Ein Bild, das Pixel, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.3.3 Eigene Klassen implementieren**

Bisher hat das Framework sich um unsere Klassen gekümmert, doch das ändern wir jetzt. In den folgenden Kapiteln werden Sie selbst einige Klassen des Frameworks neu implementieren. Wir beginnen dabei mit unserem Helden, für welchen wir nun das folgende, gekürzte Klassendiagramm verwenden.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp : int, richtung : str, w : bool)  geh()  links()  rechts()  zurueck() |

**Aufgabe 1**

Laden Sie in der schueler.py Level 35. Ein leeres Level erscheint, in dem eigentlich unser Held stehen sollte. Das Framework kümmert sich selbst um die Erzeugung der Objekte anhand der Level-Datei, aber dazu muss die Helden-Klasse auch verfügbar sein.

1. Ergänzen Sie in Ihrem bekannten Programmierbereich die folgenden Zeilen:

**class** **Held**:

**def** **\_\_init\_\_**(self,xp,yp,richtung,w):

self.x = 0

Führen Sie das Programm aus.

1. Ergänzen Sie nun selbstständig die fehlenden Befehle, um der neuen Helden Klasse alle Attribute zuzuweisen. Verwenden Sie dabei sinnvolle Standardwerte, welche dem Klassendiagramm entsprechen. Den Standardnamen eines Helden dürfen Sie frei wählen, der Typ des Helden ist „Held“.
2. Möglicherweise ist Ihnen bereits aufgefallen, dass das Level nicht abgeschlossen werden kann, obwohl alle Attribute gesetzt und der Held sichtbar ist. Das liegt daran, dass im Level genau festgelegt ist, wo der Held steht, in welche Richtung er schaut und ob „er“ weiblich ist. Letzteres steuern Sie über level.lade(35,**weiblich=…**) sogar selbst!

Der Konstruktor des Helden erhält über Parameter genau diese Standardwerte:

Held(xp,yp : int, rp : str, w : bool)

Die Parameter sollten Sie dann auch verwenden. Ändern Sie die Zeilen daher wie folgt ab:

self.x = xp

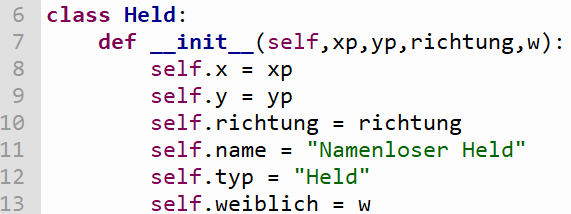
1. Der Parameter für die Richtung heißt wie das Attribut selbst. Der Parameter für weiblich enthält ebenfalls kein p. Macht dies Probleme?
2. Sobald Ihre Lösung **Level 35** löst („Level abgeschlossen“), laden Sie auch **Level 36** mit der gleichen Lösung. Ist dies ebenfalls direkt abgeschlossen, so haben Sie das Level gelöst und können fortfahren!

**Merke:**

Eine Klasse kann anhand des Klassendiagramms schnell implementiert werden. Eingeleitet wird eine Klassendefinition mit dem Schlüsselwort **class**, gefolgt vom Namen der Klasse und einem Doppelpunkt.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp,richtung,w)  … |

Beispiel



Alle Methoden der Klasse werden dann eingerückt implementiert, beginnend beim **Konstruktor** der Klasse. Obwohl er beim Aufruf wie die Klasse selbst heißt, heißt er bei der Implementierung **\_\_init\_\_** und enthält als Parameter immer self. Die übrigen Parameter werden aus dem dem Klassendiagramm übernommen.

Alle Attribute werden dann in der Form self.attribut initialisiert. Entweder:

* Durch Verwendung eines Parameters, z.B. bei x oder weiblich
* Durch Setzen eines sinnvollen Standardwerts, z.B. bei typ oder name.

**Aufgabe 2**

Im Konstruktor können nicht nur stumpfe Werte oder Parameter gesetzt werden. Ändern Sie den Konstruktor so um, dass „Namenloser Held“ bei männlichen und „Namenlose Heldin“ bei weiblichen Helden als Name gesetzt wird.

*Hinweis: Obwohl es sinnvoll wäre, tun Sie dies nicht beim Attribut Typ, da in diesem Fall die Spiellogik des Frameworks gebrochen wird.*

**Aufgabe 3**

Jetzt fügen wir der Helden-Klasse die Fähigkeit hinzu, sich zu bewegen. Unser Ziel wird es sein, die Funtionalität des Frameworks Schritt für Schritt wiederherzustellen, so dass Sie nach Implementierung der Klassen(methoden) **die Level mit der Tastatur** lösen können. Sie müssen also ab sofort keine Programme zum Lösen der Level selbst mehr schreiben.

1. Öffnen Sie **Level 37**. Dieses Level erfordert zum Lösen, dass sich der Held zum markierten Zielfeld bewegt und alle Bewegungsmethoden implementiert sind.
2. Fügen Sie der Klasse Held eine neue Methode **geh** hinzu, welche die gleiche Einrückung wie der Konstruktor besitzt:

**def** geh(self):

**if** self.richtung == "up":

    self.y = self.y + ...

Implementieren Sie die Methode geh vollständig. Testen Sie Ihre Implementierung, indem Sie Level 37 starten und den Helden mit Pfeiltaste oben **⬆️** bewegen.

1. Obwohl die Methode **geh** laut Klassendiagramm keine Parameter besitzt, müssen Methoden einer Klasse in Python immer den Parameter self erhalten. Testen Sie jetzt den Befehl held.geh() im Hauptprogramm, also dort, wo Sie auch zuvor programmiert hatten (Einrückung 0, z.B. direkt vor der Zeile framework.starten()).
2. Implementieren Sie auf die gleiche Art und Weise die Methoden links, rechts und zurueck, um Level 37 mithilfe der Pfeiltasten zu lösen.

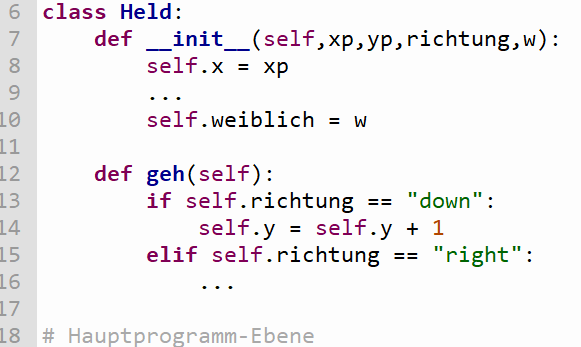
**Merke:**

Methoden einer Klasse können nach dem Konstruktor mit **def** methode(…) auf gleicher Einrückungsebene hinzugefügt werden.

Innerhalb der Klasse können die Attribute mit self.attribut referenziert werden. Das self ist dabei sozusagen eine **Selbstreferenz**, auf genau das Objekt, welches gerade die Methode ausführt. Dieses **self** muss als Parameter jeder Klassenmethode aufgeführt werden, obwohl dieser beim Aufruf später nicht mehr sichtbar ist.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp,richtung,w)  geh()  links()  … |

Beispiel



Der erste Befehl, welcher sich wieder auf der Einrückungsstufe der Klassendefinition befindet, ist nicht mehr Teil der Klasse, sondern z.B. Teil des Hauptprogramms. Klassen können theoretisch auch auf Ebene eines Unterprogramms definiert werden.

**Übungsaufgaben**

* Passen Sie den Konstruktor so an, dass nur up, down, left und right als Werte für richtung akzeptiert werden. Ansonsten wird down als Standard gesetzt
* Passen Sie den Konstruktor so an, dass nur positive Werte für x und y gesetzt werden können
* Schreiben Sie einen kleinen Namensgenerator, welcher beim Erzeugen des Helden einen zufälligen Namen erzeugt und setzt (entweder als Unterprogramm erzeuge\_name() oder direkt im Konstruktor

**8.3.4 Modularisierung**

Ein Bild, das Text, Cartoon, Puzzle enthält.

KI-generierte Inhalte können fehlerhaft sein.

Neben unserer Klasse Held gibt es noch Hindernis, Knappe, Monster, Schluessel, Tuer, Tor und weitere, welche wir noch nic ht genannt haben. Sie können sich vorstellen, dass dies sehr **unübersichtlich** wird, wenn der Code all dieser Klassen in der schueler.py liegen würde.

Ein Bild, das Text, Screenshot, Rechteck, Design enthält.

KI-generierte Inhalte können fehlerhaft sein.

Aus diesem Grund lagern wir die Klassen in eigene Dateien aus, von wo aus wir sie später nach belieben in unsere Programme **importieren** können. Dies entspricht dem Prinzip der **Modularisierung**.

**Aufgabe 1**

1. Laden Sie **Level 38**. Das Level sollte leer sein.
2. **Ein Bild, das Text, Screenshot, Schrift, Dokument enthält.

   KI-generierte Inhalte können fehlerhaft sein.**Erstellen Sie eine leere Datei in Python und kopieren Sie dort die bisherige Klassendefinition des Helden hin. Speichern Sie diese Datei als **held.py** im Unterordner Klassen. Die Datei sollte dann (in etwa) so aussehen:
3. Löschen Sie jetzt die Klassendefinition aus der schueler.py. Sie sollte jetzt wieder so aussehen. Laden Sie erneut Level 38 und lösen Sie es mit der Tastatur.

**Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.**

**Aufgabe 2**

1. Laden Sie **Level 39**. In diesem Level wurde wohl vergessen, den Helden zu platzieren. Versuchen Sie jetzt durch Erzeugung eines passenden Helden-Objekts sowie Hinzufügen zum Level das Level abzuschließen. Welche Fehlermeldung erscheint?

*Tipp: Schauen Sie sich nochmal 8.3.2 an.*

Normalerweise kümmert sich das Framework um die Erzeugung der Objekte. Beim Programmieren des Frameworks wurde deswegen darauf geachtet, dass dieses Zugriff auf alle Klassen besitzt. Da Sie außerhalb des Frameworks ein Helden-Objekt Ihrer eigenen Helden-Klasse erzeugen möchten, müssen Sie sich selbst darum kümmern.

1. Um Ihre Heldenklasse zu importieren müssen Sie zunächst den folgenden Befehl ausführen, bevor Sie den Helden erzeugen.

**from** klassen.held **import** Held

Versuchen Sie nun erneut das Level zu lösen.

1. Testen Sie alternativ auch die folgenden Befehle. Welche funktionieren, welche nicht?

**from** klassen.held **import** **\***

**import** klassen.held

1. Was passiert, wenn der Import-Befehl nach dem Erzeugen des Helden steht?

**Erkenntnis**

Um Programmcode in unsere ausführende Datei A aus einer anderen Datei B zu \_\_\_\_\_\_\_\_\_\_\_\_\_, welche relativ zur ausführenden Datei in einem \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ liegt, so müssen wir diesen beim Import-Befehl mit angeben, gefolgt von einem \_\_\_\_\_\_\_\_\_\_\_\_ und dem Namen der \_\_\_\_\_\_\_\_\_\_\_\_.

Wäre Datei **B** im Unterordner **unter**, so wäre der passende Import-Befehl:

from import \*

**Aufgabe 3**

Wir möchten jetzt unseren Helden zufällig bewegen. Dazu steht im Ordner **framework** bereits eine Klasse für einen Würfel zur Verfügung, welche in der Datei werkzeug.py liegt. Hier ist das zugehörige Klassendiagramm:

|  |
| --- |
| Wuerfel |
| seiten : int |
| Wuerfel(s : int)  werfen() : int |

1. Bleiben Sie bei **Level 39**. Dort sollte noch Ihr Code zur Erzeugung des Helden liegen:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. Import-Befehle können zwar an (fast) beliebiger Position im Code stehen (vgl. Aufgabe 1 d) ), sollten aber eher gebündelt am Anfang des Programms stehen. Importieren Sie die Würfel-Klasse genau wie die Helden-Klasse aus dem Unterordner **framework** in Zeile 6, also nach dem Import der Helden-Klasse.
2. Erzeugen Sie jetzt ein neues Würfel-Objekt mit 6 Seiten und verwenden Sie es, um eine zufällige Zahl **zahl** zu erzeugen.
3. Unser Held soll dann genau so viele Schritte machen, wie gewürfelt wurde. Führen Sie das Programm einige Male aus. Ihnen sollten zwei Dinge auffallen:

6. Ersetzen Sie den Import-Befehl jetzt durch die Variante mit **\***, also

**from** framework.werkzeug **import** **\***

Führen Sie Ihr Programm erneut aus. Welche Beobachtung machen Sie? Öffnen Sie die werkzeug.py und finden Sie heraus, woher dieses Verhalten stammt.

**Merke:**

Unterprogramme und Klassen können über den **import** Befehl in andere Programme eingebunden werden. Dabei können entweder **integrierte Standardbibliotheken**, wie z.B. **random** eingebunden werden oder auch **eigene Dateien**, welche sich in **relativer Nähe** zur **importierenden Datei** befinden müssen.

Dabei gibt es grundsätzlich **drei Methoden**, um diese einzubinden.

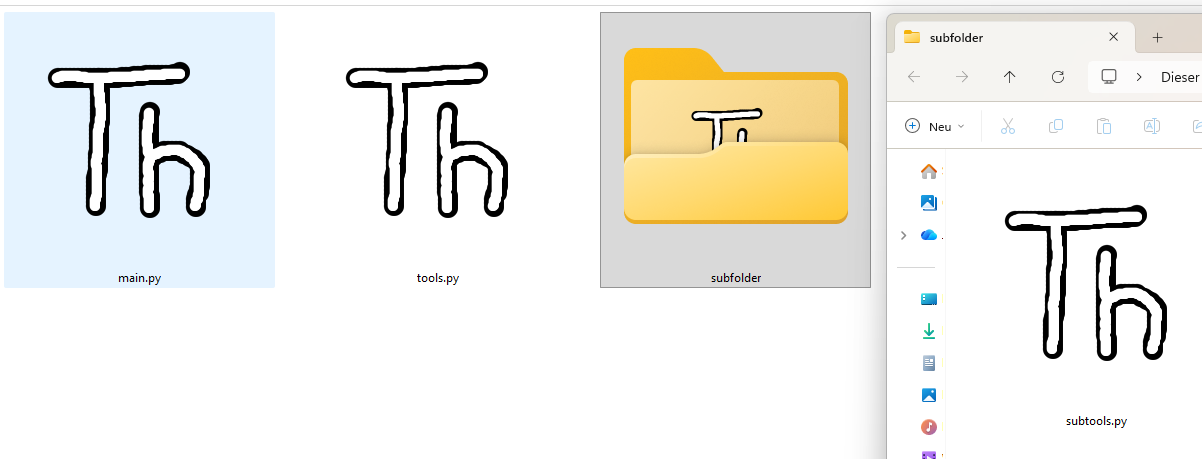
**Variante 1:**

Durch den Befehl **from** modul **import** **\*** wird im Grunde die gesamte Zieldatei in die importierende Datei eingefügt. Dies kann insbesondere dazu führen, dass bestehende Variablen, Unterprogramme oder Klassen durch den Import **überschrieben** werden. Man spricht auch davon, dass der **Namensraum** dann „**zugemüllt**“ wird (*engl. namespace pullution*). Es bedeutet, dass weniger Namen für Variablen, Methoden und Klassen zur Verfügung stehen und sollte daher nur bei kleinen Dateien verwendet werden.

**Beispiel:**

Der gesamte Inhalt der **tools.py** soll in die **main.py** importiert werden. Da sich die **tools.py** im gleichen Ordner wie die **main.py** befindet, schreiben wir

**from** tools **import** \*



**Variante 2:**

Möchte man nur einzelne Dinge importieren, so gibt man diese statt \* explizit an.

**Beispiel:**

Aus der subtools.py sollen die Variablen alter, das Unterprogramm zaehle und die Klasse Datum importiert werden. Sie befindet sich im Unterordner subfolder. Wir schreiben:

**from** subfolder.subtools **import** alter, zaehle, Datum

**Variante 3:**

Möchte man ohne die Nachteile von Variante 1 alles aus einem Modul importieren, so kann dieses mit **import modul** importiert werden. Dann muss der Modulname bei jedem Zugriff vorangeschrieben werden.

**Beispiel:**

**import** random importiert die gesamte random Bibliothek. Um die Methoden zu verwenden schreiben wir z.B. x = random.randint(1,5) oder print(random.uniform(a,b)).

**Übung**

Implementieren Sie im Unterordner klassen eine eigene Klasse Wuerfel. Importieren Sie diese dann auf verschiedene Arten in der schueler.py und nutzen Sie diese.

|  |
| --- |
| Wuerfel |
| seiten : int  bisherige\_summen : int |
| Wuerfel(s : int)  werfen() : int  reset()  mehrfach\_wurf(anzahl : int) : int |