursiv!

**Aufgabe K9 | A2:** Implementiere diese Aufgabe iterativ (also mithilfe von Schleifen!).

Welchen Ansatz empfindest du als einfacher?

**Aufgabe K9 | A3:** Schreibe eine Funktion „countdown“, die von 10 bis zur 0 herunterzählt und die einzelnen Zahlen ausgibt.

**Aufgabe K9 | A4:** Schreibe eine Funktion „ende“, die solange Zahlen vom Benutzer einliest, bis dieser eine 0 eingibt. Dann soll das Programm mit einer entsprechenden Nachricht beendet werden!

**Kapitel K9 | A5:** Schreibe eine Funktion „countingStars“, welches eine Zahl nimmt und Sterne nach folgendem Muster ausgibt.

Zahl = 2 entspricht:

\*

\*\*

Zahl = 3 entspricht:

\*

\*\*

\*\*\*

usw.

**Aufgabe K9 | A6:** Erweitere deine Funktion „countingStars“ dahingehend, dass die Anzahl der Sterne wieder abnimmt. Beispiel Zahl = 3;

\*

\*\*

\*\*\*

\*\*

\*

**Aufgabe K9 | A7:** Erstelle ein Mini-Spiel mit dem Namen „CrackingNumber“. Das Programm erzeugt eine zufällige Zahl zwischen 0 und 100 (Google: „Python Zufallszahl (Random)“ für Hilfe) und gibt dem Benutzer 5 Eingabeversuche. Bei jeder Eingabe erhält der Benutzer den Hinweis, ob die erzeugte Zahl größer oder kleiner ist. Sollte der Benutzer innerhalb von 5 Eingaben die richtige Zahl erraten erscheint eine Sieger-Nachricht. Ansonsten wird das Programm mit einer entsprechenden Nachricht terminiert!

**Aufgabe K9 | A8:** Schreibe eine Funktion „cancel“, die solange Eingaben vom Benutzer annimmt, bis dieser das Wort „Beenden“ eintippt. Das Programm wird dann entsprechend beendet!

**Aufgabe K9 | A9:** Erweitere das Spiel „crackingNumber“. Diesmal wählst du eine Zufallszahl aus und der Computer muss raten. Überlege dir vorher, mit welcher Strategie der Computer vorgehen soll, um mit möglichst wenigen Versuchen sein Ziel zu erreichen.

**Aufgabe K9 | A10:** Erstelle ein Programm „QuestionAnswer.py“. Dieses Programm stellt dem Benutzer mindestens 3 Fragen. Für jede richtige Antwort soll der Spieler einen Punkt bekommen. Am Ende erscheint die Gesamtpunktzahl.

**Aufgabe K9 | A11:** Programmiere eine Kinositzverteilung. Der Kinosaal besteht aus 6 Reihen mit je 9 Sitzplätzen. Die Loge besitzt 2 Reihen mit je 9 Plätzen. Dein Programm soll zufallsgeneriert einige Sitze als belegt hinterlegen und den gesamten Sitzplan via Konsole optisch ansprechend ausdrucken.

**Aufgabe K9 | A12:** In dieser Aufgabe sollst du ein Kampf zwischen verschiedenen Charakteren simulieren, basierend auf dem D&D-Regelwerk (sehr vereinfacht). Am Anfang können 2 Spieler jeweils einen Charakter auswählen (Krieger, Magier oder Schurke). Danach kämpfen diese abwechselnd solange gegeneinander, bis 1 Spieler keine Lebenspunkte mehr besitzt. Danach wird eine Nachricht angezeigt welcher Spieler gewonnen hat. Es fängt der Spieler an, der die höhere Initiative „würfelt“. Die Klassen besitzen folgende Werte bzw. Fähigkeiten (funktionen!):

Hinweis: 1d8 bedeutet, dass es sich hier um einen achtseitigen Würfel handelt. Du kannst Würfel via Randomfunktion abbilden (google: python random int)

Krieger:

Initiative: 1d8

Lebenspunkte: 1d10

Fähigkeiten:

Schwertschlag 1d7 Schaden

Schildblock: Reduziert nachfolgenden feindlichen Angriff um 1d4

Healthpotion: Heilt 1d6 (kann 1x verwendet werden)

# Kapitel 9 | Aufgaben

- Magier:
- Initiative: 1d6
  - Lebenspunkte: 1d6
  - Fähigkeiten:
  - Feuerball: 2d7 Schaden (benötigt eine Runde zum aufladen, die zweite zum abfeuern)
  - Magic\_Missile: 1d6 Schaden
  - Spiegelbilder: Mit einer Wahrscheinlichkeit von 50% wird kein Schaden am Magier für 2 Angriffe verursacht.
  - Kleine\_Heilung: Heilt 1d4 (kann 1x verwendet werden)
- Schurke:
- Initiative: 1d10
  - Lebenspunkte: 1d8
  - Fähigkeiten:
  - Sneak-Attack: Falls der Schurke zuerst angreift. Dann fügt der nächste Angriff zusätzlich 1d3 Schaden zu.
  - Dolchangriff: 2x 1d4 Schaden
  - Schmutz: Der Schurke wirft dem Gegner Dreck in die Augen. Trifft er, verursacht der nächste Angriff keinen Schaden.
  - Healthpotion: Heilt 1d6 (kann 1x verwendet werden)



Hinweise zur Lösung von K8 | A12:

Versuche das Programm so modular wie möglich aufzubauen und Fähigkeiten in Funktionen auszulagern. Du kannst zuerst die einfacheren Funktionen implementieren und kümmerge dich später um die schwierigeren (wie Feuerball oder Spiegelbilder).

Das Beispiel ist variabel ausbaubar. Du könntest den Charakteren mehr Fähigkeiten oder Eigenschaften geben. Du könntest die Rüstungsklasse einbauen (siehe D&D Armor Class).

Wenn du später gelernt hast wie man Dateien schreibt/liest, könntest du auch einen Highscore implementieren und diesen in eine Datei speichern und zu Programmstart laden.

Oder du vergibst Erfahrungspunkte für siegreiche Kämpfe und Levelaufstiege. Mit jedem Levelaufstieg sind zusätzliche Trefferpunkte und vllt. sogar neue Fähigkeiten verbunden.

Das Spiel lässt sich hier beliebig erweitern. Lass deiner Fantasie freien Lauf!

Eine **Liste** kann eine beliebig große Anzahl von Werten aufnehmen. Hierbei ist es (in Python!!!) möglich, Datentypen zu mischen.

Durch die Liste können wir einige Probleme angehen, die wir vorher lediglich umständlich lösen konnten. Die Erstellung ist denkbar einfach:

```
meineListe = [1,2,3] # Erzeugt eine Liste mit den Integerwerten 1, 2 und 3
```

Zugriff auf die einzelnen Werte einer Liste erhalten wir durch den jeweiligen **Index**. Entgegen der allgemeinen Intuitionen zählen wir ab 0 und nicht ab 1!!!

Möchten wir z. B. die Zahl 2 aus der obigen Liste anzeigen, können wir diese mit...

```
print(meineListe[1]) # In diesem Beispiel ist Index 0 = 1, Index 1 = 2, Index 2 = 3
```

...ausgeben lassen.

Hier ein paar Beispiele zur Erstellung von Listen:

```
meineListe = [] # Erstellt eine leere Liste
```

```
meineIntListe = [1,2,3] # Erstellt eine Liste mit Integerwerten
```

```
meineStringListe = [Hallo,Schule,doof] # Erstellt eine Liste mit Strings
```

```
allesAufEinmal = [2,test,[1,2,3],4,bla] # Verschiedene Datentypen in einer Liste
```

Listen bieten zudem den Vorteil, dass sie einige nützliche Methoden beinhalten, die wir wie Funktionen verwenden können. Nachfolgend sind einige Methoden genannt, die dir weiterhelfen könnten:

```
meineListe.append(x) # fügt den Wert x der Liste hinzu ( an das Ende!)
```

```
meineListe.pop(x) # liefert das x.te Element aus der Liste und entfernt dieses aus der Liste. Bei Aufruf ohne  
x wird das letzte Element aus der Liste entnommen!
```

```
y = meineListe.copy() # Erstellt eine Kopie der Liste. Wichtig ist zu verstehen, dass y = meineListe KEINE  
Kopie der Liste erzeugt!
```

```
meineListe.count(x) # zählt wie oft x in der Liste vorkommt.
```

```
meineListe.reverse() # Dreht die Liste um.
```

```
meineListe.sort() # sortiert die Liste.
```

```
len(meineListe) # liefert die Länge der Liste.
```

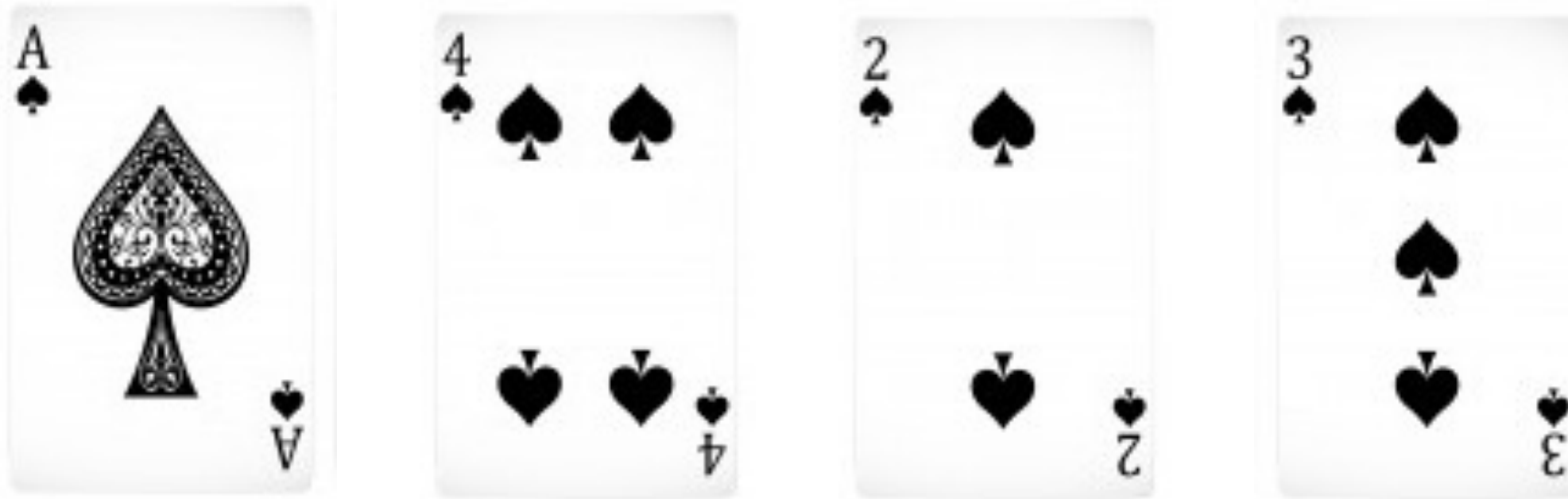
Der Umgang mit Indizes wird in vielen Bereichen der Programmierung noch wichtig werden. Daher üben wir die Verwendung anhand erster, ‚komplexerer‘ Algorithmen.

Bevor das Kapitel gelesen wird gelten folgende Regeln:

1. Versuche unbedingt die Übungsaufgaben **selbstständig** zu lösen **ohne** die Lösung im Internet nachzuschlagen (das ist sehr wichtig für den Lernerfolg!)
2. Die Verwendung der Methode `.sort` & `.reverse` ist verboten, um die Aufgaben zu lösen.

# Kapitel 10 | Sortialgorithmen

Nachfolgend siehst du vier Spielkarten. Wir wollen diese (aufsteigend) sortieren. Überlege dir zuerst wie du diese Spielkarten sortieren würdest, wenn man sie dir auf die Hand gibt.



**Arbeitsauftrag:** Notiere dir schriftlich, wie du (ganz kleinschrittig vorgehen würdest) um die Karten zu sortieren.

Um deine Lösung in Python umzusetzen kannst du einen der beiden (folgenden) Varianten verwenden. Entweder du arbeitest mit einer Liste (unsereListe = [1,4,2,3]) und versucht die Werte **innerhalb** dieser Liste zu sortieren

ODER

Du erstellst eine zweite, leere Liste und sortierst (je nach Strategie) die Karten in die Zweite Liste ein.

Für welche Variante du dich entscheidest bleibt dir überlassen. Variante 2 ist am Anfang häufig einfacher!

# Kapitel 10 | Aufgaben

**Aufgabe K10 | A1:** Erstelle eine Pythondatei mit dem Namen `sortieren.py`, indem du die schriftlichen Gedanken aus der Aufgabe vorher als Kommentar übernimmst.

**Aufgabe K10 | A2:** Schreibe eine Funktion `meinSort(meineListe)`, die deine unsortierte Liste nimmt, diese nach deiner Strategie sortiert und ausgibt.



Es gibt eine Vielzahl von Sortialgorithmen, die sich in ihrer Effizienz und Einsatzbereich unterscheiden. Schüler wählen instinktiv (in der Regel) entweder den **Selectionsort** oder den **Bubblesort** Algorithmus, um ihre Karten (Zahlen) zu sortieren.

Bei dem **Selectionsort** sucht man zuerst den kleinsten Wert innerhalb der Liste. Danach nimmt man diesen Wert aus der Liste heraus und speichert ihn in die neue Liste (funktioniert **in place** ebenfalls).

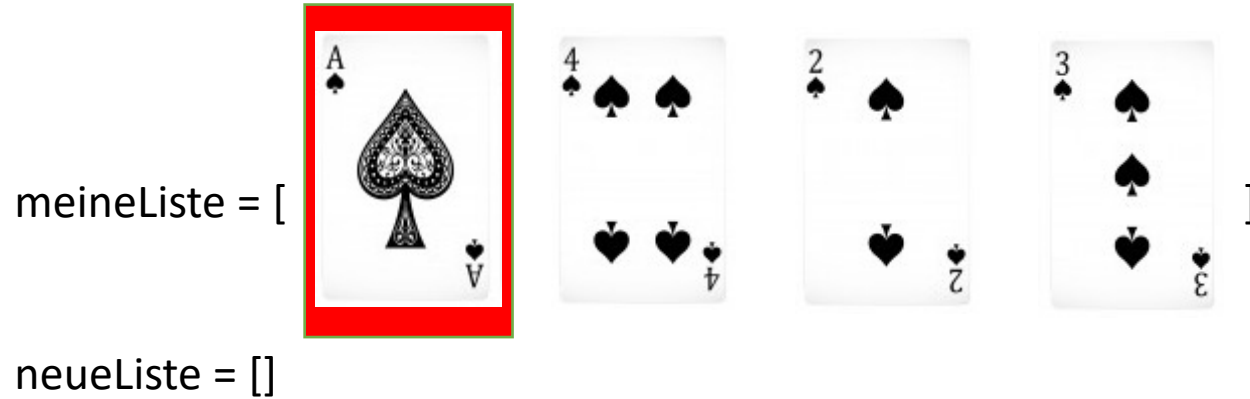
**Bubblesort** hingegen vergleicht einen Index  $x$  mit seinem Nachbarn  $x+1$ . Falls  $x$  größer ist als  $x+1$ , vertauscht er beide Werte und vergleicht danach  $x+1$  mit  $x+2$ . Dadurch wird bei einem Durchlauf die größte Zahl am Ende der Liste stehen. Führt man den Algorithmus  $n-1$  mal aus, ist die gesamte Liste aufsteigend sortiert.

*Hinweis: Der Tausch zweier Werte ist in Python extrem einfach zu realisieren. Hierfür muss nicht eine eigene `swap()` Funktion gebaut werden! Es reicht folgender Befehl:*

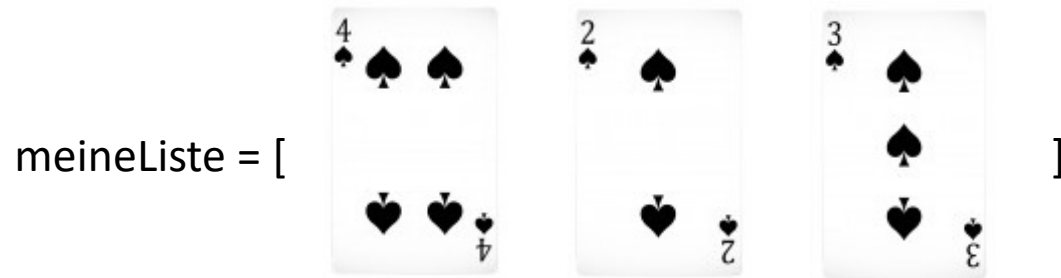
*`a, b = b, a`*

Beispielhaft der Algorithmus Selectionsort visualisiert:

1. Schritt: Kleinstes Element aus der Liste suchen.



2. Schritt: Element aus der Liste entfernen und der neuen Liste hinzufügen



3. Schritt: wiederhole Schritt 1 & 2 solange, bis die Ursprungsliste leer ist.

# Kapitel 10 | Aufgaben

**Aufgabe K10 | A3:** Implementieren Sie den Selectionsortalgorithmus als Funktion. Dieser soll out-of-place (also mit 2 Listen) arbeiten

**Aufgabe K10 | A4:** Implementieren Sie eine zweite Selectionsort-Funktion. Diesmal soll der Algorithmus in-place arbeiten.

**Aufgabe K10 | A5:** Implementieren Sie den Bubblesortalgorithmus (in-place)

Eine weitere Möglichkeit, Werte zu sortieren besteht über die Verwendung des Teile-und-Herrsche Prinzips. **Quicksort** ist der bekannteste Vertreter dieser Gruppe. Im Vergleich zu Selection- und Bubblesort ist dieser wesentlich schneller!!!

**Aufgabe K10 | A6:** Lese dir im Internet einen Artikel zum Thema Quicksort deiner Wahl durch. Visualisiere die Vorgehensweise mithilfe einer Grafik (z. B. in deinem Block oder Programm nach Wahl)

**Aufgabe K10 | A7:** Versuche den Quicksortalgorithmus in Python umzusetzen. Schreibe hierzu eine entsprechende Funktion, welche eine Liste nimmt und diese mit Quicksort sortiert.

**Aufgabe K10 | A8:** Folgende Liste ist gegeben passwords = ["ababa", „baab“, „sdiopc“, „ajkwei“, „bab“] Erstelle eine Funktion, die die Liste nimmt und überprüft ob „ab“ oder „ba“ in den jeweiligen Strings vorkommt. Wenn ja, soll das Passwort komplett ausgedruckt werden.

**Aufgabe K10 | A9:** Gegeben ist folgende Funktion:  
def blablub(lis,obj):

Vervollständige die Funktion so, dass diese überprüft ob das Objekt bereits Teil der Liste ist. Falls nicht, wird das Objekt an die Liste angehängen und die Liste zurückgegeben.

**Aufgabe K10 | A10:**

Schreibe eine Funktion „print\_it(number)“, die eine Zahl nimmt und jede Zahl von 0 bis zu dieser Zahl in 0.1 Schritten ausgibt.

**Aufgabe K10 | A11:**

Schreibe eine Funktion, die eine Liste als Parameter nimmt. Entweder wird das letzte Element der Liste zurückgeliefert oder (falls die Liste leer sein sollte), wird der String „Leere Liste“ zurückgeliefert

Neben Listen unterstützt Python das **Dictionary**.

Ein Dictionary beinhaltet sogenannte *Schlüssel-Werte-Paare* (eng: Key/Value). Einem Schlüssel ist ein Wert zugeordnet. Hier ein Beispiel:

```
Noten_BSFI = {"Hugo": 3, "Murad": 4, "Julia": 2}
```

Wir können mit `Noten_BSFI["Hugo"]` auf die Note von Hugo zugreifen. Möchten wir das gesamte Dictionary ausgeben, können wir den Namen ohne Index aufrufen. Möchten wir eine Noten abändern, können wir das beispielsweise mit...

```
Noten_BSFI["Hugo"] = 5
```

...bewertstelligen.

Auf Dictionaries können einige nützliche Funktionen angewendet werden. Z. B.

`len(d)` # gibt die Länge des Dictionaries zurück.

`del d[k]` # löscht den Schlüssel k zusammen mit seinem Wert.

`k in d` # liefert True, wenn es im Dictionary d einen Schlüssel k gibt.



Ein Dictionary ist im Gegensatz zu einer Liste kein sequenzieller Datentyp und besitzt demnach keinen fortlaufenden Index. Trotzdem können wir mit der Methode `pop()` arbeiten. Hier ist allerdings die Angabe von einem `key` wichtig. Dieser Aufruf liefert dann den dazugehörigen Wert (Value) und löscht danach diesen Wert inkl. Key aus dem Dictionary.

**Aufgabe K11 | A1:** Folgendes Dictionary ist gegeben:

```
tel = {  
    'arbeit': {'Jan': '50', 'Mark': '81'},  
    'private': {'Julia': '0151438924', 'Mark': '016932834'}  
}
```

Welche Ausgaben liefern folgende Befehle:

1. `len(tel)`
2. `len(tel['arbeit'])`
3. `tel['arbeit'].get('Mark')`
4. `'Jan' in tel['arbeit']`
5. `list(tel['private'].values())`
6. `tel['private'].copy()`

**Aufgabe K11 | A2:** Folgendes Dictionary ist gegeben:

```
tel = {  
    'arbeit': {'Jan': '50', 'Mark': '81'},  
    'private': {'Julia': '0151438924', 'Mark': '016932834'}  
}
```

Folgende Operationen sollen durchgeführt werden:

1. Ändere nachträglich die interne Durchwahl von Mark zu 80.
2. Lösche die Eintrag von Julia inkl. Handynummer
3. Ändere den Namen Mark im privaten Bereich in „Murad“

**Aufgabe K11 | A3:** Schreibe eine Funktion `supersearch(d,value)`, welches ein Dictionary und einen value (Wert) nimmt und diesen Wert im Dictionary sucht. Falls der Wert gefunden wird, soll die Funktion `True` zurückliefern, falls nicht `False`.

*Hinweis: build-in Suchfunktionen (z. B. `d.get()`) sind nicht erlaubt!*

**Aufgabe K11 | A4:** Schreibe ein Programm, in dem ein Benutzer beliebig lange Länder inkl. Hauptstädte eingeben kann, bis er Beenden eintippt.

Die Länder + Hauptstädte werden in einem Dictionary gespeichert und danach auf der Konsole ausgegeben. Das Ausgabeformat lautet:

“	Land		Hauptstadt	“
“1.	Deutschland		Berlin	“
“2.	Frankreich		Paris	“

**Aufgabe K11 | A5:** Schreibe eine Funktion `foo(number)`, die eine ganze Zahl nimmt und überprüft ob diese positiv, negativ, gerade, ungerade oder gar keine Integer ist. Die Ausgabe soll in Form eines Dictionaries erfolgen.

Beispiel:

`foo(10) => {"vorzeichen: positiv", "parität": "gerade"}`

`foo(-5.3) => {"vorzeichen: negativ", "parität": "kein Integer"}`

Mit RegEx Ausdrücken kann vor allem die Suche nach Dateien, Ordnern oder Namen stark vereinfacht werden. Wenn du z. B. alle Personen in der Schule auflisten möchtest, die mit Sch im Nachnamen beginnen, könnte man diese mit Sch\* auflisten lassen (funktioniert in vielen Programmen). Der \* ist hier eine **Wildcard** und steht für „beliebig viele Zeichen die ab \* folgen“. Oder du möchtest alle .exe Dateien in deinem System suchen. Das könnte mit \*.exe bewerkstelligt werden. Vielleicht möchtest du auch alle Dateien auf einem Linux-Server löschen, die sich in einem speziellen Ordner befinden.

Mit dem Modul re (import re) gibt dir Python ein mächtiges Werkzeug an die Hand, solche Ausdrücke selbst zu schreiben und dir Dinge damit zu vereinfachen.

Ähnlich verhält es sich mit der List Comprehension. Damit ist eine verkürzte Schreibweise möglich. Nehmen wir folgende Liste als Beispiel:

```
S = [3, 1, 2, 9, 10, 0]
```

In dieser Liste sind gerade und ungerade Zahlen. Wenn ich alle ungeraden Zahlen aus der Liste entfernen möchte bzw. eine Liste mit nur geraden Zahlen benötige, könnte ich diese wie folgt erstellen:

```
B = [x for x in S if x % 2 == 0]
```

Hier wird x bei jedem Durchgang der for-Schleife ein Wert von S zugewiesen und danach geprüft, ob die Zahl gerade ist. Nur wenn die Zahl gerade ist, wird diese in die neue Liste übernommen, ansonsten geschieht nichts.

Ohne List Comprehension müssten wir einige Zeilen Code mehr schreiben, um die Aufgabe zu lösen!

RegEx & List-Compr. sehen auf den ersten Blick kryptisch und kompliziert aus. Mit etwas Übung sind diese allerdings gar nicht so schwierig zu verstehen!



**Aufgabe K12 | A1:** Schreibe eine Funktion `is_included()`, die einen String nimmt und überprüft, ob das Wort „funktioniert“ beinhaltet ist. Falls ja, liefert die Funktion `True` zurück. Falls nicht, `False`. Nutze für die Lösung RegEx-Ausdrücke.

**Aufgabe K12 | A2:** Schreibe eine Funktion `nur_ungerade()`, die eine Liste mit Zahlen übergeben bekommt und eine neue Liste erzeugt, in der lediglich die ungeraden Zahlen vorhanden sind (nutze hierfür List Comprehension)

In der Realität müssen wir häufig mit externen Dateien arbeiten (z. B. einem Highscore, dessen Werte in einer Excel-Datei stehen oder einen neuen Charakter, den wir in unser Rollenspiel importieren möchten, oder oder oder). Das Einlesen von Daten in Python ist unglaublich einfach (im vgl. zu anderen Programmiersprachen).

Die Syntax lautet wie folgt:

```
data = open(„dateiname.txt“, “r“) # hier steht das “r“ für lesen (read).
```

Liegt die Datei in einem anderen Pfad, kann dieser auch **absolut** angegeben werden. Das führt allerdings häufiger zu Problemen (warum?).

Nach dem Öffnen stehen verschiedene **Methoden** bereit, um mit der Datei zu interagieren. Z. B. könntest du `print(data.readline())` aufrufen, um einzelne Zeilen einzulesen.

Möchte man die gesamte Datei Zeile für Zeile einlesen, könnte man z. B. `for x in data:` verwenden. Wichtig ist vor allem, dass die Datei danach wieder geschlossen wird. Dies wird mit `data.close()` gewährleistet.

Dateien mit `data = open()` zu öffnen, führt zu einigen Nachteilen, die du zum gegenwertigen Zeitpunkt noch nicht verstehen würdest. Daher sollte besser folgende Zeile verwendet werden:

`with open("datei.txt", "r") as data:`

**Aufgabe K13 | A1:** Erstelle eine .txt Datei mit folgendem Inhalt:

Markus,Burg,19  
Julia,Scherger,18  
Murad,Surgy,18

**Aufgabe K13 | A2:** Schreibe eine Funktion `read_every_line()`, die beim Aufruf jede Zeile auf der Konsole ausgibt.

**Aufgabe K13 | A3:** Schreibe eine Funktion `read_one_value()`, welches jeden Wert (z. B.) Markus einzeln auf der Konsole ausgibt und danach einen Zeilenumbruch erzeugt. Ausgabe ist also:

Markus  
Burg  
19  
Julia  
...

**Aufgabe K13 | A4:** Schreibe eine Funktion `just_eighteen()`, welches nur Personen auf die Konsole druckt, die genau 18 Jahre alt sind.

Das Schreiben in eine Datei funktioniert ähnlich.

Die Syntax lautet wie folgt:

With `open("dateiname.txt", mode="w") as data`: # hier steht das "w" für lesen (write).

Sollte die Datei noch nicht existieren, wird diese erzeugt. Auch hier ist es wichtig, nach der Operation die Datei wieder zu schließen.

Das Lesen und Schreiben von Dateien können mithilfe des Moduls `csv` (`import csv`) vereinfacht werden. CSV steht hier für „Comma-separated values“ und ist ein weitläufiges Format. Ursprünglich wurden Werte mit Komma (daher der Name) in diesem Austauschformat getrennt. Also z. B.  
`Julia,Mueller,Burgstr,5`

Hier wird aber direkt ersichtlich, dass man ein Komma vielleicht im Wert (value) selbst benötigt. Daher kann ein beliebiges Trennzeichen verwendet werden. Trotzdem nennt man die Datei CSV!  
Z. B. `Julia#Mueller#Burgstr#5`

**Aufgabe K14 | A1:** Schreibe eine Funktion, welche die Zahlen von 1 bis 10 erzeugt und diese Zahlen mit Komma getrennt in eine neue Datei schreibt. Der Dateiname ist „my\_first\_numbers.csv“.

**Aufgabe K14 | A2:** Wie kann verhindert werden, dass ein erneuter Aufruf der Funktion dazu führt, dass der ursprüngliche Inhalt überschrieben wird? Beschreibe dein Vorgehen in einer .py Datei und implementiere die Lösung

**Aufgabe K14 | A3:** Schreibe eine Funktion „super\_fun(number)“, welche eine Zahl nimmt und die Anzahl an Dateien nach dem Schema „file1.txt“, „file2.txt“ usw. erstellt.

Überprüfe danach was passiert, wenn du als Zahl 9999 übergibst. Werden dann 9999 Dateien erstellt?

# Kapitel 15 | OOP

## Klassen, Objekte & Konstruktoren

Python unterstützt die Objektorientierung. Diese ermöglicht es, Sachverhalte mit ihren Eigenschaften aus der realen Welt zu übernehmen. Das erleichtert die Programmierung ungemein.

In einem objektorientierten Programm stehen **Klassen** und **Objekte** im Mittelpunkt. Eine Klasse ist ein Bauplan für Objekte. Der Bauplan hat den Vorteil, dass man aus diesem beliebig viele Objekte erzeugen kann. Eine Klasse erstellt man mit dem Keyword `class`. Zum Beispiel so:

```
class PKW:
```

Danach kann im Programm ein Objekt der Klasse PKW erzeugt werden. Ein Beispiel wäre folgendes:

```
mein_Auto = PKW()
```

Das wäre allerdings etwas langweilig, da unser PKW weder Attribute, noch Methoden (das sind Funktionen, die auf Objekten aufgerufen werden) besitzt. Daher müssen wir uns vorher überlegen, welche Eigenschaften so ein PKW besitzt. Ein Auto besitzt eine Farbe, eine Anzahl von Türen, PS, einen Preis usw.



# Kapitel 15 | OOP

## Klassen, Objekte & Konstruktoren

Diese Variablen sollten wir bereits erzeugen und mit einem Wert versehen (initialisieren), sobald ein Objekt erzeugt wird. Hierfür benötigen wir einen Konstruktor, der für uns dieses Objekt baut. Wenn wir keinen expliziten Konstruktor verwenden, wird ein impliziter Konstruktor aufgerufen (daher funktioniert der Code aus der vorangegangenen Folie!).

Ein expliziter Konstruktor ist i.d. R. allerdings besser. Hier ein Beispiel:

```
class PKW:
    def __init__(self, farbe, anz_tueren, ps, preis):
        self.farbe = farbe
        self.anz_tueren = anz_tueren
        self.ps = ps
        self.preis = preis
```

Der Konstruktor ist demnach spezielle Methode mit dem speziellen Namen `__init__`. Das `self` bezieht sich auf das aktuelle Objekt und muss daher in jeder Methode als erster Parameter stehen!).

# Kapitel 15 | OOP

## Klassen, Objekte & Konstruktoren

```
class PKW:
    def __init__(self, farbe, anz_tueren, ps, preis):
        self.farbe = farbe
        self.anz_tueren = anz_tueren
        self.ps = ps
        self.preis = preis
```

Achte auf die Zeile `self.farbe = farbe`. Hier sind zwei unterschiedliche Variablen benannt. `self.farbe` ist die Variable, die im Konstruktor erzeugt wird. Dieser wird der Wert von `farbe` zugewiesen, welches über die Methode hineingegeben wird. Daher müssen wir (im Gegensatz zu Java z. B.) keine Variablen vorher anlegen. Es reicht völlig aus, wenn diese im Konstruktor angelegt werden. Mit dem obigen Beispiel würde allerdings unsere Codezeile `mein_Auto = PKW()` nicht mehr funktionieren!

Da wir jetzt einen expliziten Konstruktor verwenden, müssen wir auch die entsprechenden Parameter übergeben, also z. B. `mein_Auto = PKW(„grün“,5,100,20000)`

# Kapitel 15 | OOP

## Klassen, Objekte & Konstruktoren

Methoden entsprechen den Funktionen mit der Ausnahme, dass als erster Parameter self übergeben wird. Generell sollten Klassen mit ihren Methoden in eine eigene Datei ausgelagert und via Import importiert werden. Das sorgt für einen besser strukturierten Code!

**Aufgabe K15 | A1:** Erstelle eine Klasse Haus inkl. Konstruktor, welches über folgende Attribute verfügt: Straße, Hausnummer, Farbe, Preis, Stockwerksanzahl, Quadratmeter.

**Aufgabe K15 | A2:** Erstelle ein Objekt der Klasse Haus mit beliebigen Werten. Gebe danach alle Werte auf der Konsole wieder aus.

**Aufgabe K15 | A3:** Erstelle 3 verschiedene Hausobjekte und speichere diese in einer Liste. Lösche danach das letzte Haus in der Liste und gebe die Liste entsprechend auf der Konsole aus.

**Aufgabe K15 | A4:** Füge der Klasse die Methode `neuer_anstrich(farbe)` hinzu, die eine neue Farbe nimmt und den Wert entsprechend abändert. Erstelle dann ein Objekt, dessen Farbe nachträglich geändert wird und gebe diesen Wert auf der Konsole erneut aus.

**Aufgabe K15 | A5:** Erstelle eine neue Hausliste mit 3 Hausobjekten. Danach soll überprüft werden, ob ein Haus über 200.000€ kostet. Falls ja, wird das Haus aus der Liste entfernt. Gebe anschließend die Liste auf der Konsole aus.

# Kapitel 16 | OOP

## Getter & Setter

Vielleicht ist dir schon aufgefallen, dass man mit **Objektname.Variable** auf die Variablen zugreifen und diese verändern kann. Normalerweise sollte jetzt erläutert werden, wie Sichtbarkeiten (Access Modifier) funktionieren, da diese in anderen Programmiersprachen (z. B. Java) eine wichtige Rolle spielen. In Python hingegen existieren andere Richtlinien mit Zugriffsmöglichkeiten und Sichtbarkeiten. Daher beschränken wir uns in diesem Kurs darauf, wie man Variablen ändern können sollte!

Ein direkter Zugriff auf die Variable ist immer ungünstig. Dadurch könnte der Wert in einen Bereich fallen, der nicht mehr valide ist. Z. B. `mein_Haus.Preis = -9999999`. Daher verwenden wir ein Konzept (getter & setter) um den Wert abzurufen (get) und zu verändern (set). In Python funktioniert das wie folgt:

```
@property
def preis(self):
    return self._preis

@preis.setter
def preis(self, value):
    #insert Code here
    self._preis = value
```

# Kapitel 16 | OOP

## Getter & Setter

Die Python-Variante der Getter & Setter ist leider etwas verwirrend für Anfänger. Hier ist unser Preis ein Attribut, welches über den **Decorator** `@property` markiert wird. Dadurch kann über die beiden Methoden auf den Preis zugegriffen werden, als wäre es weiterhin eine normale Variable. Das hat den großen Vorteil, dass ich im “setter” Code hinzufügen kann, in der z. B. die Eingabe auf Validität geprüft wird.

Der zweite Vorteil liegt darin, dass nach außen hin der Zugriff nicht verändert wird. Dieser ist weiterhin mit `Objektname.Variablenname` möglich. Im Vergleich würde man in Java beispielsweise zwei Methoden `get_preis()` und `set_preis()` erstellen und über diese Methoden den Preis holen und setzen.

Du merkst also, dass sich Programmiersprachen zwar in ihrer Implementierung unterscheiden können aber das Grundkonzept ist nahezu immer identisch oder sehr ähnlich. Daher gilt auch immer die Regel, dass wenn man eine Programmiersprache sehr gut beherrscht, das lernen einer neuen wesentlich schneller funktioniert.

**Aufgabe K16 | A1:** Erstellen Sie zur Hausklasse jeweils einen Getter & Setter für die Attribute Preis und Quadratmeter. Im Setter soll jeweils ausgeschlossen werden, dass negative Werte eingegeben werden können. Wird ein negativer Wert nachträglich eingegeben soll eine Warnmeldung in der Konsole erscheinen und der Wert wird nicht hinzugefügt. Falls der Wert valide ist, wird dieser hinzugefügt.

Überprüfen Sie die Änderung, indem Sie die Werte danach auf der Konsole ausgeben lassen.

# Kapitel 17 | OOP

## Vererbung

Oftmals stehen Klassen in einer Beziehung zueinander. Nehmen wir unser Rollenspiel aus Kapitel 9. Ein Spieler wählt einen Charakter. Dieser Charakter könnte ein Magier, Schurke oder Krieger sein. Alle Charaktere besitzen Lebenspunkte, einen Namen, Körpergröße usw. Allerdings unterscheiden diese „Klassen“ sich in gewissen Attributen. Ohne Vererbung müssten wir in jeder Klasse die Variable Lebenspunkte etc. festlegen. Das ist äußerst umständlich und in der Pflege hinderlich.

Das Konzept der Vererbung erleichtert uns diese Aufgabe. Wir definieren eine „Vaterklasse“, von der alle Attribute und Methoden geerbt werden. Dadurch müssen wir unseren Code nur an einer Stelle pflegen. Schauen wir uns folgendes Beispiel an:



# Kapitel 17 | OOP

## Vererbung

```
1 class Klassentemplate:
2     def __init__(self, name, hp, size):
3         self.name = name
4         self.hp = hp
5         self.size = size
6
7 class Mage(Klassentemplate):
8     def __init__(self, name, hp, size, mana):
9         super().__init__(name, hp, size)
10        self.mana = mana
11
12 Player1 = Mage("Gandalf", 100, 182, 200)
```

Zeile 7: Die Vaterklasse wird in Klammern geschrieben)

Zeile 9: mit `super().__init__` wird der Konstruktor der Vaterklasse aufgerufen.

Zeile 10: die Klassenspezifische(n) Variablen werden in der Subklasse gesetzt.

Zeile 12: Danach kann ein Objekt der Subklasse instanziiert werden.

Die Subklasse (Mage) erbt nicht nur die Attribute der Vaterklasse, sondern auch die Methoden, sofern es welche gibt. Wir könnten also in dem Klassentemplate eine Methode „zuschlagen()“ implementieren und diese dann jeweils vererben. Ändert sich irgendwann etwas an der Implementierung der Methode „zuschlagen()“, muss diese nur im Template angepasst werden, nicht aber in jeder Kindklasse.

**Aufgabe K17 | A1:** Erstelle eine Vaterklasse Fahrzeug und die Kindklassen Motorrad und Auto. Erstelle eine entsprechende Vererbungshierarchie zwischen den Klassen.

Überlege dir anschließend über welche Eigenschaften und Attribute ein Auto & Motorrad verfügen sollte. Falls diese übergreifend sind, implementiere diese entsprechend in der Vaterklasse. Erstelle ein entsprechendes Programm, welches einige Objekte anlegt und mit diesen interagiert.

# Kapitel 18 | OOP

## ABC (Abstract Base Class)

In unserem Beispiel macht es keinen Sinn ein Objekt der Klasse „Klassentemplate“ anzulegen. Allerdings möchten wir oftmals, dass Methoden in der Vaterklasse entsprechend in der Kindklasse implementiert und ausgebaut werden, da sich die Implementierung je nach Charakter unterscheiden könnte.

Python bietet die Möglichkeit, abstrakte Klassen bzw. Methoden zu definieren. Damit können wir erzwingen, dass Kindklassen die entsprechenden Methoden implementieren müssen.

```
from abc import ABC, abstractmethod
```

```
class AbstractClass(ABC):  
    def __init__(self, value):  
        self.value = value  
  
    @abstractmethod  
    def irgendwas(self):  
        pass
```

Das Vorgehen ist simple:

1. Importiere von abc die Klasse ABC & abstractmethod
2. Erbe von ABC
3. Schreibe vor der Methode, die abstrakt sein soll den **Dekorator** @abstractmethod

Ab jetzt ist es nicht mehr möglich Instanzen der Klasse AbstractClass zu erzeugen, da diese eine abstrakte Methode enthält.

# Kapitel 18 | OOP

## ABC (Abstract Base Class)

Eine Kindklasse könnte jetzt von der Vaterklasse erben und die Methode „irgendwas(self)“ **überschreiben**. Hierfür definiert sie selbst eine Methode „irgendwas(self)“. Dadurch kann demnach die Kindklasse gezwungen werden, alle abstrakte Methoden zu implementieren.

Mit dieser Technik ist es auch möglich ein **Interface** nachzubauen. In anderen Programmiersprachen (z. B. Java) unterscheiden sich abstrakte Klassen und Interfaces enorm voneinander. In Python existiert ein Interface als solches nicht, da es in vielen Fällen nicht benötigt wird.

Möchte man dennoch ein Interface erstellen, kann man ABC hierfür verwenden und ein Pseudo-Interface nachkreieren.

**Aufgabe K18 | A1:** Erstelle für unser „RPG-Fighting-Game“ eine Vaterklasse „Creature“, sowie die Kindklassen „Mage“ und „Warrior“. Die Vaterklasse beinhaltet neben den Attributen „hp,name,size,gold“, die Methoden „roll\_dice(int)“ und „fist\_hit()“. Beide Methoden sollen abstrakt sein und in den Kindklassen entsprechend implementiert werden. Bei roll\_dice(int) wird imaginär gewürfelt. Der Parameter entspricht den Seiten des Würfels. Also z. B. roll\_dice(6) -> 1 bis 6 als möglicher Output. fist\_hit() verwendet roll\_dice um den Schaden zu ermitteln. Der Magier macht 1d3 Schaden, der Krieger hingegen 1d4. Daher sollten sich die Implementierungen entsprechend unterscheiden.

Das Attribut Gold soll einen getter&setter besitzen, wie es in den Folien beschrieben wird (also mit Dekorator).

Implementiere alle Vorgaben in einem Programm und teste es anschließend.

# Kapitel 19 | OOP

## Polymorphie

Polymorphie bedeutet Vielgestaltigkeit. Schauen wir uns direkt ein Beispiel mit Ausgabe an:

```
1 class Mage(object):
2     def introduction(self):
3         print("Ich bin ein nobler Magier!")
4
5
6 class Warrior(object):
7     def introduction(self):
8         print("Nur eine Plattenrüstung ist eine Rüstung!")
9
10
11 def introduce_yourself(Creature):
12     Creature.introduction()
13
14
15 mageObj = Mage()
16 warriorObj = Warrior()
17
18 introduce_yourself(mageObj)
19 introduce_yourself(warriorObj)
```

Ausgabe: Ich bin ein nobler Magier!  
Nur eine Plattenrüstung ist eine Rüstung!

Interessant ist hier Zeile 11. Diese nimmt einen Datentyp, von dem erst zur Laufzeit klar ist, ob es sich um einen Magier oder Krieger handelt. Wir können zur Laufzeit ein beliebiges Objekt hinzufügen, welches über die entsprechende Methode verfügt.

Das heißt, Polymorphie (hier auf Funktionsebene) erlaubt es uns, weniger Code schreiben zu müssen und diesen zu generalisieren.

Bisher sind wir davon ausgegangen, dass sofern unser Code kompiliert, dieser auch fehlerfrei laufen wird. Es gibt allerdings Situationen, in denen häufiger zur Laufzeit etwas schief geht. Wenn ihr z. B. eine Datei via Python öffnen wollt, die allerdings nicht existiert, stürzt euer Programm zur Laufzeit, nicht zur Kompilzeit ab.

Oder du möchtest, dass der Benutzer eine Zahl eingibt, dieser gibt allerdings einen Buchstaben ein. Das Programm unkontrolliert abstürzen zu lassen, ist keine gute Idee. Es frustriert den Benutzer und kann je nach Kontext zu schwerwiegenden Problemen führen (Stichpunkt: offene Dateien, Datenbankverbindung usw.).

Daher möchten wir kritische Codeteile überwachen um in Fehlerfall darauf reagieren zu können. Python bietet hierfür deine klassische Fehlerbehandlung, wie sie auch in vielen anderen Programmiersprachen implementiert ist.

Die Syntax zur Überwachung eines Codeabschnitts sieht wie folgt aus:

```
try:
    f = open("mypath", "r")

except IOError:
    print("Arg, kann datei nicht öffnen")
else:
    f.close()
```

Der entsprechende Abschnitt wird in einen `try:` Block gepackt. Danach können entstehende Fehler mit `except` gefangen und behandelt werden. Sollte der Name des Fehlers nicht bekannt sein, kann auch `except` ohne Spezifizierung verwendet werden.

Mehrere `except`-Blöcke sind möglich und auch sinnvoll.

Optional kann am Ende ein `else`-Block hinzugefügt werden. Dieser wird ausgeführt, sofern keine Exception geworfen wird. Hierdurch ist es möglich, dass Programm kontrolliert herunterzufahren.



Manchmal möchte man eine eigene Error-Klasse kreieren, die im Fehlerfall Aktionen vornimmt. In Python wird eine eigene Fehlerklasse erstellt, indem man von Exception erbt. Also `class Fehlername(Exception)`.

Möchte man im Hauptprogramm die eigene Fehlerklasse ansprechen, verwendet man das Schlüsselwort **raise**.

Nachfolgende Beispiel zeigt die Vorteile einer eigenen Fehlerklasse und wann diese Sinn machen:

```
class HighscoreError(Exception):  
    pass  
  
try:  
    # If something_bad_happens:  
    raise HighscoreError()  
except HighscoreError as e:  
    print("Hier aufruf der eigenen Klassenmethoden", e.methode())
```

Im Beispiel wird ein `HighscoreError` erzeugt. Die Klasse kann beliebig komplex sein. Beim Erzeugen eines Highscores bzw. dem Abruf der Daten von einer Datei, die im Internet liegt kann so einiges schief gehen. Innerhalb der Klasse kann speziell darauf eingegangen werden. Natürlich könnte man jeden Fall auch in das „Hauptprogramm“ schreiben, was zu unleserlichem Code führen würde.

Daher sollten solche Programmsegmente sinnvollerweise in eine Klasse ausgelagert werden, die sich um das Problem kümmert, sollte es auftreten.

**Aufgabe K20 | A1:** Öffne eine .txt Datei auf deiner Festplatte mit der open() Anweisung (ohne das Schlüsselwort which).

Welche Fehlermeldungen können auftreten? Erstelle für jede Fehlervariante eine entsprechende Exception, bei der du den User mit einer print-Ausgabe auf den Fehler aufmerksam machst und dort beschreibst, was er tun soll um den Fehler zu beheben.

Eine der wichtigsten Bestandteile im Ablauf einer Softwareerstellung ist das **Testen**. Es gibt eine Vielzahl von Testtypen und Arten (Blackboxtest, Whiteboxtest, Unittest, Integrationtest, Systemtest usw. usw.).

Wir beschränken uns in diesem Kurs auf den **Unittest**. Eine Unit ist in der OOP-Welt entweder eine Funktion, Methode oder Klasse. Natürlich könnten wir unser Programm nach dem **Trial-&-Error** Verfahren testen. Das ist allerdings sehr aufwändig und bei Codeänderungen, müssten wir jede betroffene Funktion, Methode oder Klasse erneut testen.

Daher automatisieren wir diesen Schritt. Hierfür bietet Python verschiedene **Testframeworks** an. Wir verwenden das Framework **unittest**, welches bereits in Python integriert ist.

VSCode bietet eine integrierte Testumgebung, die im Skript schwierig zu beschreiben ist und ich dir live vorführen werde!

Im Skript können wir uns allerdings die Syntax ansehen.

# Kapitel 21 | Testen

Zuerst erstellt man eine neue .py Datei mit dem Namen `test_name`. Der Name entspricht der zu testenden Funktion bzw. Klasse. Danach importiert man unittest und schreibt die Testfälle. Schauen wir uns hierzu ein einfaches Beispiel an:

The image illustrates the process of creating a unit test in VS Code. It is divided into three main sections:

- File Explorer (Left):** Shows a project structure with a folder named `TESTAREA`. Inside, there are files `.venv`, `.vscode`, `somestuff.py`, and `test_superadd.py`. The `test_superadd.py` file is selected.
- Code Editor (Top Right):** Displays the source code of `somestuff.py`:

```
class SomeStuff:  
    def superadd(self, number1, number2):  
        return number1 + number2
```
- Code Editor (Bottom Right):** Displays the test code in `test_superadd.py`:

```
import unittest  
from somestuff import SomeStuff  
  
class TestSomeStuff(unittest.TestCase):  
    def test_add(self):  
        testobj = SomeStuff()  
        self.assertEqual(testobj.superadd(2, 3), 5)
```
- Test Runner (Bottom Left):** Shows the output of the test runner, indicating that the test passed:

```
✓ test_superadd.py  
  ✓ TestSomeStuff  
    ✓ test_add
```

Red arrows indicate the relationship between the files: one arrow points from `somestuff.py` in the file explorer to the source code editor, and another points from `test_superadd.py` to the test code editor.

Unsere TestKlasse erbt von unittest.TestCase. Daher müssen wir unittest vorher importieren. Danach haben wir Zugriff auf eine Vielzahl von Methoden. Darunter befindet sich die Methode assertEquals!

Diese prüft, ob die Ausgabe der Methode superadd(2,3), zum erwarteten Ergebnis 5 passt. 2+3 ist 5 und demnach läuft der Test erfolgreich durch. Würden wir die 5 abändern, würde der Test fehlschlagen, da das Ergebnis von superadd(2,3) != x wäre.

Natürlich stellt sich hier die Frage, warum wir diesen Aufwand betreiben sollen, da jeder sofort sehen kann, was das Ergebnis von a + b sein soll. Allerdings geht es hier lediglich darum, die Funktionsweise von unittest zu erläutern.

In der Praxis wirst du weit komplexere Methoden, Funktionen und Klassen schreiben. Es kann dann (je nach Kontext) Sinn machen, zuerst die Testfälle zu schreiben und danach das zu programmierende Programm. Dieses Vorgehen nennt sich **Test-Driven-Development**.

Außerdem können Tools verwendet werden, die nach jeder Codeänderung, diesen gegen die vorhandenen Tests checken. Das ist vor allem interessant, wenn man im Team programmiert.

**Aufgabe K21 | A1:** Erstelle eine Klasse mit folgenden Methoden:

`is_empty_list(list)` nimmt eine Liste und prüft ob diese leer ist. Falls ja, wird `True` zurückgegeben.

`wurzel(number)` nimmt eine Zahl und berechnet die Wurzel daraus.

Erstelle für beide Methoden jeweils geeignete Testfälle (mindestens 4 pro Methode). Jeweils zwei Testfall für valide und invalide Eingaben.

Teste danach, ob VSCode die Testfälle richtig auswertet.

Nachdem alle erforderlichen Grundlagen gelegt wurden, können wir damit beginnen, eine grafische Oberfläche (**GUI** = graphical user interface) zu konstruieren.

Python bietet eine Vielzahl von **Frameworks**, um diese Aufgabe zu erleichtern. Die bekanntesten sind **Tkinter** & **PyQT**. Tkinter wird bereits mit Python mitgeliefert und muss nicht separat installiert werden. Damit können einfache Oberflächen gebaut werden und ist vom Lernprozess für viele einfacher zu verstehen als PyQT.

Der Nachteil besteht darin, dass Oberflächen von Tkinter häufig altbacken aussehen. Eine grafische moderne Oberfläche, die via css-style formatiert wird und **Entwicklungsmuster** im Hintergrund verwendet, sprengt bei weitem das Niveau des Kurses.



Eine graphische Oberfläche besteht aus mindestens einem Fenster. Dieses Fenster soll permanent geöffnet sein, bis eine Aktion eintritt. Um ein Hauptfenster zu öffnen und dieses offen zu halten, benötigen wir lediglich wenige Zeilen Quellcode:

```
import tkinter

form = tkinter.Tk()
form.title("unser Hauptfenster")

form.mainloop()
```

mit `import tkinter` wird das Python-Framework hinzugefügt. Wichtig ist vor allem die letzte Zeile, die dafür sorgt, dass das Fenster geöffnet bleibt. Der Output sieht dann wie folgt aus:



Je nach Betriebssystem unterscheidet sich die Titelleiste! Das ist eine wichtige Erkenntnis, wenn es darum geht, plattformübergreifende Programme zu entwickeln.

Jetzt fehlen unserem Hauptfenster noch Elemente. Diese Elemente werden **Widgets** oder **Controls** genannt. Es gibt eine sehr große Anzahl an Widgets, darunter den Button, Radiobutton, Checkbox, usw.

Eventuell ist ein Widget standardmäßig nicht in Tkinter enthalten und muss nachinstalliert werden.

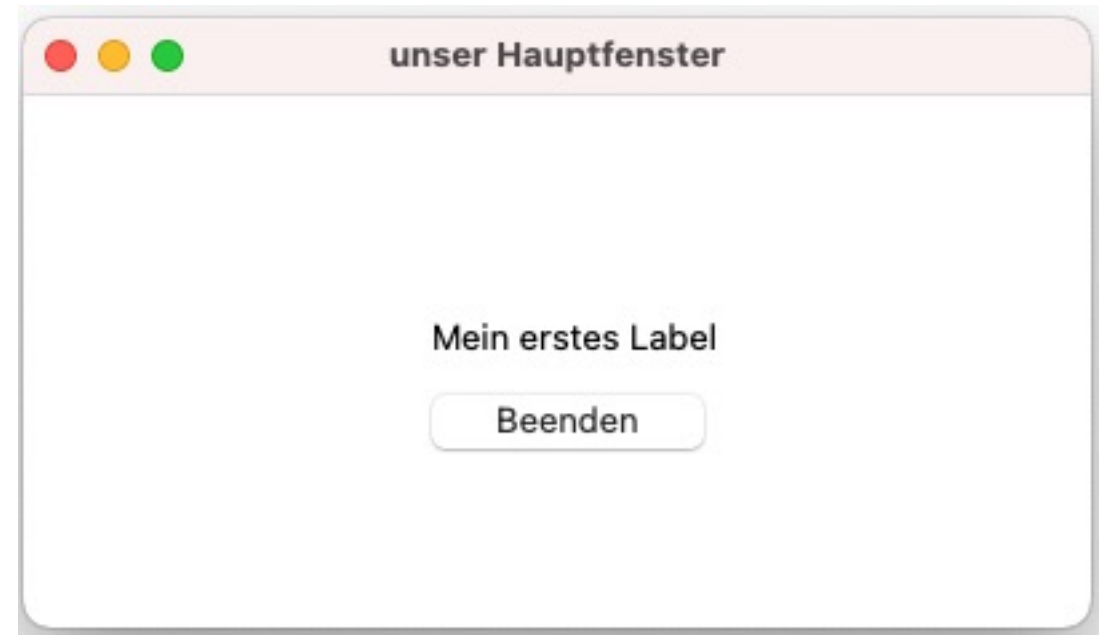
Die zwei bekanntesten Widgets/Controls sind der Text (engl: label) und der Knopf (engl: Button).  
Nachfolgend siehst du den Quellcode zum Hinzufügen eines lbl und Buttons:

```
import tkinter

form = tkinter.Tk()
form.title("unser Hauptfenster")
form.geometry('400x200')

# Label und Button hinzufügen
lbl = tkinter.Label(form, text="Mein erstes Label")
lbl.place(x=150, y=80)
btnEnd = tkinter.Button(form, text="Beenden")
btnEnd["height"] = 2
btnEnd["width"] = 10
btnEnd.place(x=150, y=100)

# Endlosschleife
form.mainloop()
```



Ein Widget wird also zuerst erstellt und mit und einem Fenster (form) zugewiesen. Danach kann das Widget mit place() gesetzt werden. Die Widgets passen sich in diesem Fall nicht richtig dem Fenster an. Das wird ein Teil der Übungsaufgaben sein, daher musst du dich erst einmal damit begnügen. 😊

Aktuell passiert noch nichts, wenn wir auf den Button klicken. Es kann daher in der Parameterliste von BtnEnd eine Funktion definiert werden, welche aufgerufen wird, sobald der Button angeklickt wird. In unserem Fall soll das Fenster geschlossen werden. Das Codesnippet hierfür lautet:

```
7
8  def btnEnd_click():
9      form.destroy()
10
11
12  # Label und Button hinzufügen
13  lbl = tkinter.Label(form, text="Mein erstes Label")
14  lbl.place(x=150, y=80)
15  btnEnd = tkinter.Button(form, text="Beenden", command=btnEnd_click)
```

Wichtig ist hier, dass die Funktion **VOR** dem **command** definiert wird. Ansonsten kann Python den Aufruf von `btnEnd_click` nicht zuordnen.

Ein weiterer Tipp: tkinter bietet das Modul `ttk`. Dieses wiederum enthält 18 widgets, die moderner aussehen als die tkinter-Widgets. Daher kann anstatt tkinter auch `ttk` verwendet werden.

Hierzu wird anstatt:

```
import tkinter
```

das `ttk` Modul importiert:

```
from tkinter import ttk
```

Hier endet bereits die Einführung in Tkinter. Sofern du dem Pythonkurs gefolgt bist und diesen gewissenhaft bearbeitet hast, bist du jetzt in der Lage jedes Widget einzusetzen. Die Liste von verfügbaren Widgets lässt sich im Internet nachschlagen. Die Funktionsweise ist immer ähnlich.

Einige Ratschläge: In der Praxis sollte eher **PyQT** verwendet werden. Damit sind wesentlich moderne Oberflächen möglich (bzw. mit weniger Aufwand zu erreichen). Der **PyQT-Designer** erleichtert die Arbeit in größeren Projekten enorm.

Im Internet finden sich eine Vielzahl von Tutorials für dieses Framework. Geachtet werden sollte darauf, dass immer:

1. Der **OOP**-Ansatz verwendet wird!
2. **Stylesheets** genutzt werden (also das separieren von Design und Logik)
3. Evtl. das **MVC**-Muster Anwendung findet

**Aufgabe K22 | A1:** Erstelle ein Fenster mit einem Button, welcher auf Knopfdruck die das Fenster rot färbt.

**Aufgabe K22 | A2:** Erstelle ein Fenster mit einem Button (der Name lautet „klick mich!“). Immer wenn der Button geklickt wird, verschwindet dieser und erscheint irgendwo anders auf dem Fenster (zufällig).

**Aufgabe K22 | A3:** In dieser Aufgabe sollst du einen Taschenrechner nachprogrammieren. Du benötigst 2 Inputboxen, ein Label und die Grundrechenarten als Button. In die Inputboxen wird jeweils eine Zahl eingegeben und das Ergebnis erscheint im Label. Achte hier darauf, dass Fehleingaben vom Benutzer (Buchstaben anstatt zahlen) entsprechend abgefangen werden. Vielleicht erscheint dann eine Nachricht, dass bitte nur Zahlen eingegeben werden.

**Aufgabe K22 | A4:** In diesem Programm möchtest du Eingaben vom Hauptfenster, in einem Subfenster anzeigen lassen. Der Benutzer gibt seinen Namen in eine Inputbox ein. Diese wird dann in einem neuen Fenster angezeigt.

**Aufgabe K22 | A5:** Dein nächstes Programm enthält eine Menüleiste. In der Menüleiste existiert der Punkt „about“. Wird dieser angeklickt erscheint ein Fenster mit deinem Namen.

**Aufgabe K22 | A6:** In den nächsten Aufgaben wollen wir Daten manipulieren, die von außerhalb kommen. Schreibe eine GUI, in der man eine .csv Datei laden kann. Die Daten erscheinen in einer Inputbox.

Die Daten sollen überschreibbar sein. Danach kann mit einem „save“-Button der neue Inhalt zurück in die .CSV Datei geschrieben werden.

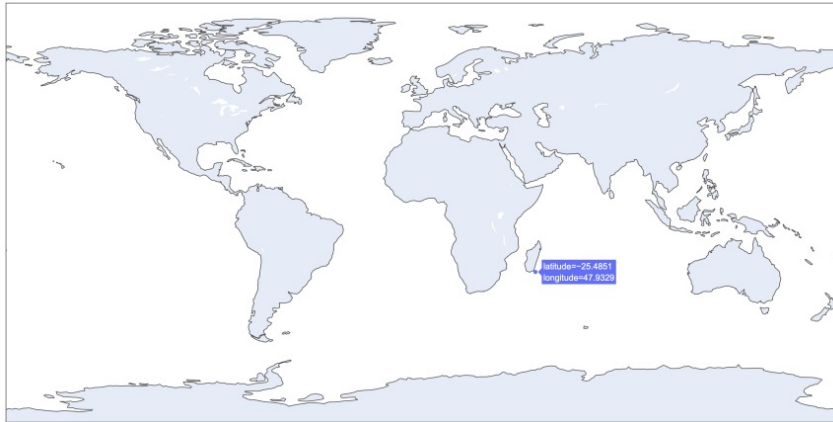


**Aufgabe K22 | A7:** Programmiere das Spiel Tic-Tac-Toe in einer GUI nach. Hierzu benötigst du das Spielfeld und 9 Buttons. Nacheinander können die Spieler jeweils ein X oder O setzen. Sofern ein Spieler gewinnt, soll das Spiel automatisch mit einer entsprechenden Nachricht beendet werden. Sind alle Buttons betätigt worden, aber kein Spieler hat gewonnen, erscheint die Nachricht „unentschieden“.

# Kapitel 23 | Diverse Aufgaben

**Aufgabe K23 | A1:** Gegeben ist folgende API: <http://api.open-notify.org/iss-now.json> .  
Zudem dürfen folgende Module verwendet werden: pandas, requests und plotly.

Schreibe ein Programm, welches die aktuelle Position der ISS graphisch ausgibt. Das Ergebnis sieht wie folgt aus:



Tipps:

- 1: Die Module müssen vorher mit “pip install pandas” usw. installiert werden. Das gesamte Programm benötigt lediglich zwischen 10-20 Zeilen Code.
- 2: Die Daten liegen in einem „falschen“ Format vor, wenn sie über den Request geholt werden. Damit kann plotly nicht arbeiten. Daher müssen die Dateien entsprechend verändert werden, damit diese lesbar sind.

# Feedback

Nachfolgend wird dein gesamtes Wissen gefordert, um praxisnahe Projekte zu entwickeln. Der Grundlagenkurs ist an dieser Stelle beendet.

Ich freue mich immer über Feedback zum Skript, den Übungsaufgaben und co. Solltest du Verbesserungsvorschläge haben, kannst du mir gerne eine Teamsnachricht bzw. E-Mail zukommen lassen.

Die nachfolgenden Projekte setzen **alle** Themen im Skript voraus. Darüber hinaus werden teilweise Technologien oder Frameworks verlangt, die im Unterricht nicht behandelt wurden. Das ist an sich kein Problem, setzt aber ein grundlegendes Verständnis der Sprache voraus.

Die Projekte sind modular aufgebaut und können daher schrittweise gebaut werden. Der Schwierigkeitsgrad der einzelnen Projekte ist abhängig von den Funktionalitäten die implementiert werden sollen.

Unser Seriennummergenerator besteht aus 2 Teilen. Dem Generator an sich und einer Funktionalität zum testen von validen Seriennummern.

Funktionalität:

- Der Generator soll wahlweise via Konsole oder GUI angesteuert werden können.
- Es soll möglich sein, zwischen verschiedenen Seriennummertypen zu wählen (z. B. nur Zahlen, Buchstaben, Länge).
- Mit einem Button (GUI) bzw. Zahl (Konsole) können Seriennummern erzeugt werden.
- Diese werden in einer .csv Datei (valid\_numbers.csv) zeilenweise abgespeichert
- Das Programm bietet die Möglichkeit eine Seriennummer einzugeben. Danach prüft das Programm anhand der Datei valide\_number.csv ob die Seriennummer dort vorliegt. Falls ja, wird „Valid“ als Text ausgegeben. Falls nicht erscheint „not Valid“.
- Wurde eine Seriennummer als Valid eingestuft, wird in der Datei die Seriennummer als verbraucht markiert.
- Versucht man eine bereits verwendete Seriennummer einzugeben, erscheint die Nachricht „already used“

Das Projekt erweitert die Aufgabe K9 | A12. Hier noch einmal die Ursprungsversion, allerdings erweitert:

In dieser Aufgabe sollst du ein Kampf zwischen verschiedenen Charakteren simulieren (**mit GUI & OOP-basiert!**), basierend auf dem D&D-Regelwerk (sehr vereinfacht). Am Anfang können 2 Spieler jeweils einen Charakter auswählen (Krieger, Magier oder Schurke). Danach kämpfen diese abwechselnd solange gegeneinander, bis 1 Spieler keine Lebenspunkte mehr besitzt. Danach wird eine Nachricht angezeigt welcher Spieler gewonnen hat. Es fängt der Spieler an, der die höhere Initiative „würfelt“. Die Klassen besitzen folgende Werte bzw. Fähigkeiten (Funktionen!):

## P2 | RPG-Fighting-Game

Magier:

Initiative: 1d6

Lebenspunkte: 1d6

Fähigkeiten:

Fireball: 2d7 Schaden (benötigt eine Runde zum aufladen, die zweite zum abfeuern)

Magic\_Missile: 1d6 Schaden

Mirror-Image: Mit einer Wahrscheinlichkeit von 50% wird kein Schaden am Magier für 2 Angriffe verursacht.

Healing: Heilt 1d4 (kann 1x verwendet werden)

Schurke:

Initiative: 1d10

Lebenspunkte: 1d8

Fähigkeiten:

Sneak-Attack: Falls der Schurke zuerst angreift. Dann fügt der nächste Angriff zusätzlich 1d3 Schaden zu.

Dolchangriff: 2x 1d4 Schaden

Dirt: Der Schurke wirft dem Gegner Dreck in die Augen. Trifft er, verursacht der nächste Angriff keinen Schaden.

Healthpotion: Heilt 1d6 (kann 1x verwendet werden)

Erweiterungsmöglichkeiten:

- Man fügt Monster dem Spiel hinzu.
- Spieler können alleine oder zu 2 gegen ein Monster kämpfen. Vor dem Kampf wird die Initiative ausgewürfelt und danach gespielt.
- Das Monster soll von der „KI“ gesteuert werden. Das Pythonprogramm übernimmt also das Monster.
- Implementiere verschiedene Schwierigkeitsgrade. Auf einem höheren Schwierigkeitsgrad kann das Monster z. B. mehr HP besitzen, stärkere Angriffe oder intelligenter sein. Ein intelligentes Monster greift vermutlich nicht den Krieger zuerst an, sondern den Magier!
- Schreibe eine Funktion, die den gesamten Kampf in eine .log Datei schreibt, um danach den epischen Kampf noch einmal nachverfolgen zu können.
- Implementiere einen Button „Custom“, in der eine .csv Datei eingelesen wird. Diese soll es ermöglichen einen eigenen Charakter zu implementieren (z. B. Barbar, Paladin usw.). Hierfür musst du entsprechend dokumentieren, wie die CSV Datei aufgebaut sein muss, damit der Import funktioniert.
- Füge visuelle Effekte deinem Programm hinzu!



Erweiterungsmöglichkeiten 2:

Jetzt wollen wir das Fighting-Game um ein Adventurepart erweitern:

- Die Spieler betreten nacheinander 10 Level. Ein Lvl kann bestehen aus einem Monster, einem Shop oder einem Dorf. Welcher Lvl erscheint ist zufallsbasiert. Ein Monster führt immer zu einem Kampf. Allerdings lässt das Monster jetzt Goldstücke fallen, die gerecht aufgeteilt werden. In einem Shop können neue Items gekauft werden. Ein Krieger könnte vllt. ein neues Schwert kaufen, welches mehr Schaden verursacht oder einen Heiltrank kaufen. Ein Magier könnte einen neuen Zauberspruch kaufen usw.
- Im Dorf besteht die Möglichkeit gegen Gold in einem Gasthaus zu schlafen. Dadurch werden die HP wieder vollständig geheilt.
- Im letzten Lvl (10) erscheint ein Bossmonster, welches besonders hart ist.
- Sollten Spieler auf dem Weg sterben, werden die Namen der Spieler, das erreichte Level und das Gold in einer .CSV Datei ("Highscore") gespeichert. Diese kann beim Spielstart eingeblendet werden.

### Erweiterungsmöglichkeiten 3:

- Das Spiel ist sehr wahrscheinlich nicht ausbalanciert. Passe die einzelnen Werte so an, dass das Spiel spannend wird.
- Verbinde Projekt 1 mit 2. Im Projekt 2 existiert ein Button „Seriennummer eingeben“, in die eine Seriennummer eingegeben werden kann. Steht diese in der Datei „valid\_numbers.csv“, wird ein DLC freigeschaltet. Dieser beinhaltet eine neue Klasse und ein neues Monster. (natürlich ist dieses Vorgehen sicherheitstechnisch eine Katastrophe). Es geht lediglich um die Implementierung.
- Du kannst natürlich auch versuchen, die Implementierung so anzupassen, dass niemand die Seriennummern derart einfach manipulieren kann.
- Baue neben Gold weitere Items ein, die von Monstern zu einer geringen Wahrscheinlichkeit gedropped werden können. Das können Heiltränke, Ausrüstung oder Schriftrollen sein.

### Erweiterungsmöglichkeiten 4:

- Alleine spielen ist oft langweilig und einen Multiplayermodus würde den Rahmen bei weitem sprengen. Daher solltest du im Hauptmenü einen Button “With Companion“ implementieren, in der ein Einzelspieler mit der KI zusammen spielen kann. Die KI übernimmt die Rolle von einer Klasse und handelt komplett eigenständig. Hier wäre ein DLC „Healer“ eine klasse Idee.
- Der DLC beinhaltet die Klasse „Priester“, welches lediglich von der KI ausgewählt werden kann. Diese kann 1d4 Schaden heilen oder den Gegner verfluchen, damit dieser zu einer gewissen Wahrscheinlichkeit daneben schlägt. Eine intelligente KI wird immer verfluchen, so lange der Hauptcharakter oder sie selbst keinen Schaden erhalten hat. Erst wenn es brenzlich wird, wird die KI anfangen zu heilen.

Ideen zur Implementierung: Für die Lvl reicht ein Imagefile, welches geladen wird. Etwas passende Hintergrundmusik kann ebenfalls eingebunden wird.

Steam, Epic Games Store und co. bieten die Möglichkeit Spiele zu sortieren, zu bewerten, zu favorisieren usw. Allerdings sind diese Plattformen aufgebläht mit Funktionen, die wir nicht benötigen. Daher schreiben wir in diesem Projekt unsere eigene Spieleverwaltung:

Die Spieleverwaltung benötigt zwingend eine GUI und vorzugsweise eine Datenbank. Folgende Funktionalitäten sollen ermöglicht werden:

- Spiele können hinzugefügt, geändert und gelöscht werden
- Es kann ein Fenster angezeigt werden mit Spielnamen, einem Bild des Spieles (benötigt Importfunktionalitäten), einem kurzen Text, dem aktuellen Preis und einer eigenen Bewertung (Sterne).
- Preise können via Amazon oder andere Seiten in das Programm geladen werden (how to: Google)
- Spiele können in Genres gruppiert werden.
- Jedes Spiel beinhaltet einen Kauflink, der auf eine Plattform führt, auf der das Spiel gekauft werden kann.
- Es besteht die Möglichkeit, die jeweilige Spielseite optisch ansprechend zu drucken (z. B. als Wunschzettel)

# Changelog:

Version 0.10: Kapitel 0 & 1 erstellt

Version 0.11: Inhaltsangabe erweitert; Kapitel 2 & 3 erstellt

Version 0.12: Kapitel 4 erstellt

Version 0.13: Fehler verbessert, Farben hinzugefügt

Version 0.14: Kapitel 5,6,7,8 hinzugefügt

Version 0.15: Kapitel 9 & 10 hinzugefügt

Version 0.16: Kapitel 9 Rekursion hinzugefügt

Version 0.17: Kapitel 1 erweitert

Version 0.18: Übungsaufgaben hinzugefügt; IDE geändert; Fehler korrigiert, Ansprache angepasst

Version 0.19: Rechtschreibfehler korrigiert; Nummerierung angepasst und korrigiert

Version 0.20: Kapitel 10-16 hinzugefügt; Übungsaufgaben in verschiedenen Kapiteln hinzugefügt

Version 0.21: Kapitel 17-21 hinzugefügt.

Version **1.00**: Kapitel 22 hinzugefügt, Vorwort ergänzt.

Version 1.01: Kapitel 23 hinzugefügt

Version 1.02: Absatz zu PyQt in Kapitel Tkinter hinzugefügt.

Version 1.03: Rechtschreibfehler und kleinere inhaltliche Fehler korrigiert.

Version 1.04: Update: Programmierbeispiele Kap 1-9

Version 1.05: Zwei Fehler in Kap. 11 korrigiert.

Version 1.06: Fehler in Kapitel Dictionary korrigiert

# Changelog:

Version 1.07: Hinweise gekürzt, diverse Rechtschreibfehler korrigiert

Version 1.08: Fehler in Grafik (Kap 14) geändert

## Quellen:



Icon made by ultimatearm

<https://www.flaticon.com/authors/ultimatearm>



Icon made by pixelperfect

<https://www.flaticon.com/authors/pixelperfect>

Rekursion-Meme

<https://i.redd.it/0wap3cp4khm01.jpg>

## Nützliche Literatur:

*Ich übernehme keine Verantwortung bezüglich der Inhalte der folgenden Links!  
Sie dienen lediglich der Einarbeitung in das Thema. Sollten Links nicht mehr funktionieren  
oder auf den folgenden Seiten „problematische“ Inhalte publiziert werden, bitte ich um eine Rückmeldung.  
Der betreffende Link wird dann entfernt!*

Python-Kurs, der die relevanten Grundlagen behandelt:

[https://www.python-kurs.eu/python3\\_kurs.php](https://www.python-kurs.eu/python3_kurs.php)

5-Stunden Tutorial, welches die Grundlagen von Python abdeckt.

<https://www.youtube.com/watch?v=rfscVS0vtbw&t=15167s>

Ein weiteres Python-Tutorial (in englisch):

<https://www.w3schools.com/python/default.asp>