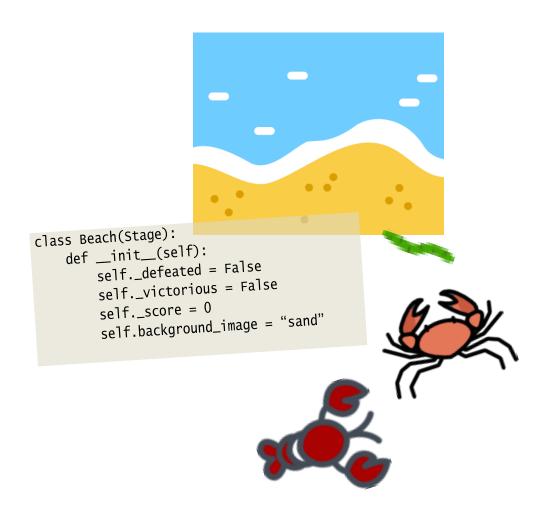
Einführung in die objektorientierte Programmierung mit Python und Pygame Zero

Robert Garmann

Version 0.9.2 vom 26. Mai. 2018



2 Inhalt

Inhalt

1	Einle	itung	5
	1.1	Was ist Python?	5
	1.2	Was ist Pygame?	5
	1.3	Was ist Pygame Zero?	5
	1.4	Für wen ist dieses Buch?	6
	1.5	Vorbereitungen	7
	1.6	Credits	8
	1.7	Feedback	8
2	Pyga	me Zero kennenlernen	9
	2.1	Ein minimales Spiel	9
	2.2	Den Spielbereich festlegen	9
	2.3	Spielfigur	11
	2.4	Koordinatensystem	. 12
	2.5	Wartbare Software	. 13
	2.6	Eine zweite Spielfigur	15
	2.7	Variablen	15
	2.8	Anweisungsreihenfolge	. 16
	2.9	Gameloop	. 18
	2.10	Spielmechanik	. 21
	2.11	Ereignisverarbeitung	.22
	2.12	Spielmechanik II	.23
	2.13	Zusammenfassung	.24
	2.14	Übungen	. 25
	2.15	Exkurs: Bessere Kollisionserkennung	. 25
3	Objel	kte	. 27
	3.1	Funktionsbibliothek	. 27
	3.2	Zwei Himmelsobjekte	. 27
	3.3	Objektverhalten und Objektzustand	.30
	3.4	Gemeinsames Verhalten "orbit"	. 31
	3.5	Zustandsabfrage	.32
	3.6	Methode oder Funktion?	.34
	3.7	Zusammenfassung	.34
4	Bühn	e frei für Little Crab	.35
	4.1	Titel	.35
	4.2	Stage	.35
	4.3	Game Assets	.36
5	Wir s	chreiben eine Klasse "Crab"	.38
	5.1	Die erste Spielfigur	.38
	5.2	Die Krabbe in Bewegung setzen	.38
	5.3	Fensterrand	.39
	5.4	Zufälliges Verhalten	.40
	5.5	Eine zweite Krabbe und prozedurale Zerlegung	. 41
	5.6	Verhaltensparameter	.42
	5.7	Initialisierungsfunktion	.43
	5.8	Verhaltensfunktion im Krabbenobjekt speichern	.44
	5.9	Zerlegung durch Klassen	
	5.10	Randbemerkung: Didaktik meets Technik	.48

6	Krabl	beninvasion	50
Ĭ	6.1	Zufällige Startposition	_
	6.2	Krabbeninvasion	•
	6.3	" und Action bitte"	J
7		ben fressen Würmer	•
1	7.1	Worm-Klasse	
	7.1		• .
0		Würmer fressen	
8		mer fressen Krabben	• ,
	8.1	Hummer hinzufügen	= -
	8.2	Tastatursteuerung	· ·
	8.3	Das vollständige bisherige Programm	
9	Spiel	ende	
	9.1	Die update- und draw-Funktionen	
	9.2	Bühnenmechanik	62
	9.3	Einen Text anzeigen	63
	9.4	Eine Beach-Klasse	65
	9.5	Effizienz	67
	9.6	Kapselung	68
	9.7	Effiziente und wartbare Detektion einer Niederlage	71
	9.8	clock	73
	9.9	Neustart statt Spielende	75
	9.10	Zusammenfassung	76
10	Spiel	standstand	
	10.1	Score	•
	10.2	Umwandlung zwischen Zeichenketten und Zahlen	,
	10.3	Score bei jedem gefressenen Wurm erhöhen	, -
	10.4	Spiellevel	
	10.5	Das vollständige bisherige Programm	
11		ierte Spielobjekte	
' '	11.1	Daumenkino	•
		Konstanten	
	11.3	Bildzähler	
	11.4	Streckenzähler	,
	11.5	Innere Werte	•
	11.6	Hummer animieren	
	11.7	Würmer animieren	
	11.8	Das vollständige bisherige Programm	
12		Krabbe wird unverwundbar	
	12.1	Schutzschild-Grafiken erstellen	•
	12.2	Schutzschild anzeigen	•
	12.3	Ein schützender Schutzschild	
	12.4	Energie	•
13	Startl	bühne	112
	13.1	Eine Klasse für die Startbühne	112
	13.2	Das Spiel mit der Leertaste starten	112
	13.3	Das Spiel mit der Maus starten	114
	13.4	Nach Spielende wieder zurück zur Startbühne	117
	13.5	Ein Quit-Button	119

4 Inhalt

14	Musik	(122
	14.1	Game Assets	122
	14.2	Sound-Button	122
	14.3	Sound-Button in beiden Bühnen verwenden	123
	14.4	Das vollständige Programm	125
		etzt?	
		ng	
		Dateien	
	16.2	Dokumentation zu pazo.pv	132

Robert Garmann

Fakultät IV – Wirtschaft und Informatik

Hochschule Hannover

 $\hbox{E-Mail: } robert.garmann@hs-hannover.de$

1.1 Was ist Python?

1 Einleitung

1.1 Was ist Python?

Python¹ ist eine Programmiersprache. Sie gilt wegen ihrer reduzierten Syntax als einfach zu erlernen. Python wird gemeinschaftlich entwickelt. Die Weiterentwicklung wird durch die gemeinnützige Python Software Foundation gestützt.

Python wurde von Guido van Rossum in Amsterdam entwickelt. Der Name der Sprache und das in Abbildung 1 dargestellte Logo legen zwar eine Verbindung zur



Abbildung 1: Python-Logo (Quelle: Projekt-Webseite¹)

gleichnamigen Schlangengattung nahe. Tatsächlich bezieht sich der Name jedoch auf die Komikergruppe Monty Python.

Die erste Vollversion erschien im Januar 1994 unter der Bezeichnung Python 1.0. Python 2.0 erschien am 16. Oktober 2000, Python 3.0 am 3. Dezember 2008. Python wird beständig weiterentwickelt. Die derzeit aktuelle Version 3.6 wurde am 23. Dezember 2016 veröffentlicht und ist unter einer zur GNU General Public License kompatiblen Open Source Lizenz nutzbar.

1.2 Was ist Pygame?

Pygame² ist eine von Pete Shinners entwickelte Python- **Programmbibliothek**³ zur Spieleprogrammierung⁴. Es enthält Module zum Abspielen und Steuern von Grafik und Sound sowie zum Abfragen von Eingabegeräten (Tastatur, Maus, Joystick). Ziel ist es, Computerspiele entwickeln zu können, ohne auf Low-Level-Programmiersprachen wie C zurückgreifen zu müssen.



Abbildung 2: Pygame-Logo (Quelle: Projekt-Webseite²)

Pygame ist seit 2000 ein Gemeinschaftsprojekt und wird unter der Open-Source-Software GNU Lesser General Public License veröffentlicht. Es gibt einen regelmäßigen Wettbewerb namens PyWeek, um Spiele mit Python zu schreiben (und normalerweise, aber nicht unbedingt mit Pygame). Die Community hat viele Tutorials für Pygame erstellt, die im Internet zu finden sind. Als erste Anlaufstelle hierfür kann der "References"-Abschnitt⁵ des englischen Wikipedia-Artikels über Pygame dienen. Die derzeit aktuelle stabile Version 1.9.3 wurde im Januar 2017 veröffentlicht.

1.3 Was ist Pygame Zero?

Pygame Zero⁶ basiert auf Pygame und wurde mit dem Ziel einer möglichst flachen Lernkurve für Programmiereinsteiger entwickelt. Pygame (ohne Zero) ist nicht für blutige Anfänger geeignet. Eine Einführung in ein minimales Pygame-Spiel kommt nicht ohne die Vermittlung von durchaus schon komplexen Konzepten wie einer Schleife aus. Dagegen begnügt sich Pygame Zero zu Beginn mit einfachen Anweisungen und Zuweisungen. So können auch Lernende ohne Vorkenntnisse schnell zu ersten Erfolgserlebnissen gelangen. Da Pygame Zero auf Pygame basiert, ist für Fortgeschrittene ein fließender Übergang zu anspruchsvolleren Programmierkonzepten möglich.



Abbildung 3:Pygame Zero Logo (Quelle: Projekt-Webseite⁶)

¹ Projekt-Webseite: https://www.python.org/

² Projekt-Webseite: http://www.pygame.org

 $^{^{3}}$ Was in diesem Zusammenhang eine Bibliothek ist, werden wir im Verlaufe dieses Buchs klären.

⁴ Dieser Abschnitt basiert auf https://de.wikipedia.org/w/index.php?title=Pygame&oldid=176883200

⁵ https://en.wikipedia.org/wiki/Pygame#References

⁶ Projekt-Webseite: https://pygame-zero.readthedocs.io

Pygame Zero ist somit eine für Einsteiger konzipierte Programmierumgebung zur Entwicklung von Computerspielen mit der Programmiersprache Python. Sie eignet sich zum Selbststudium und zum Einsatz im Schul- und Hochschulunterricht.

Pygame Zero wurde in der Version 1.0beta1 am 19. Mai 2015 erstmals veröffentlicht. Die aktuelle Version 1.2 wurde am 24. Februar 2018 veröffentlicht und ist unter der Open-Source-Software GNU Lesser General Public License frei nutzbar.

1.4 Für wen ist dieses Buch?

Der Hintergrund dieses Buchs ist eine Projektveranstaltung für Erstsemester der Hochschule Hannover in Informatik-Studiengängen. Studierende kommen mit sehr unterschiedlichem Vorwissen an die Hochschule. Nicht wenige haben noch nie in ihrem Leben programmiert. Dennoch sollen sie in diesem Projekt bereits ab dem ersten Tag damit beginnen, ein Computerspiel auf die Beine zu stellen. Dieses Buch soll denjenigen, die noch keine Erfahrung mit Python haben, eine erste Anlaufstelle sein. Wenn Sie dieses Buch durchgearbeitet haben, haben Sie so viel erste Erfahrung mit Python, dass Sie sich gut anlassbezogen weiterbilden können. Erste Zielgruppe sind also Anfängerinnen und Anfänger in der Programmierung. Deshalb werden in diesem Buch grundlegende Konzepte ausführlich erklärt.

Anders als in "normalen" Einführungsbüchern⁷ in die Programmiersprache Python orientieren wir uns bei der Reihenfolge der eingeführten Konzepte an dem, was wir gerade brauchen, um ein neues Feature in ein Computerspiel einzubauen. Dadurch finden sich etwa die numerischen Operatoren in vielen Abschnitten verstreut. Das Buch hat nicht den Anspruch, die Programmiersprache Python vollständig darzustellen, sondern fast immer nur gerade das, was uns bei der Programmierung eines Spiels weiter bringt. Wir werden sogar einige grundlegende Konzepte weglassen (z. B. while-Schleifen), weil wir zunächst mit for-Schleifen ausreichend versorgt sind. Dennoch werden Sie viel über die Programmierung und über die Programmiersprache Python lernen⁸. Wir erläutern neue Konzepte, wie etwas das Klassenkonzept, sehr ausführlich und besprechen auch Hintergründe, warum dieses Konzept sinnvoll ist. Ziel ist es, dass Sie nicht nur in der Lage sind, Codeschnipsel einzutippen, sondern dass Sie Wissen aufbauen, das Ihnen erlaubt, tiefer in die Programmierung einzusteigen. Am Ende der Lektüre dieses Buchs bietet es sich an, noch einmal ein "normales" einführendes Python-Buch in die Hand zu nehmen. Vieles, was Sie dann in dem "normalen" Python-Buch lesen werden, wird für Sie erstens schneller verständlich sein und zweitens werden Sie vielleicht häufig schon einen guten Anwendungsfall in Ihrem nächsten Spiel-Programmierprojekt im Kopf haben.

Ein Hauptziel des hier vorliegenden Buchs ist also, schnell in Python und Pygame Zero arbeitsfähig zu sein und dabei viel über die Programmierung zu lernen. Außerdem soll vermittelt werden, dass es nicht reicht, Programme zu schreiben, die funktionieren. Wir besprechen verschiedene Möglichkeiten, wie man ein Programm gestalten kann, dass es "schöner" und "übersichtlicher" wird. Wohlgemerkt: schöner für den Programmierer, nicht für den Spieler. Es geht also keineswegs nur darum, ein Spiel irgendwie "zusammen zu wurschteln", sondern wir legen Wert auf Programmcode, den Sie nach Ihrer Heimkehr aus einem zweiwöchigen Urlaub immer noch verstehen, weil er die darin codierten Absichten des Programmierers klar erkennbar dokumentiert.

Dieses Buch kann man wohl nur von vorne nach hinten durchlesen. Es ist nicht vorgesehen, dass Sie Teile überspringen oder z. B. direkt in Kapitel 4 einsteigen. Während der Lektüre wird dringend angeraten, alle Beispiele selbst auszuprobieren. Ansonsten verpassen Sie die Chance, dass Sie ein besseres Verständnis für das Gelesene aufbauen. Alle benötigten Bild-, Sound- und Schriftartdateien sind als Beilage zu diesem Buch verfügbar. Der größte Teil des Buchs basiert zudem auf einer sog. Bibliotheksdatei⁹, die ebenfalls als Beilage zu diesem Buch verfügbar ist.

⁷ Ein empfehlenswertes solches "normales" Einführungsbuch ist von Bernd Klein: Einführung in Python 3, 3. Auflage, Hanser, 2018. Auch sehr empfehlenswert ist die zugehörige Webseite https://python-kurs.eu/.

 $^{^8}$ Deswegen heißt dieses Buch auch "Einführung in die objektorientierte Programmierung ...".

⁹ Ein paar didaktische Anmerkungen zur Bibliotheksdatei seien an dieser Stelle erlaubt:

[•] Pygame Zero sieht vor, dass für ein Spiel mehrere Funktionen implementiert werden: update für die Spielmechanik, draw für die Zeichenoperationen, und Funktionen wie on_mouse_down, on_key_up für Maus- und Tastatur-Eingabe-Ereignisse. Je mehr zusätzliche

1.5 Vorbereitungen 7

Wir wollen noch einen kurzen Überblick über die einzelnen Kapitel geben:

• In Abschnitt 1.5 erfahren Sie, wie Sie Ihren Rechner vorbereiten können, um selbst Beispiele nachzuprogrammieren.

- Kapitel 2 führt anhand eines einfachen Spiels in Pygame Zero ein. Das Spiel ist nicht objektorientiert programmiert. Es geht um grundlegende Pygame Zero Konzepte (Spielbereich,
 Spielfiguren, Koordinatensystem, Gameloop, Spielmechanik, Ereignisverarbeitung) und um
 grundlegende Programmierkonzepte (Anweisungen, Variablen, Funktionen). Auch einfache
 Kontrollstrukturen wie die if-Anweisung werden hier bereits eingeführt.
- Kapitel 3 führt in eine Planetensimulation ein und illustriert daran das Konzept des Objektzustands und des Objektverhaltens. Dieses Beispiel nutzt bereits die o. g. Bibliotheksdatei pgzo.py.
- Ab Kapitel 4 geht es um ein umfangreiches Spiel "Little Crab". Wir werden dieses Spiel im Zuge der weiteren Kapitel immer weiter aus- und umbauen und dabei viele neue Einsichten in Python und Pygame Zero erhalten. Das Spiel basiert in erheblichem Maße auf der Bibliotheksdatei pgzo.py.
- Am Ende geben wir in Kapitel 15 einige Tipps, wie Sie sich weiterbilden können.

1.5 Vorbereitungen

1.5.1 Installation

Sie sollten während der Lektüre die Beispiele immer gleich selbst ausprobieren. Dazu können Sie Pygame Zero auf Ihrem eigenen Rechner installieren. Pygame Zero funktioniert auf Windows, Linux und Mac. Gehen Sie bei der Installation wie folgt vor:

- Zuerst installieren Sie die Programmiersprache Python von der Projekt-Webseite¹.
- Dann installieren Sie direkt Pygame Zero. Die Installation ist im User Guide¹⁰ auf der Webseite beschrieben. Dieser Schritt installiert automatisch alle notwendigen Pakete, insb. die benötigte Pygame-Version.

Die Beispiele dieses Buchs wurden mit den folgenden Versionen getestet:

Funktionen geschrieben werden, die aus dem Gameloop heraus aufgerufen werden, desto undurchdringlicher wird für Anfänger letztendlich die Ausführungsreihenfolge zur Laufzeit. Viele Beispiele für Pygame Zero nutzen vermutlich auch aus diesem Grund die Möglichkeit, in der update-Funktion direkt den Zustand der Tastatur abzufragen, d. h. sie verwenden einen Pull-Stil bei der Ereignisverarbeitung. Wir haben uns daher entschieden, auch die Mauseingabeverarbeitung auf eine Pull-Verarbeitung umzustellen, und dazu ein zusätzliches Objekt mouse_state in der Bibliotheksdatei implementiert.

- Zum zweiten führt die völlige Abwesenheit von Klassen in ersten Pygame Zero Programmen zu einem prozeduralen Programmierstil. Das muss nicht schlimm sein, wenn es um die ersten Schritte in der Programmierung geht. Eine gewisse Grundfertigkeit in der Programmierung mit Objekten lässt sich zudem in Pygame Zero mit der mitgelieferten Actor-Klasse vermitteln. Beispielsweise werden in dem mit Pygame Zero ausgelieferten flappybird. py-Beispiel die Eigenschaften der Spielfigur (Score, Geschwindigkeit, Lebendig,JN) nicht einfach als globale Variablen abgelegt, sondern kurzerhand als weitere Attribute im Actor-Objekt angelegt. So lange wir es mit einer Spielfigur zu tun haben, funktioniert das noch gut. Irgendwann jedoch werden auch von Anfängern programmierte Spiele bald so umfangreich, dass sie viele Instanzen verschiedener Akteursklassen besitzen. Hier darauf zu setzen, dass Studierende selbst Subklassen von Actor bilden, während die Haupt-Funktionen draw und update immer noch "klassenlos" sind, erscheint uns nicht schlüssig. Außerdem dauert es in der Regel nicht lang, bis Studierende die Notwendigkeit für mehrere Spiellevels sehen. Will man nun verschiedene Spiellevels in einer einzigen draw- und in einer einzigen update-Methode verarbeiten, ist das zwar möglich, erschwert aber die Vermittlung wichtiger Prinzipien wie des Prinzips der eindeutigen Verantwortlichkeit und des Geheimnisprinzips. Wir denken, dass das Bühnenkonzept, wie es etwa in Scratch umgesetzt ist, oder das World-Konzept von Greenfoot gut geeignet ist, die Zuständigkeiten der einzelnen Spiellevels klar voneinander zu trennen. Nach unserer Erfahrung kommen Studierende nicht selbst auf die Idee, für verschiedene, sich in Aussehen, Zustand und Verhalten voneinander unterscheidende Spiellevel jeweils eigene Objekte zu erschaffen (oder gar Klassen, wenn es mehrere Instanzen desselben Spiellevel-Typs gibt). Die in der Bibliotheksdatei implementierte Klasse Stage bringt die Konzepte mit, die für Spiele benötigt werden, die aus mehreren Szenentypen bestehen, die ihrerseits mehrere Akteure beheimaten. Statt zweier globaler Funktionen draw und update implementieren Studierende in jeder Bühne separate draw- und update-Methoden.
- Zum dritten bringt die Pygame Zero Actor-Klasse bereits einige hervorragende Möglichkeiten mit, Spielfiguren auf dem Bildschirm zu positionieren. Inspiriert durch das Crab-Beispiel im Greenfoot-Einführungsbuch haben wir uns entschieden, Subklassen von Actor zur Verfügung zu stellen, die darüber hinaus gehende Funktionen (Vorwärtsbewegung, Drehbewegung, Geschwindigkeit, Erkennung des Fensterrandes) bereits mitbringen. Zusätzlich bieten die Subklassen Methoden zur pixelgenauen Kollisionserkennung, da die einfache Hüllrechtecküberlappung in der Regel zu Unzufriedenheit unter Studierenden führt ("der Feind hat mich doch noch gar nicht berührt!"). Die Pygame-Bibliotheksmethoden zur pixelgenauen Überlappungserkennung erschienen uns in der Handhabung zu komplex für Programmieranfänger.

8 1.6 Credits

- Python 3.6.5
- Pygame 1.9.3
- Pygame Zero 1.2

Wichtig ist eine Version von Python, die mit einer 3 beginnt. Ansonsten sind die Beispiele dieses Buchs sehr wahrscheinlich auch mit anderen Versionskombinationen kompatibel.

1.5.2 Informationsquellen

Falls Sie beim Nachvollziehen der Beispiele dieses Buches nicht recht weiter kommen, sollten Sie in Ihrem bevorzugten Internetbrowser Lesezeichen für die folgenden Informationsquellen anlegen:

- Python-Dokumentation mit einer vollständigen Auflistung aller Funktionen der Standardbibliothek (unter der Überschrift "General Index") und einer vollständigen Beschreibung der Sprache: https://docs.python.org/3/index.html.
- Pygame Zero Reference mit einer Erläuterung aller in Pygame Zero eingebauter Objekte: https://pygame-zero.readthedocs.io/en/stable/index.html#reference.
- Ggf. zusätzlich: Pygame-Dokumentation mit einer vollständigen Referenz aller von Pygame angebotenen Objekte: https://www.pygame.org/docs/.

Alle Codebeispiele dieses Buchs sind in Dateiform in einem ZIP-Archiv verfügbar. Der Inhalt des ZIP-Archivs ist im Anhang ab s. 131 aufgelistet.

Außerdem gibt es im Anhang dieses Buchs ab Seite 132 eine Auflistung aller von der pgzo.py-Bibliothek exportierten Klassen und Funktionen.

1.6 Credits

Die Idee des in diesem Buch ab Kapitel 4 präsentierten Spiels "Little Crab" basiert in weiten Teilen auf der Idee des einführenden Buches zu Greenfoot¹¹, welches Ziele für die Programmiersprache Java verfolgt, die mit den Zielen von Pygame Zero hinsichtlich der Programmiersprache Python vergleichbar sind.

Kapitel 2 basiert auf einem von Daniel Pope auf der EuroPython Conference 2016 gehaltenen Vortrag. Daniel Pope ist einer der Hauptentwickler von Pygame Zero. Ein Video des Vortrages ist hier online: https://www.youtube.com/watch?v=Pzs-c-B3RiM.

In den Programmbeispielen wurden Bilder, Sounds, Schriftarten und Musikdateien aus verschiedenen freien Internetquellen verwendet. Die Quellen sind jeweils an der entsprechenden Stelle in Fußnoten benannt.

1.7 Feedback

Über Hinweise von Leserinnen und Lesern zu Fehlern und Verbesserungsmöglichkeiten freue ich mich. Kontaktdaten finden Sie auf Seite 4.

¹¹ Michael Kölling: Einführung in Java mit Greenfoot, Pearson Studium, 2010

2 Pygame Zero kennenlernen

2.1 Ein minimales Spiel

Starten Sie die Python-Programmierumgebung "IDLE", erstellen darin eine leere Datei und speichern diese unter dem Namen myfirstgame.py (s. Abbildung 4).

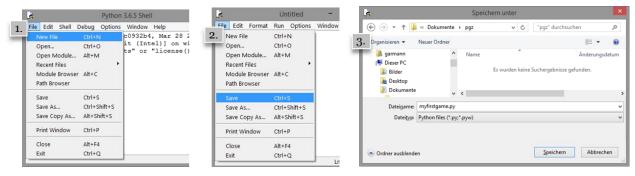


Abbildung 4: Anlegen einer ersten Python-Programmdatei

Herzlichen Glückwunsch, Sie haben soeben ein erstes Pygame Zero Spiel programmiert! Im Ernst: eine leere Datei ist bereits ein gültiges Pygame Zero Programm.

Sie haben nun zwei Möglichkeiten, Ihr neues Spiel zu starten.

2.1.1 Von der Console starten

Öffnen Sie ein Kommandozeilenfenster und wechseln in das Verzeichnis, in dem Sie die Datei gespeichert haben:

```
$ cd Documents/pgz/ Enter J
$ pgzrun myfirstgame.py Enter J
```

In den zwei zuvor abgedruckten Zeilen steht das \$-Zeichen für die Eingabeaufforderung. Dieses wird nicht selbst eingegeben, sondern steht in der Regel schon da. Vielleicht steht auf Ihrem Rechner dort ein anderes Zeichen (z. B. >) oder gar ein ganzes Wort (z. B. meyer@notebook7:~>).

Der letzte Befehl pgzrun myfirstgame.py startet das "Spiel" und öffnet ein Fenster mit schwarzem Hintergrund. Das Fenster können Sie wie gewohnt mit einem Mausklick auf das x schließen. Alternativ führt die Eingabe von Strg-Q zum Schließen des Fensters.

2.1.2 Starten mit IDLE

Sollten Sie das Programm lieber aus IDLE heraus aufrufen, müssen Sie die beiden folgenden Zeilen in IDLE eintippen und unter dem oben gewählten Namen (myfirstgame.py) abspeichern:

```
import pgzrun

pgzrun.go()
```

Anschließend starten Sie das Spiel aus IDLE heraus mit F5 oder über das Menü Run > Run Module. Auch hier schließen Sie das Fenster entweder mit der Maus x oder mit Strg-Q.

2.2 Den Spielbereich festlegen

Ergänzen Sie in der Datei den folgenden Code:

```
import pgzrun

def draw():
    screen.fill("blue")

pgzrun.go()
```

(Die erste und letzte Zeile sind nur für den Start aus IDLE heraus wichtig und werden ab sofort weggelassen.)

Wenn Sie dieses Programm¹² starten, sehen Sie ein Fenster mit blauem Hintergrund. Die beiden Zeilen, die wir hinzugefügt haben, beschreiben eine Funktion. Python-Programme bestehen aus Funktionen - in der Regel aus hunderten oder tausenden von Funktionen. Unsere erste Funktion definieren wir (def) mit dem Namen draw. Die beiden runden Klammern () legen fest, dass die Funktion ohne weitere anzugebende Informationen (ohne **Parameter**) funktioniert. Die Definition des Funktionsnamens und seiner Parameter nennt man auch Funktionskopf. Die Zeichenfolge ist genau vorgegeben: zuerst kommt das sog. Schlüsselwort def, dann der Name der Funktion, dann die beiden runden Klammern und dann folgt am Ende ein Doppelpunkt. Danach geht es in der zweiten Zeile weiter, die um genau vier Leerzeichen nach rechts eingerückt ist. Die zweite Zeile screen.fill("blue") beschreibt, was die neue Funktion tut. Dies ist der sog. Funktionsrumpf. Die zweite Zeile benutzt den Bildschirm (screen) als im Speicher des Computers repräsentiertes **Objekt**. An dieses durch screen benannte Objekt wird die Nachricht fill gesendet. fill ist ihrerseits eine Funktion, die jedoch im Unterschied zu draw einen Parameter benötigt, nämlich die Farbe des Hintergrunds. Der Parameter wird in runden Klammern angefügt. Dass "blue" in Anführungszeichen eingefasst wird liegt daran, dass hier Daten in Form von Text an die Funktion fill übergeben werden sollen. Textdaten (auch **Zeichenketten** genannt, oder Englisch Strings) werden in Python immer in Anführungszeichen gesetzt. In Python kann man statt der doppelten Anführungszeichen genauso gut einfache Apostroph-Zeichen verwenden. Das folgende wäre auch ein gültiges Programm:

```
def draw():
    screen.fill('blue')
```

Es ist wichtig, den Unterschied zwischen den verschiedenen Objekt- und Funktionsnamen screen, draw, fill und den Zeichendaten "blue" zu unterscheiden. Die ersten drei sind Beispiele für Objekt- und Funktionsnamen. So wie im echten Leben z. B. der Name meinauto ein Name für ein ganz bestimmtes Auto ist, so ist der Name nicht das Auto, sondern er bezeichnet es lediglich. Es ist sogar wahrscheinlich, dass der Name meinauto heute dieses und in 20 Jahren ein anderes Auto bezeichnet, dazu später mehr. Das durch meinauto bezeichnete Auto selbst ist ein Objekt, welches verschiedene Daten enthält die seinen Zustand ausmachen. Beispielsweise enthält ein Auto Daten über seine Farbe "rot" und über seine aktuelle Geschwindigkeit 73. In Analogie könnte man im echten Leben "programmieren":

```
meinauto.lackieren("rot")
meinauto.beschleunigen(73)
```

Daten wie "rot" und 73 sind etwas völlig anderes als Objektnamen. Beides (Daten und Objektnamen) wird jedoch in einem Python-Programm benötigt. Die Daten nennt man auch **Literale**, die Objektnamen nennt man auch **Bezeichner** oder in Englisch **identifier**.

Damit die Laufzeitumgebung zwischen Objektnamen und Zeichenkettendaten (Zeichenketten-Literalen oder kurz **Strings**) unterscheiden kann, werden in Python-Programmen Strings immer in Anführungszeichen eingefasst. Zahldaten (Zahl-Literale) werden in Python-Programmen immer ohne Anführungszeichen angegeben. Die Zahl 73 ist ein Beispiel dafür. Im Folgenden sehen wir gleich ein weiteres Beispiel.

Objektnamen (Bezeichner) beginnen immer mit einem Buchstaben. Deshalb kann Python ohne weiteres zwischen Zahl-Literalen und Bezeichnern unterscheiden. Für Zahl-Literale sind Anführungszeichen weder erforderlich noch erlaubt.

¹² Die Idee für dieses erste Programm basiert auf einem Vortrag von Daniel Pope auf der EuroPython Conference 2016.Ein Video des Vortrages ist hier online: https://www.youtube.com/watch?v=Pzs-c-B3RiM.

2.3 Spielfigur

Wir verändern die Größe des Spielbereichs. Dazu fügen wir zwei neue Zeilen ein. Hier werden keine Funktionen definiert, sondern sog. **Variablen**. Was Variablen genau sind, wird später noch erklärt. Hier dienen die Zahlwerte zur Festlegung der Breite (WIDTH) und Höhe (HEIGHT) des Fensters. In Ausführung erscheint ein entsprechend kleineres Fenster:

```
WIDTH = 400
HEIGHT = 300

def draw():
    screen.fill("blue")
```

2.3 Spielfigur

Nun fügen wir eine Spielfigur hinzu. Die Spielfigur benötigt ein Bild zur Anzeige. Das Bild muss in einem von Pygame unterstützten Grafikformat vorliegen (z. B. PNG oder JPG) und im Unterverzeichnis images liegen. Wir legen gleich zwei Bilddateien¹³ ab, weil wir zwei Spielfiguren erstellen wollen:



Die eine davon verwenden wir nun für die erste Spielfigur:

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", (200, 300))

def draw():
    screen.fill("blue")
    me.draw()
```

Spielfiguren heißen in Pygame Zero **Actors**. Eine Spielfigur wird durch Ausführen der Funktion Actor mit den bereits bekannten runden Klammern dahinter erzeugt. Zwischen den runden Klammern stehen nun die benötigten Parameter. Hier sind es gleich mehrere Parameter:

- Der erste Parameter ist eine Zeichenkette "mouth", die Pygame Zero verrät, wie die für die Spielfigur zu verwendende Bilddatei heißt. Pygame Zero sucht nach dieser Datei automatisch im Unterverzeichnis images. Die Dateiendung .png fehlt hier im Programm, weil die ja sowieso eindeutig ist.
- Der zweite Parameter bezeichnet die Position der Spielfigur auf dem Bildschirm. Die Position setzt sich aus zwei Werten zusammen, die wieder in runde Klammern eingefasst sind: (200, 300). Der erste der beiden Werte bezeichnet die x-Koordinate, der zweite die y-Koordinate. So ein zusammengefasster, aus mehreren Einzelwerten bestehender Wert wird in Python **Tupel** genannt. Tupel kennt man aus der Mathematik. Ein 2D-Vektor ist ein Tupel, das aus den Einzelwerten (den Koordinaten) besteht. In Python gibt es neben den **Tupeln** noch einen anderen Typ zusammengesetzter Werte: die sog. **Listen**, die wir später kennen lernen werden.

Nachdem wir die Spielfigur erzeugt haben, müssen wir sie zeichnen. Dies tun wir nicht direkt, sondern wir sammeln alle Zeichenoperationen in der bereits vorhandenen Funktion draw. Die Zeile me.draw() führt die Zeichenoperationen für das Bild aus. Wie schon bei screen.fill("blue") können wir uns das so vorstellen, dass eine Nachricht draw an das Objekt me gesendet wird. Das Objekt beginnt daraufhin, sich selbst zu zeichnen.

_

¹³ Bildquellen: mouth.png von Aha-Soft (http://www.aha-soft.com/) via

In Abbildung 5 sehen wir das Ergebnis der Ausführung. Vermutlich ist dies nicht ganz das, was Sie sich vorgestellt haben. Wie sind die Zahlwerte (200, 300) zu verstehen? Dazu müssen wir uns ansehen, wie Koordinaten auf dem Computerbildschirm genutzt werden.



Abbildung 5: Das Bild steht unten

2.4 Koordinatensystem

Anders als im Mathematikunterricht gelernt, verläuft auf dem Computerbildschirm die y-Achse von oben nach unten. Dies hat vermutlich damit zu tun, dass ganz früher bei zeilenbasierten Textbildschirmen ganz oben auf dem Bildschirm die Zeile 1 stand und dann darunter die Zeilen mit aufsteigenden Nummern. Interpretiert man die Zeilennummern als y-Koordinaten, ergibt sich eine im Vergleich zur Mathematik verkehrten Achsrichtung (vgl. Abbildung 6). Außerdem gibt es auf dem Computerbildschirm keine negativen Koordinaten. Die obere linke Ecke besitzt die Koordinaten (0,0).

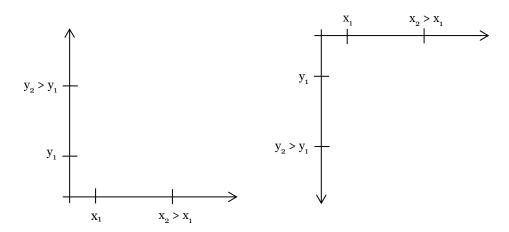


Abbildung 6: Links: Koordinatensystem eines Funktionsgraphen. Rechts: Koordinatensystem auf dem Bildschirm

Mit diesem Wissen ist zumindest schon einmal erklärt, dass die Spielfigur am unteren Rand des Spielfensters zu sehen ist, denn die Position y=300 entspricht ja gerade der Höhe des Fensters. Aber weshalb ist der Mund nur halb zu sehen? Das liegt daran, dass Pygame als **Referenzpunkt** des Bildes die Bildmitte verwendet (vgl. Abbildung 7). Der Referenzpunkt des Bildes wird auf y=300 festgelegt. Dadurch ragt die untere Hälfte des Bildes über den Fensterrand hinaus und wird abgeschnitten.

2.5 Wartbare Software



Abbildung 7: Der Referenzpunkt in der Mitte des Bildes wird mit der unteren Fensterkante zur Deckung gebracht

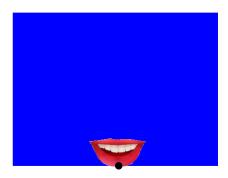


Abbildung 8: Der Referenzpunkt midbottom

Es ist möglich, einen anderen Referenzpunkt anzugeben. Wenn wir als Referenzpunkt die untere Bildkante wählen, sähe dies wie in Abbildung 8 aus. Wir ändern das Programm entsprechend:

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(200, 300))

def draw():
    screen.fill("blue")
    me.draw()
```

Hier haben wir also den zweiten Parameter, den wir an die Actor-Funktion übergeben haben, mit einem Namen ausgestattet. In Python ist es möglich, einige Parameter an Funktionen mit einem Namen auszuzeichnen. Dies ist z. B. auch dann eine gute Idee, wenn man sich nicht mehr genau an die Reihenfolge erinnern kann, in der die Parameter von der Funktion erwartet werden. Beispielsweise würde das folgende Programm, in dem die Parameter in der falschen Reihenfolge angegeben wurden, mit einem Fehler abbrechen:

```
me = Actor((200, 300), "mouth")
```

Der von der Python-Laufzeitumgebung erzeugte Fehler lautet: TypeError: join() argument must be str or bytes, not 'tuple'. Übersetzt: Der erste vom Programmierer angegebene Parameter (200,300) ist ein Tupel (englisch tuple). Die Funktion Actor erwartet jedoch eine Zeichenkette (Typ str). Der Fehlermeldung ist auch zu entnehmen, dass Parameter manchmal **Argumente** genannt werden.

Mit einer Benennung der Parameter wäre die neue Reihenfolge jedoch wiederum akzeptabel:

```
me = Actor(midbottom=(200, 300), image="mouth")
```

Diese Variante läuft fehlerfrei.

2.5 Wartbare Software

Computerprogramme unterliegen wie andere technische Produkte regelmäßigen Wartungsprozessen. Im Gegensatz zu einem Pkw, dessen Wartung die Beibehaltung des unverändert fehlerfreien Betriebs zum

2.5 Wartbare Software

Ziel hat, kann die Wartung von Software durchaus zum Ziel haben, die Funktion der Software anzupassen. Als Analogie mag gelten, dass ein Pkw mit einer neueren Abgasvorrichtung ausgestattet wird, um neuen gesetzlichen Anforderungen zu genügen. Unser Computerspiel soll vielleicht der neuen Anforderung "600 Pixel breit, 350 Pixel hoch" an die Fenstergröße genügen, weil vielleicht kleinere Fenstergrößen neuerdings augenmedizinisch als unvertretbar angesehen werden. Wir können die neue Anforderung nun wie folgt umsetzen:

```
WIDTH = 600
HEIGHT = 350

me = Actor("mouth", midbottom=(200, 300))

def draw():
    screen.fill("blue")
    me.draw()
```

Auf diese Weise erhalten wir jedoch einen logischen Programmfehler (vgl. Abbildung 9): die Spielfigur ist nicht mehr mittig und auch nicht mehr am unteren Fensterrand positioniert. Wohlgemerkt: das Spiel wird ausgeführt, d. h. wir beobachten keinen Abbruch. Aber das Programm verhält sich nicht wie erwartet. Dies nennt man einen **logischen Fehler.**

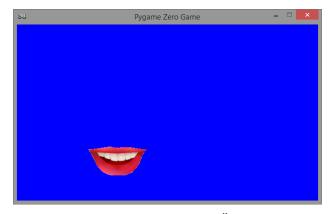


Abbildung 9: Logischer Programmfehler nach Änderung der Fenstergröße

Der logische Fehler lässt sich durch weitere Programmkorrekturen beheben:

```
me = Actor("mouth", midbottom=(300, 350))
```

Da dies jedoch mit fehleranfälligem Kopfrechnen verbunden ist und die Gefahr birgt, noch weitere Stellen im Programm zu übersehen, die ebenfalls verändert werden müssen, gehen Programmierer in der Regel auf Nummer sicher: sie lassen rechnen.

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))

def draw():
    screen.fill("blue")
    me.draw()
```

Statt konkreter Zahlwerte stehen hier nun Terme, die der Computer dann ausrechnet, wenn es nötig ist:

- WIDTH // 2 ist das Ergebnis der Division von 400 durch 2. Der doppelte Schrägstrich teilt dem Computer mit, dass das Ergebnis wieder eine ganze Zahl sein soll. Wenn die Division nicht glatt aufginge, würde der Computer als Ergebnis die nächstkleinere ganze Zahl ausrechnen.
- HEIGHT ist einfach ein Verweis auf den Zahlwert 300, der etwas weiter oben definiert wurde.

Dieses Programm ist nun **wartungsfreundlicher**. Man kann die Fenstergröße einfach in den ersten beiden Zeilen verändern und muss sich als Programmierer um den Rest des Programms nicht mehr kümmern. Der Rest des Programms ist immer korrekt, weil er sich auf die beiden Namen WIDTH und

HEIGHT bezieht, anstatt Zahlwerte zu verwenden. Wann immer möglich, sollte man versuchen, Programme auf diese oder ähnliche Weise wartbar zu gestalten.

Um die Bedeutung des doppelten Schrägstrichs besser zu verstehen, geben wir in der Shell von IDLE ein paar Beispiele ein:

Alle Ergebnisse sind, wie wir sehen können, ganzzahlig. In Python und in vielen anderen Programmiersprachen heißen ganze Zahlen auch int-Zahlen (für Englisch integer). Der //-Operator liefert, wenn er auf zwei int-Zahlen angewendet wird, als Ergebnis wieder eine int-Zahl. Man nennt eine int-Zahl auch eine Zahl vom Datentyp int.

2.6 Eine zweite Spielfigur

Wir fügen eine zweite Spielfigur hinzu:

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")

def draw():
    screen.fill("blue")
    me.draw()
    food.draw()
```

Da wir keine Position für den Pizza-Actor angegeben haben, wird als Standard-Position die linke obere Ecke des Bildes mit der linken oberen Ecke des Fensters zur Deckung gebracht (vgl. Abbildung 10).



Abbildung 10: Die Standardposition ist links oben

2.7 Variablen

Es ist an der Zeit, das Programm in seiner Struktur zu diskutieren. Die ersten vier Zeilen dienen der Definition der Fenstergröße und dem Anlegen von Spielfiguren. Diese Operationen müssen in diesem kleinen Spiel nur einmal zu Beginn ausgeführt werden. Sobald die benötigten Objekte (die Zahl-Objekte 400 und 300 sowie die Actor-Objekte me und food) angelegt wurden und mit je einem Namen versehen wurden, kann man später auf diese zugreifen. Die Namen WIDTH, HEIGHT, me und food heißen auch Variablen. Sie heißen deswegen so, weil man einen alten Namen für ein neues Objekt wiederverwenden

kann. Der Name bezeichnet also zu verschiedenen Zeiten unterschiedliche Objekte. So etwas gibt es ja auch im richtigen Leben. Der Name meinauto bezeichnet heute vermutlich ein anderes Ding als in 20 Jahren. In 20 Jahren habe ich mir möglicherweise schon mehrfach neuere Autos zugelegt. Alle diese Autos hießen immer meinauto – nur eben zu unterschiedlichen Zeiten. In einem Python-Programm kann man Objekte mit Namen versehen. Bspw. könnte der Name feind in einem ersten Spiellevel einen zahmen Gegner bezeichnen, und schon wenige Minuten später in einem höheren Spiellevel bezeichnet feind einen bis an die Zähne bewaffneten, kaum zu besiegenden Gegner. Man spricht statt vom bezeichneten Objekt auch vom referenzierten Objekt. Die Variable me referenziert ein Actor-Objekt.

Im richtigen Leben referenziert meinauto das Auto, das ich zurzeit mein Eigen nenne. Damit ich mein Auto wiederfinden kann, stellen wir uns vor, dass wir in der Variablen meinauto stets den gegenwärtigen Ort meines Autos speichern, z. B. die Geo-Koordinaten. Wenn ich mein Auto am 1. Januar 1992 um 19.15 Uhr an der Geoposition 51.502599, 7.461562 geparkt habe und ich diese Geoposition in der Variablen meinauto gespeichert habe, dann konnte ich es am 2. Januar 1993 um 8.20 Uhr wieder finden, um es z. B. neu zu lackieren (meinauto.lackieren ("rot")). Ein Jahr später am 13. Februar 1993 habe ich mein Auto an fast derselben Geoposition geparkt (51.501704, 7.461680), allerdings hatte ich mir inzwischen ein neues Auto gekauft. Gleicher Name, anderer Zeitpunkt, anderes referenziertes Objekt. Ganz genau so funktionieren Variablen in Python.

Zu verschiedenen Zeiten referenzieren Variablen unterschiedliche Objekte im Speicher des Computers. Genau genommen besitzt me als Wert die Adresse der Speicherzellen, in denen das gegenwärtig referenzierte Actor-Objekt gespeichert ist. Deswegen spricht man auch vom Wert einer Variablen und meint damit das referenzierte Objekt.

Einmal ist es mir passiert, dass ich mein neues Auto exakt an der altbekannten Position 51.502599, 7.461562 geparkt habe. Auch das kann in Python-Programmen passieren, dass eine Variable exakt dieselbe Speicheradresse speichert, die sie schon wenige Millisekunden früher gespeichert hatte, nun aber ein völlig anderes Objekt bezeichnet, welches zwischenzeitlich an dieser Speicheradresse abgelegt wurde. Als Python-Programmierer haben wir auf den Speicherort von Objekten keinen Einfluss. Das müssen wir auch gar nicht, denn im Gegensatz zum richtigen Leben, in dem wir mit der in meinauto gespeicherten Geoposition auf die Suche nach unserem Auto gehen müssen, erledigt die Python-Laufzeitumgebung die Suche nach der in me abgelegten Speicheradresse automatisch.

Manchmal werden Variablen auch grafisch wie in Abbildung 11 durch **Zeiger** skizziert. In den blauen Kästchen stellen wir uns dabei vor, dass dort die Adresse der jeweiligen Speicherzelle abgelegt wird.

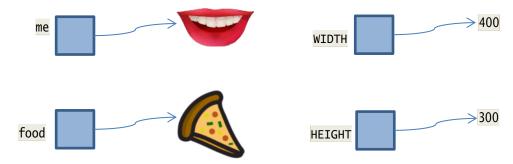


Abbildung 11:Visualisierung von Objektreferenzen durch Zeiger

2.8 Anweisungsreihenfolge

Unser Python-Programm wird von der Python-Laufzeitumgebung Zeile für Zeile von oben nach unten abgearbeitet. Man sagt auch, das Programm wird Zeile für Zeile **interpretiert**. Den Teil der Laufzeitumgebung, der dafür verantwortlich ist, das Python-Programm in Prozessoranweisungen zu übersetzen, nennt man **Interpreter**. Ja, Sie haben richtig gelesen. Der Prozessor Ihres Computers versteht gar kein Python-Programm. Prozessoren verstehen nur sog. **Maschinencode** – eine lange Folge von Zahlen. Das Python-Programm muss zuerst in Maschinencode übersetzt werden, damit der Prozessor

dieses ausführen kann. Dies leistet der Python-Interpreter. Zum Glück arbeitet der Interpreter sehr schnell und vollautomatisch, so dass wir von der Übersetzung gar nichts mitbekommen.

Da der Interpreter das Programm zeilenweise abarbeitet, ist die Anweisungsreihenfolge wichtig. Wir bauen mutwillig einen Fehler in unser Programm ein, indem wir Zeilen vertauschen. Links ist die korrekte Variante zu sehen und rechts die falsche Variante mit vertauschten Zeilen:

```
WIDTH = 400
                                                             me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
HEIGHT = 300
                                                             food = Actor("pizza")
me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
                                                             WIDTH = 400
food = Actor("pizza")
                                                             HEIGHT = 300
def draw():
                                                             def draw():
    screen.fill("blue")
                                                                 screen.fill("blue")
    me.draw()
                                                                 me.draw()
    food.draw()
                                                                 food.draw()
```

Wenn wir das rechte Programm ausführen, erhalten wir die Fehlermeldung:

```
me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
NameError: name 'WIDTH' is not defined
```

Der Interpreter versucht in der rechten Variante die erste Zeile auszuführen. Er kennt jedoch die Bedeutung von WIDTH nicht, da diese Variable erst weiter unten definiert wird. Es scheint also eine gute Idee zu sein, Objektnamen, die man als Parameter an eine Funktion übergeben will, vor dem Aufruf der Funktion zu definieren.

Eine Besonderheit sind def-Anweisungen. Der Interpreter versteht den folgenden Block zunächst als eine einzige Anweisung:

```
def draw():
    screen.fill("blue")
    me.draw()
    food.draw()
```

Der Interpreter erzeugt hieraus ein Funktionsobjekt namens draw. In dem Funktionsobjekt ist der Funktionsrumpf gespeichert (vgl. Abbildung 12). Die einzelnen Anweisungen der Funktion draw werden nicht sofort ausgeführt, sondern erst einmal in das Funktionsobjekt kopiert. Ausgeführt werden die drei Anweisungen erst später, wenn irgendjemand die draw-Funktion tatsächlich ausführt.

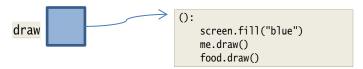


Abbildung 12:Visualisierung eines Funktionsobjekts

Dieses Verhalten des Interpreters führt dazu, dass die folgende Umordnung der Zeilen korrekt ist:

```
def draw():
    screen.fill("blue")
    me.draw()
    food.draw()

WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")
pgzrun.go()
```

Die im Funktionsrumpf von draw enthaltenen Objektreferenzen me und food führen hier nicht zu einem Fehler, obwohl die Definition von me und food ja weiter unten steht. Das liegt daran, dass me und food erst dann benutzt werden, wenn draw das erste Mal ausgeführt wird. Es ist die Zeile pgzrun.go(), die dafür sorgt, dass draw immer wieder (60 Mal pro Sekunde) ausgeführt wird. Da pgzrun.go() ganz unten

18 2.9 Gameloop

in unserem Programm steht, wurden bis dahin die beiden Objektnamen me und food definiert, so dass kein Fehler entsteht.

Die Anweisung pgzrun.go() startet übrigens den sog. **Gameloop** und endet erst dann, wenn der Benutzer das Spiel beendet hat (z. B. mit Strg-Q). Um den Gameloop geht es im folgenden Abschnitt.

2.9 Gameloop

Die drei Anweisungen, die unterhalb von draw etwas eingerückt dastehen, müssen vom Computer sehr häufig ausgeführt werden. Sehr, sehr häufig. Zigmal pro Sekunde. Das Fenster auf dem Bildschirm wird vom Computer immer wieder neu gezeichnet. Nur so ist es möglich, dass das Fenster nach z. B. kurzzeitiger Verdeckung durch ein anderes Fenster danach wieder zum Vorschein kommt. Es wird einfach vom Computer wieder neu gezeichnet, indem die Python-Laufzeitumgebung die Anweisungen der draw-Funktion erneut ausführt. Der Teil der Python-Laufzeitumgebung, der dafür sorgt, dass die draw-Funktion immer wieder neu ausgeführt wird, heißt Gameloop¹⁴.

Wir betrachten die ursprüngliche Variante unseres Programms ohne Zeilenvertauschungen und diskutieren, wie die Zusammenarbeit zwischen unserem Programm und dem Gameloop funktioniert:

```
import pgzrun

WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")

def draw():
    screen.fill("blue")
    me.draw()
    food.draw()

pgzrun.go()
```

Das Programm beginnt mit 5 Anweisungen, die unmittelbar vom Interpreter ausgeführt werden. In Abbildung 13 sind diese 5 Anweisungen als graue abgerundete Kästchen in der Spalte "myfirstgame.py" dargestellt. Das sechste Kästchen entspricht der Definition der draw-Funktion. Bisher wurde noch nicht einmal ein Fenster angezeigt und unser Programm ist schon fast zu Ende. Die letzte Anweisung ist pgzrun.go(). Diese Anweisung übergibt die Kontrolle an die rechte, mit "Pygame Zero" betitelte Spalte. Erst jetzt tritt Pygame Zero erstmalig in Aktion, indem ein Fenster angezeigt wird (oberstes abgerundetes Kästchen in der Spalte "Pygame Zero"). Danach betritt Pygame Zero den Gameloop. Der Gameloop übernimmt verschiedene Aufgaben. Zwei davon sind – wie in Abbildung 13 dargestellt – die beiden Funktionen update und draw auszuführen. Da wir keine update-Funktion geschrieben haben, wird diese übersprungen ("nichts zu tun"). Im folgenden Abschnitt werden wir sehen, wie man die update-Funktion sinnvoll für die sog. Spielmechanik nutzen kann. Als nächstes führt der Gameloop die Anweisung draw() aus. Hierbei gibt der Gameloop die Kontrolle wieder an unser Programm zurück, wie an dem nach links verlaufenden Pfeil zu erkennen ist. Jetzt werden endlich die drei Anweisungen ausgeführt, die in draw etwas eingerückt programmiert wurden. Am Ende dieser drei Anweisungen gibt unser Programm die Kontrolle wieder an Pygame Zero ab (Pfeil nach rechts). Nun ist der Gameloop wieder aktiv und übergibt an das nachfolgende Rautesymbol mit dem großen X darin. An dieser Stelle prüft Pygame Zero, ob der Benutzer eine Taste oder einen Mausklick betätigt hat, die zum Beenden des Programms führen soll. D. h. es wird konkret überprüft, ob der Benutzer Strg-Q getippt oder auf das x am oberen Fensterrand geklickt hat. Falls nicht ("Nein"), geht es zurück zum Anfang des Gameloop. Der Gameloop benachrichtigt wieder unsere draw-Funktion und diese zeichnet erneut alles auf den Bildschirm. Falls jedoch der Benutzer Strg-Q oder x betätigt hat, dann schließt Pygame Zero als nächstes das Fenster und gibt dann die Kontrolle an unser Programm ab. Nun befinden wir uns in der Ausführung direkt unterhalb

¹⁴ Der Pygame Zero Gameloop ist auch hier beschrieben: http://pygame-zero.readthedocs.io/en/stable/hooks.html

2.9 Gameloop

der Anweisung pgzrun.go(). Wenn dort jetzt noch weitere Anweisungen stünden, würden diese jetzt ausgeführt. Da aber keine weiteren Anweisungen folgen, endet das Programm.

Alle Kommandos in draw werden in Pygame Zero 60 Mal pro Sekunde ausgeführt. Deshalb ist es wichtig, dass in der draw-Funktion keine aufwändigen Berechnungen stattfinden. Außerdem soll sich die Funktion draw nur mit ihrem eigentlichen Zweck, nämlich dem Zeichnen beschäftigen. Andere Funktionen wie die Verarbeitung von Eingaben des Spielers oder die automatische Berechnung neuer Spielsituationen, gehören nicht in die draw-Funktion. Dafür gibt es die update-Funktion. Mit dem Namen update wird ausgedrückt, dass in dieser zweiten Funktion der Spielzustand auf den neuesten Stand gebracht wird. Dazu ist es u. a. erforderlich, alle durch Eingaben des Spielers entgegengenommenen Wünsche zu verarbeiten, denn diese wirken sich ja auf die Spielsituation aus. Außerdem wird in update noch die sog. Spielmechanik programmiert. Dazu gehört in klassischen "Ballerspielen", dass der Computer feststellen muss, ob sich zwei Spielfiguren (Projektil und Feindobjekt) berühren, um dann anschließend die Feind-Spielfigur durch eine Explosions-Spielfigur zu ersetzen.

Der Pygame-Gameloop ruft in schneller Folge nacheinander zuerst die update -Funktion und dann die draw-Funktion auf. Um Zeit zu sparen, kann Pygame sowohl bei der einen als auch bei der anderen Funktion Optimierungen vornehmen. Wenn Pygame etwa feststellt, dass seit dem letzten Zeichnen keine Benutzereingabe vorgenommen wurde und auch keine spielmechanischen Änderungen in der update-Funktion durchgeführt wurden, dann kann Pygame beim Neuzeichnen den ganzen Fensterbereich oder gezielte Teilbereiche unverändert lassen. Das funktioniert aber nur zuverlässig, wenn die Spielmechanik und die Eingabeverarbeitung tatsächlich in update durchgeführt wird und das Zeichnen tatsächlich in draw. Wir sollten tunlichst darauf achten, in draw keine Operationen durchzuführen, die der Spielmechanik und der Eingabeverarbeitung zuzurechnen sind und umgekehrt in update keine Zeichenoperationen durchzuführen.

Übrigens wurde in Abbildung 13 eine von Informatik-Profis und von solchen Leuten, die aufschreiben, was Informatik-Profis programmieren sollen, häufig eingesetzte Standard-Notation für Abläufe genutzt, die sog. BPMN (Business Process Modeling Notation). Mindestens genauso häufig wird eine andere Notation eingesetzt: die UML (Unified Modeling Language). Die Namen dieser Notationen haben Sie nun schon einmal gehört. In einem Informatik-Studium oder einem Studium mit Informatik-Bezug (z. B. Mediendesigninformatik, Angewandte Informatik, Wirtschaftsinformatik) lernen Sie mindestens eine, wenn nicht beide Notationen ausführlich kennen.

2.9 Gameloop

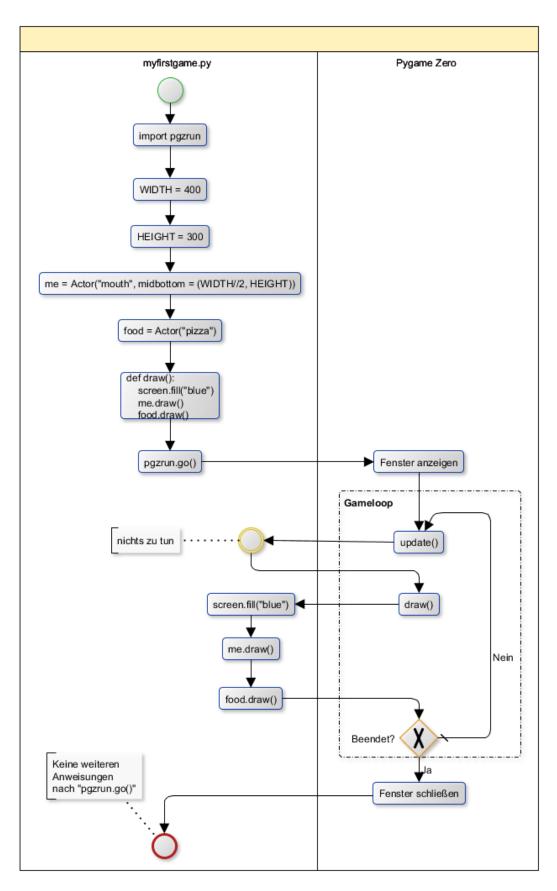


Abbildung 13: Zusammenarbeit zwischen unserem Programm und dem Gameloop¹⁵.

-

 $^{^{15}\} Diese\ Darstellung\ unterschl\"{a}gt\ zur\ vereinfachten\ Darstellung\ weitere\ vom\ Gameloop\ aufgerufene\ sog.\ Ereignis-Handler.\ Dazu\ sp\"{a}ter\ mehr.$

2.10 Spielmechanik

2.10 Spielmechanik

Wir ergänzen die bereits angekündigte update-Funktion:

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")

def draw():
    screen.fill("blue")
    me.draw()
    food.draw()

def update():
    food.y = food.y + 3
```

Die Anweisung food.y = food.y + 3 ist in zwei Schritten zu lesen. Zunächst wird der Teil rechts vom Gleichheitszeichen ausgeführt: food.y + 3. Der Name food.y bezeichnet ein Teilobjekt der Pizza-Spielfigur, nämlich die y-Koordinate der Spielfigur (vgl. Abbildung 14). Namen von Teil-Objekten (sog. **Attributen**) von Objekten werden in Python mit einem Punkt vom umfassenden Objekt getrennt. Zu Beginn referenziert food.y das Zahlobjekt 0.

Der Ausdruck food.y + 3 berechnet einen Zahlwert als Summe aus dem gegenwärtigen Wert der y-Koordinate (zu Beginn 0) und dem Wert 3. Die Summe o+3=3 ist aus Sicht der Python-Laufzeitumgebung ein brandneues Objekt, das vorher noch nicht gesehen wurde¹6. Das neue Objekt hat noch keinen Namen erhalten, sondern besitzt bisher nur einen Wert. Dieses ist in Abbildung 15 dargestellt. Der erste Teil der Anweisung food.y = ... vergibt nun einen Namen für das neue Objekt. Oder besser gesagt: dieser Teil der Anweisung verwendet den alten Namen für das neue Ergebnis. Das ist ja das Wesen von Variablen, dass sie zu unterschiedlichen Zeiten unterschiedliche Werte annehmen können. Abbildung 16 zeigt das Ergebnis. Das zuvor referenzierte Objekt 0 ist nun sozusagen "herrenlos" geworden und wird von der Laufzeitumgebung früher oder später automatisch entsorgt.



Abbildung 14: Referenziertes Teilobjekt

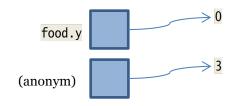


Abbildung 15: Neues Objekt 🛭 – noch ohne Namen

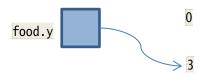


Abbildung 16: Das neue Objekt hat einen Namen bekommen

¹⁶ Das brandneue Objekt hat zwar den gleichen Zahlwert wie die im Programm stehende Zahl 3. Dennoch handelt es sich um ein brandneues Objekt. So wie zwei brandneue VW Golf, die vielleicht mit exakt derselben Ausstattung nacheinander vom Band laufen, zwar den gleichen Zustand besitzen, aber doch zwei eigene, brandneue Objekte sind.

Es ist wichtig, zu verstehen, dass durch die Anweisung <code>food.y = food.y + 3</code> sich noch nichts auf dem Bildschirm verändert hat. Lediglich die Objekte der Spielfiguren im Speicher haben sich verändert. Später, wenn wieder die <code>draw</code>-Funktion dran ist, dann wird die neue y-Koordinate aus dem Speicher ausgelesen und zum Zeichnen verwendet. D. h. nach dem nächsten Ausführen der <code>draw</code>-Funktion wird die Pizza ein bisschen tiefer als jetzt stehen. Da die <code>update</code>-Funktion 60 Mal in der Sekunde ausgeführt wird, entsteht nun wie beim Daumenkino durch eine schnelle Folge leicht voneinander abweichender Bilder der Eindruck einer flüssigen Bewegung: Die Pizza wandert zügig über den gesamten Fensterbereich an der linken Fensterkante entlang nach unten bis sie am unteren Fensterrand verschwindet.

2.11 Ereignisverarbeitung

Wenn von Benutzereingaben am Computer gesprochen wird, reden Programmierer über **Ereignisse**. Die Maus hat sich ein bisschen bewegt? Das ist ein Mausereignis. Der Benutzer hat eine Taste heruntergedrückt: ein Tastaturereignis. Er hat die Taste wieder losgelassen: noch ein Tastaturereignis.

Viele Computerspiele, und dazu zählen auch mit Pygame und Pygame Zero erstellte, definieren häufig eine Unzahl von sog. Ereignis-Behandler-Funktionen. Für jedes spezielle Ereignis eine eigene Funktion. Wir werden hier zunächst einen etwas einfacheren Weg gehen: wir programmieren in der update-Funktion einfach eine Abfrage, die prüft, ob sich auf der Tastatur gerade etwas Relevantes tut. Diese Art der Ereignisverarbeitung nennt man auch Pull-Verarbeitung (pull ist Englisch und bedeutet ziehen). Unser Computerprogramm "zieht" dann, wenn es die Information gerade gebrauchen kann, die benötigten Informationen aktiv ab. Im Gegensatz dazu verlässt sich Push-Verarbeitung darauf, dass die Pygame-Laufzeitumgebung in dem Moment, in dem ein Ereignis eintritt, eine ganz bestimmte, vom Programmierer nur für dieses Ereignis vorgesehene Funktion ausführt. Der Ablauf ist also bei der Push-Verarbeitung etwas unübersichtlicher, weil hier zusätzlich zu der vom Gameloop aufgerufenen update-Funktion immer noch weitere, vom Benutzer angestoßene Funktionsaufrufe dazwischen geraten.

In unserem kleinen Spiel wollen wir in Pull-Manier prüfen, ob der Spieler eine der beiden nach links und rechts gerichteten Pfeiltasten betätigt hat:

```
WIDTH = 400
HEIGHT = 300

me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")

def draw():
    screen.fill("blue")
    me.draw()
    food.draw()

def update():
    food.y = food.y + 3

    if keyboard.left:
        food.x = food.x - 5

    if keyboard.right:
        food.x = food.x + 5
```

Die Bedeutung der letzten vier Zeilen ist:

- Falls der Spieler die linke Pfeiltaste betätigt hat, soll die Pizza beim nächsten draw-Aufruf etwas weiter links stehen.
- Falls der Spieler die rechte Pfeiltaste betätigt hat, soll die Pizza beim nächsten draw-Aufruf etwas weiter rechts stehen.

Die so mit einer Bedingung ("falls …") versehenen Aktionen können in Python mit der sog. if-Anweisung realisiert werden. Hinter dem if steht die Bedingung, nach dem Doppelpunkt etwas eingerückt dann die nur unter dieser Bedingung auszuführende Aktion.

Die Aktionen gleichen der schon bekannten Aktion, bei der die y-Koordinate der Pizza-Spielfigur erhöht wurde. Hier wird im Gegensatz zu eben die x-Koordinate manipuliert.

Die Bedingungen verweisen auf ein von Pygame Zero bereit gestelltes Objekt keyboard. Dieses Objekt besitzt Teilobjekte (Attribute) für so ziemlich jede Taste auf der Tatstatur. Die Attribute left und right besitzen dann den Wert "Bedingung erfüllt", wenn die jeweilige Taste gedrückt wurde. Die durch keyboard left bzw. keyboard right bezeichneten Objekte sind sog. Wahrheits-Objekte, die weder Zahlen noch Buchstaben sind, sondern die nur einen von zwei Werten annehmen können: True (wahr, Bedingung erfüllt) oder False (falsch, Bedingung nicht erfüllt).

Abbildung 17 zeigt eine Spielsituation, nachdem ich die rechte Pfeiltaste eine Weile festgehalten habe und abgewartet habe, bis die Pizza nach etwa einer Sekunde nach rechts unten Richtung Mund gewandert ist. Für Die Bewegung in vertikaler Richtung war die in schneller Folge immer wieder neu ausgeführte Anweisung food.y = food.y + 3 verantwortlich. Für die Bewegung in horizontaler Richtung war die in schneller Folge immer wieder neu ausgeführte Anweisung food.x = food.x + 5 verantwortlich.



Abbildung 17: Spielsituation nach Tastatureingabe

2.12 Spielmechanik II

Wohl kein grafisches Spiel kommt ohne die Erkennung von **Kollisionen** aus. Gemeint sind zwei Spielfiguren, die sich berühren oder überlappen. Abhängig von den beteiligten Spielfiguren und der Art der Berührung / Überlappung kann ein Programmierer entscheiden, ganz bestimmte Aktionen auszulösen, die den Spielzustand verändern.

Wir wollen eine Berührung zwischen Pizza und Mund zum Anlass nehmen, einen Sound abzuspielen:

```
WIDTH = 400
HEIGHT = 300
me = Actor("mouth", midbottom=(WIDTH // 2, HEIGHT))
food = Actor("pizza")
def draw():
    screen.fill("blue")
    me.draw()
    food.draw()
def update():
    food.y = food.y + 3
    if kevboard.left:
        food.x = food.x - 5
    if keyboard.right:
        food.x = food.x + 5
    if food.colliderect(me):
        sounds.burp.play()
```

Hierzu erstellen wir eine Sound-Datei¹⁷ burp.wav im Unterverzeichnis sounds:

Die Aktion sounds.burp.play() führt zum Abspielen eines kurzen Geräuschs menschlichen Ursprungs. Bei Sounds haben sich die Pygame Zero Macher im Gegensatz zu Bildern entschieden, die Sounddatei nicht als Zeichenkette anzugeben, sondern direkt als Attribut des sounds-Objektes. Der Variablenname sounds.burp bezeichnet ein digitalisiertes Sound-Objekt im Speicher des Programms. Die Laufzeitumgebung sorgt im Hintergrund dafür, dass automatisch im Verzeichnis sounds nach einer Datei gesucht wird, die mit burp... anfängt. Hier findet die Laufzeitumgebung die Datei burp.wav, lädt diese automatisch in den Speicher und erzeugt daraus ein Objekt, welches mit dem Namen sounds.burp referenzierbar ist.

Das Geräusch wird nur dann abgespielt, wenn die davor hinter dem if-Schlüsselwort angegebene Bedingung wahr ist. Der Wahrheitswert der Bedingung wird durch einen Aufruf der Funktion colliderect ermittelt, die aus Sicht der Spielfigur food prüft, ob diese von der als Parameter angegebenen me-Spielfigur ganz oder teilweise überlappt wird.

Wenn man das Spiel ausführt, wird man feststellen, dass das Geräusch nicht nur einmal abgespielt wird, sondern sehr häufig. Und zwar genau so lange, wie sich die Pizza mit dem Mund überlappt. Erst wenn fortwährend durchgeführte, weitere Aufrufe der update-Funktion dazu geführt haben, dass die y-Koordinate der Pizza größer als 300 wird, erst dann verschwindet die Pizza aus dem Blickfeld und es besteht sicher keine Überlappung mehr, so dass die Lautsprecher verstummen.

2.13 Zusammenfassung

Das "Spiel" hat einige Schwächen, die wir hier jedoch nicht beheben wollen:

Der Sound wird zu oft abgespielt.

¹⁷ Dateiquelle: eigene Modifikation basierend auf "Super Burp Sound" (http://soundbible.com/403-Super-Burp.html) von MikeKoenig, http://soundbible.com/65-Burp-Human.html, Lizenz Attribution 3.0

2.14 Übungen

• Wenn die Pizza am unteren Bildrand verschwindet, passiert nichts mehr. Vielleicht sollte sie wieder oben auftauchen?

- Ein Punktezähler wäre schön.
- Usw.

Zusammengefasst haben wir ein erstes kleines Pygame Zero Spiel erstellt. Wir haben gelernt, was Objekte und Literale von Objektnamen unterscheidet. Außerdem haben wir die zwei grundlegenden Funktionen eines Spiels (update für die Spielmechanik und die Ereignisverarbeitung, draw für das Zeichnen) kennen gelernt. Wir kennen nun das Koordinatensystem eines Computerbildschirms und wissen, dass Positionen auf dem Bildschirm durch zusammengesetzte Werte (sog. Python-Tupel) festgelegt werden. Wir wissen, dass sich ein Programm aus mehreren Funktionen zusammen setzt und wir wissen, wie man eine Python-Funktion mit dem Schlüsselwort def definiert. Wenn einige Spielaktionen nur unter bestimmten Bedingungen ausgeführt werden sollen, kann uns die if-Anweisung gute Dienste leisten. Außerdem haben wir eine erste Rechenoperation // benutzt, wir haben vom int-Datentyp erfahren und wir haben Bild- und Sounddateien eingesetzt.

2.14 Übungen

Sie können versuchen, das Spiel zu erweitern.

- **Spielrunden:** Wenn die Pizza den Mund berührt, soll die Pizza wieder automatisch an der oberen Fensterkante auftauchen.
- **Der Schwerkraft entgegen:** Wenn der Spieler die obere Pfeiltaste betätigt, soll die Pizza nach oben wandern.

2.15 Exkurs: Bessere Kollisionserkennung

Vielleicht haben Sie schon festgestellt, dass die Kollisionserkennung nicht besonders gut funktioniert. Abbildung 18 zeigt eine Spielsituation, in der eine Kollision erkannt wird und ein Sound abgespielt wird, obwohl sich die beiden Spielfiguren noch gar nicht berühren. Die Rechtecke¹⁸ sind zur Verdeutlichung des Problems ergänzt worden. Die Überlappung der Rechtecke ist zu ungenau.

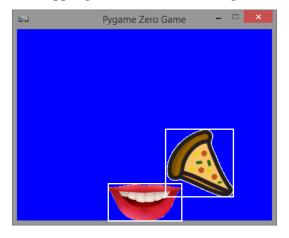


Abbildung 18: Eine Kollision, die keine ist

Fügen Sie den folgenden Programmcode an den Anfang Ihres Programms <u>unmittelbar nach der Zeile</u> import pgzrun ein:

¹⁸ Genauer "die achsenparallelen Hüllrechtecke"

```
import pygame
def overlaps(self, other):
    if self.colliderect(other):
        ox = round(self.topleft[0] - other.topleft[0])
        oy = round(self.topleft[1] - other.topleft[1])
        offset = (ox, oy)
        my_mask = pygame.mask.from_surface(self._surf)
        other_mask = pygame.mask.from_surface(other._surf)
        if other_mask.overlap(my_mask, offset) is not None:
            return True
    return False
Actor.overlaps = overlaps
```

Verwenden Sie statt colliderect nun die neue Funktion overlaps, um eine Kollision zu erkennen:

```
if food.overlaps(me):
    sounds.burp.play()
```

Den obigen Codeausschnitt müssen Sie jetzt noch nicht verstehen. Sie können hieran aber einige Eigenschaften von Python beobachten:

- In Pygame Zero können alle Pygame-Operationen eingesetzt werden. Ihnen steht also von der ersten Minute an der volle Funktionsumfang von Pygame zur Verfügung.
- Auf Elemente eines Tupels greift man durch Angabe eines Indexes in eckigen Klammern zu (bspw. self.topleft[0]).
- None ist das Literal für ein nicht existierendes Objekt.
- True und False sind die beiden Literale für Wahrheitswerte.
- if-Anweisungen kann man ineinander schachteln.
- Mit return kann eine Funktion einen berechneten Wert zurückgeben.
- Actor ist eine sog. **Klasse** für Spielfiguren, die in Pygame Zero definiert ist. Man kann die Klasse mit weiteren nützlichen Funktionen ausstatten.

3.1 Funktionsbibliothek

3 Objekte

3.1 Funktionsbibliothek

In dem folgenden Beispiel verwenden wir eine **Bibliothek**. In Computerprogrammen bezeichnet eine Bibliothek nicht etwa einen Raum mit vielen Büchern, sondern einen virtuellen Bereich mit vielen Funktionen. In Computerprogrammen wird eine Bibliothek deshalb auch **Funktionsbibliothek** genannt. So wie man im richtigen Leben immer wieder an eine Bibliothek heran tritt und um ein nützliches Buch bittet, so treten Programmierer immer mal wieder an eine Funktionsbibliothek heran, um nützliche Funktionen auszuführen. Die Bibliothek¹⁹ pgzo.py enthält einige Funktionen, die uns in den verbleibenden Beispielen dieses Buchs das Leben etwas leichter machen sollen.

Die Funktionsbibliothek kann wie folgt eingesetzt werden:

```
import pgzrun
from pgzo import *
pgzrun.go()
```

Die Bedeutung der Anweisung from pgzo import * ist, dass aus der Datei pgzo.py alle darin definierten Funktionen geladen werden. Die Dateiendung .py gibt man nicht an. Das * ist das Jokerzeichen und meint alle definierten Funktionen. Die geladenen Funktionen sind in unserem Programm dann ohne weiteres einsetzbar.

Wie bisher lassen wir in der folgenden Darstellung die erste und letzte Zeile weg.

3.2 Zwei Himmelsobjekte

Wir beginnen damit, den obigen Code in einer Datei earth.py zu speichern. Im selben Verzeichnis legen wir die Datei pgzo.py ab. Außerdem kopieren wir drei neue Bilddateien²⁰ in das images-Unterverzeichnis. Das Ergebnis sieht so aus²¹:

Wir ergänzen in der Datei earth.py wie folgt die Fenstergröße und ein erstes Spielobjekt. Beachten Sie, dass wir nun nicht die Funktion Actor aufrufen, sondern eine neue Funktion GameObj. Die neue Funktion wird uns von der Funktionsbibliothek pgzo.py zur Verfügung gestellt. Das folgende Programm zeigt, wenn es ausgeführt wird, eine Sonne auf dunkelblauem Hintergrund.

¹⁹ Die Buchstaben pgzo bedeuten **p**y**g**ame **z**ero **o**bjects.

 $^{^{\}rm 20}$ Bildquelle der in diesem Kapitel verwendeten Himmelskörper: Sonne von Joey Yakimowich-Payne

⁽https://www.iconfinder.com/icons/367526/sun_icon#size=64), Erde und Mars von Iconika (https://www.iconfinder.com/icons/1715795/earth_planet_space_icon#size=64 bzw. https://www.iconfinder.com/icons/1715796/mars_planet_space_icon#size=64). Alle drei lizensiert unter Creative Commons (Attribution 3.0 Unported), http://creativecommons.org/licenses/by/3.0/.

²¹ Wir haben uns entschiedene, sowohl das Spielprojekt aus Kapitel 2 als auch das neue Spielprojekt in einem gemeinsamen Verzeichnis zu speichern. Wenn Sie das lieber voneinander getrennt haben möchten, spricht nichts dagegen.

```
from pgzo import *
wIDTH = 300
HEIGHT = 300
sun = GameObj("sun", (WIDTH / 2, HEIGHT / 2))
def draw():
    screen.fill("darkblue")
    sun.draw()
```

Für Profis wie wir es inzwischen sind, ist es kein Problem, zwischen dem Literal "sun" und dem Objektnamen sun zu unterscheiden. Ich habe den Objektnamen nicht so gewählt, um Sie zu verwirren, sondern deshalb, weil es üblich ist, Objekte so zu benennen (bezeichnen), das man sofort weiß, um welches Objekt es sich handelt. Zur Sicherheit noch einmal zur Unterscheidung: "sun" ist ein Zeichenketten-Literal für den Dateinamen, sun ist der Bezeichner für das neue Himmelskörper-Objekt.

Beachten Sie, dass wir die Position der Sonne nun nicht durch den Operator //, sondern durch einen einfachen Schrägstrich // berechnet haben. Der Unterschied zwischen // und // ist, dass // ein exaktes Ergebnis ausrechnet, das eventuell Nachkommastellen beinhaltet. Da wir gleich vorhaben, die Himmelskörper auf eine Umlaufbahn zu schicken, und weil die dafür notwendigen Berechnungen exakt sein müssen, haben wir uns hier für eine exakte Positionierung entschieden. Intern rechnet GameObj bei der Anzeige der Sonne die Position in ganzzahlige Pixelkoordinaten um.

Probieren Sie einmal auf der IDLE-Shell folgende Eingaben aus:

```
>>> 400 / 2
200.0
>>> 5 / 2
2.5
>>> -5 / 2
-2.5
>>> -6 / 2
-3.0
>>> 5 / 3
1.6666666666666667
>>> 10 / 3
3.33333333333333333
```

Wir sehen, dass sogar die ganzzahligen Ergebnisse mit einem Dezimalpunkt und einer weiteren 0 als Nachkommastelle ausgegeben werden. Statt eines Dezimalkommas wird in Python-Programmen bei Zahl-Literalen mit Nachkommastellen immer der Dezimalpunkt verwendet. Die ausgegebenen Zahlen sind Zahlen vom Zahltyp float. Dieser Zahltyp bezeichnet sog. floating point numbers (Fließkommazahlen). Ohne in die Details zu gehen, kann man sagen, dass float-Zahlen Zahlen mit Nachkommastellen sind. Auch dann, wenn eigentlich keine Nachkommastellen benötigt werden, wird bei float-Zahlen mindestens eine Nachkommastelle angegeben. Der 7-Operator produziert als Ergebnis immer eine exakte float-Zahl.

Wir wollen die Erde hinzufügen. Wir wählen eine Position am rechten Fensterrand. Durch den Operator bezeichnen wir die Multiplikation. Der Ausdruck WIDTH * 0.8 bezeichnet 80 Prozent der Gesamtbreite:

```
from pgzo import *

width = 300
HEIGHT = 300

sun = GameObj("sun", (width / 2, HEIGHT / 2))
earth = GameObj("earth", (width * 0.8, HEIGHT/2))

def draw():
    screen.fill("darkblue")
    sun.draw()
    earth.draw()
```

Führen wir dieses Programm aus, erscheint die in Abbildung 19 dargestellte Szene.

Wir haben bisher das Wort **Anweisung** benutzt, wenn wir eine Programmzeile meinten, die "etwas tut". In Englisch nennt man Anweisungen **statements**. Beispiele für Anweisungen sind:

- sun.draw(). Diese Anweisungsart wird auch **Funktionsaufruf**, oder Englisch **function call** genannt.
- WIDTH = 300. Diese Anweisungsart wird auch Zuweisung, oder Englisch assignment genannt.

Anweisungen enthalten häufig einen oder mehrere **Ausdrücke** (Englisch **expressions**). Ein **Ausdruck** ist ein Teil eines Programms, der einen Wert besitzt.

- Manchmal ist der Wert auf den ersten Blick erkennbar. Z. B. ist in der Anweisung WIDTH = 300 die Zahl 300 ein Ausdruck mit dem Wert 300. Diese Ausdrucksart nennt man ein **Literal**.
- Manchmal muss Python den Wert eines Ausdrucks zur Laufzeit ausrechnen. Beispielsweise haben wir es in der Zeile sun = GameObj("sun", (WIDTH / 2, HEIGHT / 2)) mit vielen Ausdrücken zu tun:
 - "sun" ist ein Literal mit dem offensichtlichen Wert "sun".
 - o **WIDTH** ist ein Ausdruck mit dem von Python zur Laufzeit ermittelten Wert 150. Diese Ausdrucksart nennt man eine **Objektreferenz**.
 - o 2 ist ein Literal mit dem Wert 2.
 - o WIDTH / 2 besitzt den von Python ausgerechneten Wert 150.0. Es handelt sich um einen Ausdruck, der aus zwei **Operanden** und einem **Operator** zusammengesetzt ist.
 - o HEIGHT / 2 besitzt den von Python ausgerechneten Wert 150.0.
 - o (WIDTH / 2, HEIGHT / 2) besitzt als Wert das von Python zusammengesetzte Tupel (150.0, 150.0).
 - Und schließlich besitzt GameObj ("sun", (WIDTH / 2, HEIGHT / 2)) als Wert die Referenz auf das zur Laufzeit durch Python erzeugte Objekt. Diese Ausdrucksart nennt man einen Rückgabewert einer Funktionsausführung.

Man konnte an dem letzten komplexen Ausdruck sehen, dass Python einen Ausdruck immer "von innen nach außen" ausrechnet. So wie man bei der Mathematik-Aufgabe "14·(8+7)" zuerst die Summe von 8 und 7 ausrechnen muss und das Ergebnis danach mit 14 multiplizieren muss. Genauso muss Python bei dem komplexen Ausdruck GameObj("sun", (WIDTH / 2, HEIGHT / 2)) zuerst die "kleinen" Ausdrücke WIDTH / 2 und HEIGHT / 2 ausrechnen, dann den Tupel-Ausdruck (WIDTH / 2, HEIGHT / 2), und am Ende den vollständigen Ausdruck GameObj("sun", (WIDTH / 2, HEIGHT / 2)).

Das "Ausrechnen" eines Ausdrucks nennt man übrigens auch Auswertung eines Ausdrucks.

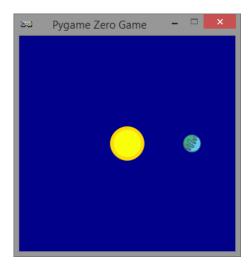


Abbildung 19: Sonne und Erde an ihrer Startposition

3.3 Objektverhalten und Objektzustand

In Computerprogrammen muss man häufig Dinge der realen Welt repräsentieren. Dazu speichert man Daten in Form von Zahlen und Texten ab. Alle Daten, die zusammen zu einem Ding gehören, möchte man in einem Computerprogramm gerne als Einheit handhaben können. In der Programmierung hat sich deshalb die sog. **objektorientierte Programmierung** als ein sehr wichtiger Programmierstil durchgesetzt. In objektorientierten Programmen werden Daten durch **Objekte** repräsentiert. In unserem kleinen Programm haben wir zwei Objekte erzeugt: die Sonne und die Erde. Die beiden Himmelskörper sind zwei Objekte mit jeweils ähnlichen aber auch unterschiedlichen Eigenschaften:

- Beide Objekte können gezeichnet werden. Beide Objekte besitzen je ein Bild und je eine Position.
- Die Bilder der beiden Objekte und auch die Positionen der beiden Objekte unterscheiden sich.

Bild und Position zusammen bezeichnen den **Zustand** unserer Objekte. Der Zustand des sun-Objekts weicht vom Zustand des earth-Objekts ab. Hinsichtlich des Bildes, das ja Teil des Zustandes ist, sehen wir das sofort. Wie sieht es mit der Position aus? Ja, wir können sehen, dass die beiden Himmelskörper an verschiedenen Stellen auf dem Fenster stehen. Wir können die intern in den Objekten gespeicherte Information über die Position sogar visualisieren. Dazu ändern wir die beiden Programmzeilen zur Erzeugung der Objekte wie folgt:

```
sun = GameObj("sun", (WIDTH / 2, HEIGHT / 2), pos_drawing_color="red")
earth= GameObj("earth", (WIDTH * 0.8, HEIGHT / 2), pos_drawing_color="red")
```

Diese Änderung bewirkt, dass der intern in den Objekten gespeicherte Positionszustand angezeigt wird (vgl. Abbildung 20). Wie wir sehen, werden auf ganze Zahlen gerundete Positionszahlen angezeigt.

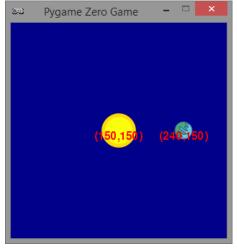


Abbildung 20: Visualisierung des in den Objekten gespeicherten Positionszustands

Die Anzeige der Position belegt, dass die beiden Objekte, so unterschiedlich sie auch aussehen mögen, dennoch ein gemeinsames **Objektverhalten** besitzen. Beide Objekte besitzen die Fähigkeit, sich selbst zusammen mit der Positionsangabe zu zeichnen. Objekte, die ein gemeinsames Objektverhalten aufweisen, gehören in der Regel zur gleichen **Klasse** von Objekten. Mit **Klasse** ist eine allgemeine Beschreibung gemeint, die auf alle Objekte der Klasse zutrifft. Die allgemeine Beschreibung für die Sonne und die Erde könnte lauten:

Sonne und Erde sind Objekte der Klasse GameObj. Alle Objekte der Klasse GameObj besitzen als Zustand eine Position und ein Bild. Alle Objekte dieser Klassen können ihr Bild zusammen mit ihrer Position zeichnen.

In Python werden Klassen durch das Schlüsselwort class programmiert. Sie können ja einmal einen Blick in die Datei pgzo.py werfen. Dort steht irgendwo die Programmzeile:

```
class GameObj(Actor):
```

die besagt, dass GameObj eine Klasse von Objekten bezeichnet, die eine Unterklasse der Klasse Actor ist. Objekte der Klasse Actor haben wir übrigens schon ganz zu Beginn angelegt, als wir eine Pizza und einen Mund erzeugt haben.

3.4 Gemeinsames Verhalten "orbit"

Wir wollen zeigen, dass die Klasse GameObj neben dem Objektverhalten "Bild und Position zeichnen" noch weiteres Verhalten für alle Objekte der Klasse GameObj definiert. Dazu legen wir als drittes Objekt den Planeten Mars an:

```
from pgzo import *

WIDTH = 300
HEIGHT = 300

sun = GameObj("sun", (WIDTH / 2, HEIGHT / 2), pos_drawing_color="red")
earth= GameObj("earth", (WIDTH * 0.8, HEIGHT / 2), pos_drawing_color="red")
mars = GameObj("mars", (WIDTH / 2, HEIGHT * 0.9), pos_drawing_color="red")

def draw():
    screen.fill("darkblue")
    sun.draw()
    earth.draw()
    mars.draw()
```

Zur Ausführung gebracht zeigt sich das Bild in Abbildung 21. Als nächstes sollen sich Erde und Mars um die Sonne bewegen. Dazu nutzen wir das Verhalten orbit, welches in der Klasse GameObj für alle Objekte dieser Klasse definiert ist. Wir ergänzen eine neue update-Funktion:

```
def update():
    earth.orbit(1)
    mars.orbit(0.53)
```

Die beiden Anweisungen führen jeweils die orbit-Funktion aus. Als Parameter übergeben wir ein Winkelinkrement in Grad. Bei jeder Ausführung von update wird der Winkel um diesen Betrag verändert. Da der Mars für eine Umrundung der Sonne fast doppelt so lange wie die Erde benötigt, erhöhen wir den Winkel des Mars entsprechend ungefähr um die Hälfte des Betrages, den wir auf den Winkel der Erde addieren.

Das Zentrum der Umrundung legen wir einmal am Anfang beim Anlegen der Objekte fest:

```
earth= GameObj("earth", (WIDTH * 0.8, HEIGHT / 2), pos_drawing_color="red", orbit_center=sun)
mars = GameObj("mars", (WIDTH / 2, HEIGHT * 0.9), pos_drawing_color="red", orbit_center=sun)
```

Das laufende Programm zeigt, wie sich Erde und Mars in unterschiedlichen Geschwindigkeiten um die Sonne bewegen. Abbildung 22 zeigt zwei Schnappschüsse.

3.5 Zustandsabfrage

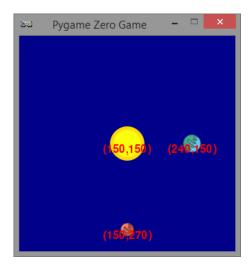


Abbildung 21: Drei Himmelskörper in der Startposition

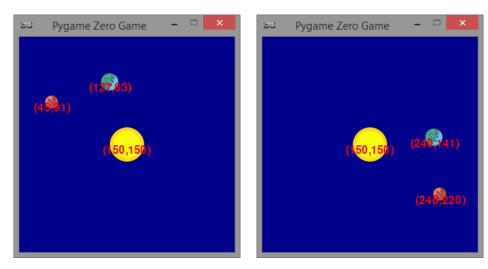


Abbildung 22: Zwei Schnappschüsse der kreisenden Planeten

3.5 Zustandsabfrage

Mit der Funktion orbit haben wir ein in der Klasse GameObj definierte Verhalten benutzt, das den Zustand des jeweiligen Objekts verändert. Die Klasse GameObj definiert außerdem eine weitere Funktion, mit der man den Zustand des Objekts abfragen kann. Zustands-Abfragefunktionen liefern demjenigen, der sie ausführt, einen Wert als Ergebnis. Man sagt auch: die Abfragefunktion liefert einen Rückgabewert.

```
def update():
    earth.orbit(1)
    mars.orbit(0.53)
    print(earth.full_orbits())
```

Die vorstehende update-Funktion, in der wir am Ende eine Anweisung hinzugefügt haben, führt zu einer sehr langen Liste von Ausgaben auf der Console bzw. in der IDLE-Shell. Zuerst werden ganz viele 0en untereinander geschrieben. Sobald die Erde eine Sonnenumrundung geschafft hat, kommen 1en, nach der zweiten Umrundung kommen 2en, usw. Der Ausdruck earth.full_orbits() führt dazu, dass die Laufzeitumgebung an das earth-Objekt eine Nachricht full_orbits sendet. Das earth-Objekt liefert als Rückgabewert die Anzahl der bisherigen Sonnenumrundungen zurück. Diese Anzahl ist ein Aspekt des earth-Zustandes. Deshalb nennt man full_orbits auch eine Zustands-Abfragefunktion.

Was machen wir mit dem erhaltenen Wert? Wir geben ihn auf der Console aus. Die **Anweisung print()** ist eine in Python eingebaute Funktion zur Ausgabe. Zwischen die Klammern schreibt man den auszugebenden Inhalt. Probieren Sie in der Python-Shell einmal die folgenden Anweisungen aus:

3.5 Zustandsabfrage 33

```
>>> print("Hallo Welt")
Hallo Welt
>>> print("Hallo", "Welt")
Hallo Welt
>>> print("Ihr IQ:", 145)
Ihr IQ: 145
>>> print(6, "*", 7, "=", 6 * 7)
6 * 7 = 42
>>> print("The answer is:", 17 + 325 / 13)
The answer is: 42.0
```

Wie wir sehen, kann man mehrere Ausgaben hintereinander in einer Zeile ausgeben, wenn man die Teile durch je ein Komma voneinander trennt. Auch verschiedene Datentypen (Strings, int-Zahlen, float-Zahlen) kann man mischen.

Wir verändern unsere update-Funktion noch einmal, um die Ausgabe verständlicher zu gestalten:

```
def update():
    earth.orbit(1)
    mars.orbit(0.53)
    print("Earth orbits:", earth.full_orbits())
    print("Mars orbits:", mars.full_orbits())
```

Wir beobachten in der Ausgabe abwechselnde Ausgaben für die Erde und den Mars, jeweils mit der zugehörigen, gegenwärtig erreichten Umrundungszahl.

Die print-Funktion ist übrigens ein hilfreiches Werkzeug, um Untersuchungen anzustellen, wenn ein Programm nicht das tut, was es soll. Wir ändern das Programm noch einmal und bauen ganz bewusst einen schwer zu findenden Fehler ein:

```
from pgzo import *
wIDTH = 300
HEIGHT = 300
sun = GameObj("sun", (WIDTH / 2, HEIGHT / 2), pos_drawing_color="red")
earth= GameObj("earth", (WIDTH * 0.8, HEIGHT * 2), pos_drawing_color="red", orbit_center=sun)
mars = GameObj("mars", (WIDTH / 2, HEIGHT * 0.9), pos_drawing_color="red", orbit_center=sun)

def draw():
    screen.fill("darkblue")
    sun.draw()
    earth.draw()
    mars.draw()

def update():
    earth.orbit(1)
    mars.orbit(0.53)
    print("Earth orbits:", earth.full_orbits())
    print("Mars orbits:", mars.full_orbits())
```

Wenn das Programm ausgeführt wird, sind nur die Sonne und der Mars zu sehen. Die Erde fehlt. Finden Sie den Fehler in dem Programm? Falls nicht, könnte es helfen, wenn das Programm die aktuelle Position der Erde fortwährend auf der Console ausgäbe. Dann wüssten wir, wo die Erde z. B. ganz am Anfang steht (wie sehen sie ja leider nicht). Wir ergänzen deshalb die folgende Anweisung ganz am Ende von update:

```
print("Earth center:", earth.center)
```

Nun starten wir das Programm und brechen es sehr schnell wieder mit Strg-Q ab. Ggf. müssen wir in der Console etwas nach oben scrollen, um das hier lesen zu können:

Da steht, dass der Erd-Mittelpunkt ganz zu Beginn eine y-Koordinate von über 600 besitzt. Und sie wird dann sogar noch größer. Da das Fenster insgesamt nur 300 x 300 Pixel groß ist, ist die Erde offenbar zu weit unten. Wir untersuchen deshalb die Anweisung zum Anlegen der Erde:

```
earth= GameObj("earth", (WIDTH * 0.8, HEIGHT * 2), pos_drawing_color= "red", orbit_center = sun)
```

Na klar, da steht HEIGHT * 2 und es müsste HEIGHT / 2 heißen. Kleine Ursache, große Wirkung.

Die print-Anweisung konnte uns bei der Fehleranalyse gute Dienste leisten.

3.6 Methode oder Funktion?

Wir haben bisher immer von **Funktionen** gesprochen, wenn wir von Programmaktionen gesprochen haben. Wir können zwei Arten von Funktionen unterschieden. Wir betrachten die beiden Beispielfunktionen print und full_orbits. Die letztgenannte Funktion wird aufgerufen, indem man direkt davor einen Objektnamen mit Punkt schreibt, etwa earth.full_orbits(). Die erstgenannte Funktion wird ohne einen solchen Objektnamen aufgerufen.

Die Funktion full_orbits ist eine Funktion, die irgendetwas mit dem Zustand eines Objekts anstellt. Sie gehört zu einem bestimmten Objekt. Funktionen, die zu einem Objekt oder eine Klasse gehören, heißen in Python und in vielen anderen objekt-orientierten Programmsprachen **Methoden**. Will man also explizit ausdrücken, dass es sich bei einer Funktion um eine zu einem Objekt gehörende Funktion handelt, nutzt man den Begriff **Methode**. Wenn es darauf nicht ankommt, ist der Oberbegriff **Funktion** auch korrekt. Die Funktion print gehört zu keinem Objekt. Deshalb nennt man print einfach **Funktion**.

3.7 Zusammenfassung

Wir haben eine Menge über Python gelernt und dabei eine kleine Planetensimulation geschrieben:

- Wir wissen, wie man eine Funktionsbibliothek importiert.
- Wir kennen den Unterschied zwischen den Operatoren / und //.
- Wir haben den Datentyp float in Aktion gesehen.
- Wir haben den Multiplikationsoperator * genutzt.
- Wir wissen, dass Ausdrücke keine Anweisungen sind und dass eine Anweisung mehrere Ausdrücke enthalten kann.
- Wir haben gelernt, dass Objekte einer Klasse angehören und dass die Klasse das Verhalten der Objekte definiert und beschreibt, was für eine Art von Zustand die Objekte besitzen.
- Wir haben konkrete Objekte mit jeweils individuellen Zuständen erzeugt.
- Wir haben eine Methode orbit zur Zustandsänderung sowie eine Methode full_orbits zur Zustandsabfrage aufgerufen.
- Wir haben die in Python eingebaute print-Funktion eingeführt.

4.1 Titel 35

4 Bühne frei für Little Crab

4.1 Titel

Wir erstellen eine neue Python-Datei namens crab.py. Wir schreiben zunächst den folgenden Code in die Datei:

```
import pgzrun
from pgzo import *

# screen size in pixels:
wIDTH = 560
HEIGHT = 460

# window title
TITLE = "Crab"

pgzrun.go()
```

Neben den bereits bekannten Variablen WIDTH und HEIGHT haben wir hier eine weitere Variable TITLE mit einem String belegt. Dies führt zu einem Fenster, das mit "Crab" betitelt ist (vgl. Abbildung 23).

Ein weiteres neues Konzept sind die Zeilen, die mit dem #-Zeichen beginnen. Diese Zeilen sind sog. Kommentarzeilen. Sie enthalten Kommentare und keine Anweisungen. Kommentare werden nicht ausgeführt, sondern dienen dazu, dass ein Programm für Menschen verständlicher wird. Man Kommentare wie oben geschehen ganz am Anfang einer eigenständigen Zeile beginnen lassen, oder man hängt einen Kommentar ans Ende einer bestehenden Zeile:

```
TITLE = "Crab" # window title
```

Alles rechts vom #-Zeichen wird nicht mehr ausgeführt. Alles links davon wird ausgeführt.

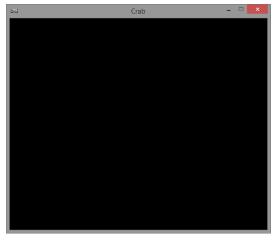


Abbildung 23: Betiteltes Fenster

4.2 Stage

In der Funktionsbibliothek pgzo.py schlummern nicht nur Funktionen, sondern ganze Klassen. Deshalb werden Bibliotheken wie diese auch **Klassenbibliotheken** genannt. Eine der Klassen in pgzo haben wir weiter oben schon kennen gelernt: GameObj. Wir werden nun eine weitere Klasse nutzen: die Klasse Stage.

Mit Stage ist eine Bühne für ein Computerspiel gemeint. Ein Computerspiel kann aus mehreren Bühnen bestehen. Stellen Sie sich z. B. vor, dass jeder Spiellevel eine eigene Bühne mit ganz besonderen Requisiten ist. Vielleicht gibt es zusätzlich noch eine Start-Bühne, auf der der Spieler Einstellungen vor dem Beginn des Spiels vornehmen kann.

In pgzo ist eine Klasse Stage für Bühnen-Objekte enthalten. Für jede Bühne in einem Computerspiel erzeugt man genau ein Bühnen-Objekt. In unserem kleinen Computerspiel, in dem wir mit einer Krabbe über den Strand laufen wollen, werden wir zwei Bühnen realisieren: eine Startbühne und eine

36 4.3 Game Assets

Strandbühne. Wir beginnen mit der Strandbühne und fügen die folgenden drei Zeilen in das oben angefangene Programm unmittelbar unter der Zeile TITLE = ... ein:

```
beach = Stage()
beach.background_image = "sand"
beach.show()
```

Wir können erkennen, dass ein Stage-Objekt erzeugt wird. Danach setzen wir das Objekt-Attribut background_image auf den Literal-Wert "sand". Das dazu gehörende Bild sand.png speichern wir im Unterordner images, wie wir das ja schon kennen. Die letzte Zeile drückt aus, dass der Vorhang zur Bühne namens beach geöffnet wird.

Wenn wir das obige Programm ausführen, sehen wir eine erste Version unserer Strandbühne (s. Abbildung 24). Man beachte: wir haben selbst keine draw-Funktion erstellt. Die draw-Funktion ist in der pgzo-Bibliothek enthalten und sie ist dort so implementiert, dass sie einfach die gerade aktive Bühne zeichnet. Wir werden auch im Folgenden erst einmal keine draw-Funktion schreiben, sondern wir werden einfach alle zu zeichnenden Spiel-Figuren zur Bühne hinzufügen ("auf der Bühne auftreten lassen"). Die Bühne ist dann selbst dafür verantwortlich, alle auf der Bühne anwesenden Spiel-Figuren zu zeichnen.

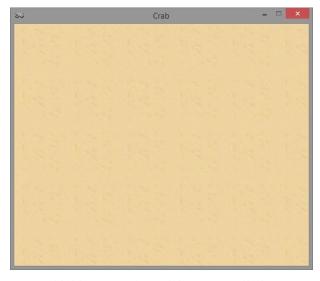


Abbildung 24: Die noch leere Strandbühne

4.3 Game Assets

In einem Computerspiel benötigen wir verschiedene Dateien, um dieses attraktiv zu gestalten. Bilddateien, Sounds und Schriftarten sind einige davon. Man nennt diese Elemente **Game Assets**.

Wir legen mehrere Assets in unserem kleinen Spielprojekt an. Dazu kopieren wir die Datei²² zachary.ttf in ein neues Unterverzeichnis fonts. Außerdem kopieren wir in das Unterverzeichnis images noch einige weitere Bilddateien²³, die wir später benötigen werden: start.png, crab.png, sand.png, worm.png, lobster.png. Schließlich befördern wir noch zwei Sound-Dateien blop.wav²⁴ und au.wav²⁵ in das Unterverzeichnis sounds. Das Ergebnis ist:

²² Quelle: http://www.fontstock.net/

²³ Bildquellen: crab.png von Freepik (http://www.freepik.com) via Flaticon (https://www.flaticon.com/free-icon/crab_311611) lizensiert unter CC 3.0 BY (http://creativecommons.org/licenses/by/3.0/), lobster.png von BomSymbols (https://creativemarket.com/BomSymbols) via https://www.iconfinder.com/icons/2427872/animal_food_lobster_restaurant_sea_food_icon#size=256, worm.png basiert auf einem Icon von Vignesh P via https://www.iconfinder.com/icons/2189567/earthworm_farming_fertilizer_gardening_growing_vermiculture_worm_icon#size=64 lizensiert unter Creative Commons (Attribution 3.0 Unported), http://creativecommons.org/licenses/by/3.0/., start.png basiert auf einem Icon von Freepik (http://www.freepik.com) via Flaticon (https://www.flaticon.com/free-icon/sun-umbrella_619003) lizensiert unter CC 3.0 BY (http://creativecommons.org/licenses/by/3.0/)

²⁴ Quelle: Mark DiAngelo (http://soundbible.com/2067-Blop.html), Lizenz Attribution 3.0

²⁵ Quelle: basierend auf https://www.youtube.com/watch?v=YVf9wo8WZzM von XRhino

4.3 Game Assets 37

5 Wir schreiben eine Klasse "Crab"

In diesem Kapitel werden wir schrittweise von der **prozeduralen Programmierung** in die **objekt-orientierte Programmierung** einführen. Am Anfang stehen Anweisungen und Funktionen als kleinste Elemente eines Programms. Am Ende werden wir eine Klasse Crab geschrieben habe. Aus dieser Klasse (man kann auch sagen aus dieser **Schablone**) werden wir anschließend viele Krabben-Objekte erzeugen können.

5.1 Die erste Spielfigur

Wir wollen eine erste Krabben-Spielfigur realisieren. Wir nutzen dazu die bereits bekannte Klasse GameObj aus der pgzo:

```
beach = Stage()
beach.background_image = "sand"
beach.show()

crab = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab.appear_on_stage(beach)
```

Anders als bisher zeichnen wir nun selbst nichts, sondern wir lassen die Spielfigur einfach auf der Bühne auftreten (appear_on_stage). Das Ergebnis ist in Abbildung 25 zu sehen.

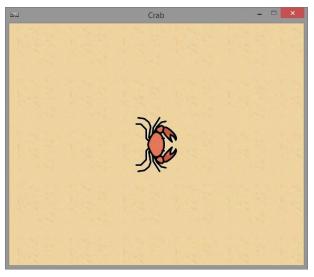


Abbildung 25: Die Krabbe erscheint auf der Strandbühne

5.2 Die Krabbe in Bewegung setzen

Um Bewegung ins Spiel zu bringen, schreiben wir eine einfache update-Funktion:

```
beach = Stage()
beach.background_image = "sand"
beach.show()

crab = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab.appear_on_stage(beach)

def update():
    crab.move()
```

Dieses Programm lässt die Krabbe langsam nach rechts aus dem Fenster heraus laufen.

Alternativ lassen wir die Krabbe um sich selbst drehen:

5.3 Fensterrand 39

```
def update():
    crab.turn()
```

Oder wir kombinieren die beiden Methoden und übergeben verschiedene Parameter. Die folgende Variante lässt die Krabbe eine kreisförmige Bewegung vollführen:

```
def update():
    crab.move(5)
    crab.turn(4)
```

Sie können mit verschiedenen Werten experimentieren und sich jeweils ansehen, zu welchen Effekten Ihr Programm führt.

5.3 Fensterrand

Im Moment kann die Krabbe einfach über den Fensterrand in den nicht sichtbaren Bereich laufen. Dann ist die Krabbe zwar nicht mehr zu sehen, aber sie ist nach wie vor "auf der Bühne". Das fühlt sich irgendwie falsch an und wir sollten das ändern. Wir wollen die Krabbe am Fensterrand "abprallen" lassen. Dazu müssen wir erst einmal eine Möglichkeit finden, heraus zu finden, ob die Krabbe in der Nähe oder gar außerhalb des Fensterrandes steht. Glücklicherweise gibt es in GameObj eine Methode, die genau diese Information liefert: can_move(distance). Diese Methode liefert einen Wahrheitswert (True oder False). Das Ergebnis ist True, wenn eine Vorwärtsbewegung um distance Pixeleinheiten das Objekt außerhalb des Bühnenrandes befördern würde. Der Rand der Bühne ist dabei gleich dem Fensterrand.

Wir geben den Wert, den diese Methode liefert, zu Testzwecken auf der Console aus:

```
def update():
    crab.move(20)
    crab.turn(1)
    print(crab.can_move(20))
```

Diese update-Funktion lässt die Krabbe in schnellen Schritten zum rechten Bildrand laufen und dann verschwinden. Auf der Console sehen wir in etwa die folgende Ausgabe:

```
====== RESTART: C:/Users/garmann/Documents/pgz/crab.py ========
True
False
False
False
False
False
```

Jede weitere Ausgabezeile lautet False, was so viel bedeutet, dass die Krabbe dauerhaft außerhalb des Bühnenbereichs verbleibt. Wir ändern die update-Funktion wie folgt:

```
def update():
    if crab.can_move():
        crab.move()
    else:
        crab.turn(25)
```

Die Funktion crab.can_move() kann auch ohne Parameter aufgerufen werden, was so viel bedeutet, dass sie true zurückgibt, wenn noch ein move-Schritt in die aktuelle Marschrichtung möglich ist, ohne den Bühnenrand zu überschreiten. Wir hätten demnach gleichbedeutend schreiben können if crab.can_move(1):.

Ein neues Konzept sehen wir mit der else-Anweisung. Man nennt den Teil ab else: auch den else-Zweig. Eine if-Anweisung wird, wie wir wissen, mit einer Bedingung eingeleitet. Die Laufzeitumgebung prüft die Bedingung. Die direkt hinter der Bedingung in der nächsten Zeile stehenden Anweisungen werden nur dann ausgeführt, wenn die Bedingung wahr ist (True). Der else-Zweig kann optional zusätzlich angegeben werden. Alles was hinter dem else-Doppelpunkt etwas eingerückt steht wird nur dann ausgeführt, wenn die Bedingung unwahr ist (False).

Nun steuert die Krabbe zunächst zielstrebig auf den rechten Fensterrand zu. Kommt sie in die Nähe, ändert sie abrupt die Richtung. Dieses Spiel wiederholt sich nun an jeder beliebigen Bühnenkante. Die Krabbe mäandert sozusagen zwischen den Bühnenrändern hin und her. Sie können das selbst ausprobieren und auch einmal den Parameterwert 25 durch andere Werte ersetzen.

5.4 Zufälliges Verhalten

Echte Krabben laufen nicht schnurgerade über den Strand. Wir wollen unsere Krabbe etwas "torkeln" lassen. Dazu führen wir zufälliges Verhalten ein. Die Krabbe soll überwiegend geradeaus laufen. Nur manchmal soll sie um einen zufälligen Drehungswinkel vom Kurs abweichen.

In Python kann eine Zufallszahl mit der folgenden Funktion erzeugt werden:

```
import random
print(random.randrange(20))
# gibt eine Zufallszahl zwischen 0 und 19 inklusive aus
```

Wir wollen die Krabbe mit einer Wahrscheinlichkeit von 10 Prozent vom Kurs abweichen lassen. Dies lässt sich realisieren, indem wir eine Zufallszahl zwischen 0 und 9 (jeweils inklusive) erzeugen und dann prüfen, ob die erzeugte Zahl o ist. Wenn die Zufallszahlen gleichverteilt sind, ist die Wahrscheinlichkeit für die 0 gerade 10%. Wir könnten alternativ jede andere feste Zahl zwischen 0 und 9 als Vergleichswert heran ziehen, Hauptsache es ist ein fester Vergleichswert.

Wie können wir Zahlen in Python vergleichen? Das geht so (neuer Code ist hervorgehoben):

```
import random
from pgzo import *
WIDTH = 560
HEIGHT = 460
TITLE = "Crab"
beach = Stage()
beach.background_image = "sand"
beach.show()
crab = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab.appear_on_stage(beach)
def update():
    if crab.can move():
        crab.move()
        crab.turn(25)
    if random.randrange(10) == 0:
        crab.turn(5)
```

Mit dem doppelten Gleichheitszeichen können wir hinter dem if-Schlüsselwort einen Vergleich durchführen. Es ist wichtig, dass man ein doppeltes Gleichheitszeichen verwendet. Das einfache Gleichheitszeichen ist ja schon vergeben für die Belegung einer Variablen mit einem Wert (**Zuweisung**).

Ein ähnliches Ergebnis hätten wir mit der folgenden Lösung erreicht, in der ein Größenvergleich "kleiner oder gleich" durchgeführt wird:

```
if random.randrange(100) <= 9:
    crab.turn(5)</pre>
```

Oder wir machen einen "echt kleiner"-Vergleich:

```
if random.randrange(100) < 10:
crab.turn(5)
```

Alle drei gezeigten Varianten führen im Endeffekt zu einem vergleichbaren Torkelverhalten.

Die von Python unterstützten Vergleichsoperatoren sind in der folgenden Tabelle dargestellt:

Operator	Bedeutung	Operator	Bedeutung
<	kleiner	>	größer
<=	kleiner oder gleich	>=	größer oder gleich
==	gleich	!= oder	ungleich

Zurzeit "torkelt" die Krabbe mit einem deutlich wahrnehmbaren Linksdrall über den Strand. Wir wollen die Krabbe gleichmäßig nach links und rechts torkeln lassen. Dazu ist die Funktion random.uniform hilfreich, bei der man die untere und obere Intervallgrenze angibt (hier inklusive):

```
if random.randrange(10) == 0:
    crab.turn(random.uniform(-5,5))
```

Die Funktion random.uniform liefert float-Werte. Im Gegensatz dazu liefert random.randrange int-Werte.

Die vorstehende Änderung führt dazu, dass die Krabbe sich in 10% aller Fälle entscheidet, vom Kurs abzuweichen. Die Höhe der Abweichung ist zufällig und liegt irgendwo zwischen -5 und +5 Grad (jeweils inklusive).

5.5 Eine zweite Krabbe und prozedurale Zerlegung

Wir wollen den Strand mit mehreren Krabben bevölkern. Eine zweite Krabbe ist leicht hergestellt, indem wir den bestehenden Code kopieren und für die zweite Krabbe etwas anpassen. Die zweite Krabbe soll einen anderen Startpunkt bekommen. Das Verhalten der zweiten Krabbe wird durch die gleichen Anweisungen beschrieben wie bei der ersten Krabbe. Durch die Verwendung von Zufallszahlen werden sich die beiden Krabben aber nicht gleichförmig verhalten. Und das ist ja gewünscht. Die Originalkrabbe bekommt nun auch einen neuen Namen crab1:

```
crab1 = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab1.appear_on_stage(beach)
crab2 = GameObj("crab", (WIDTH * 0.75, HEIGHT * 0.15))
crab2.appear_on_stage(beach)

def update():
    if crab1.can_move():
        crab1.move()
    else:
        crab1.turn(25)
    if random.randrange(10) == 0:
        crab1.turn(random.uniform(-5,5))
    if crab2.can_move():
        crab2.move()
    else:
        crab2.turn(25)
    if random.randrange(10) == 0:
        crab2.turn(25)
    if random.randrange(10) == 0:
        crab2.turn(25)
```

Man kann sich leicht vorstellen, wie die update-Funktion aussehen würde, wenn wir 100 Krabben verwenden würden. Und wenn es in diesem Spiel nicht nur Krabben, sondern auch noch andere Objekttypen geben soll, wird die update-Funktion gänzlich unübersichtlich. Man kann sich vorstellen,

²⁶ Der Operator != wird bevorzugt eingesetzt.

dass man bei den zig Anweisungen, die man bräuchte, schnell durcheinander kommen kann. War diese Anweisung nun noch für die Krabbe 12 oder für den Wurm 21 wichtig?

Statt alles in eine update-Funktion zu programmieren, schreiben wir lieber eine separate Funktion für jedes Spielobjekt. Die Funktion für die Krabbe schreiben wir in eine separate Funktion:

```
def act(self):
    if self.can_move():
        self.move()
    else:
        self.turn(25)
    if random.randrange(10) == 0:
        self.turn(random.uniform(-5,5))
```

Der Name act soll daran erinnern, dass sich die Krabben auf einer Bühne befinden und sich wie Schauspieler über die Bühne bewegen. Die act-Funktion bekommt einen Parameter namens self. self referenziert ein Krabben-Objekt, welches per self.move() und self.turn(25) in Bewegung versetzt wird. Welches Objekt der Objektname self nun genau referenziert, weiß die Funktion act nicht. Da sich das Verhalten aber sowieso für die beiden Krabben nicht unterscheidet, ist das auch unerheblich. In der update-Funktion schließlich rufen wir act zwei Mal auf und übergeben nacheinander die erste Krabbe und dann die zweite Krabbe:

```
def update():
    act(crab1)
    act(crab2)
```

So sieht die update-Funktion schon wieder viel übersichtlicher aus. Was wir hier gemacht haben, ist eine Zerlegung eines großen Problems in mehrere Teilprobleme. Für jedes Teilproblem haben wir eine Funktion geschrieben. Das Problem, eine einzelne Krabbe zu bewegen, ist in der Funktion act gelöst worden. Das Problem, zwei Krabben zu steuern, ist in update gelöst. Jede der beiden Funktionen hat eine klar umrissene Teilaufgabe zugewiesen bekommen. Man nennt diese Technik der Programmierung prozedurale Zerlegung. Wann immer möglich, sollte man Programme in mehrere Prozeduren (in Python Funktionen genannt) zerlegen.

5.6 Verhaltensparameter

Wir wollen Krabben mit einer individuellen Geschwindigkeit ausstatten. Sportliche Krabben fegen über den Strand, langsamere Krabben müssen ggf. um ihr Leben fürchten, weil sie zu einer leichten Beute werden.

Wir führen für jede der beiden Krabben je eine Variable speed... ein. Die Variable muss, da sie das move-Verhalten der Krabbe beeinflusst, als zusätzlicher Parameter an die act-Funktion übergeben werden:

```
crab1 = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
speed1 = 5
crab1.appear_on_stage(beach)
crab2 = GameObj("crab", (WIDTH * 0.75, HEIGHT * 0.15))
speed2 = 1
crab2.appear_on_stage(beach)

def act(self, speed):
    if self.can_move(speed)
    else:
        self.move(speed)
    else:
        self.turn(25)
    if random.randrange(10) == 0:
        self.turn(random.uniform(-5,5))

def update():
    act(crab1, speed1)
    act(crab2, speed2)
```

Wir überlegen, wie sich unser Programm entwickeln würde, wenn wir weitere Variablen zur Beschreibung individuellen Krabbenverhaltens einführen würden. Mit einem "Trunkenheitswert" zwischen 0 und 10

ließe sich die Wahrscheinlichkeit von Kursabweichungen steuern, mit einem "Sprunghaftigkeitswert" die Stärke der Kursabweichungen:

```
crab1 = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
speed1 = 5
drunkenness1
jumpiness1 = 15
crab1.appear_on_stage(beach)
crab2 = GameObj("crab", (WIDTH * 0.75, HEIGHT * 0.15))
speed2 = 1
drunkenness2 = 1
jumpiness2 = 25
crab2.appear_on_stage(beach)
def act(self, speed, drunkenness, jumpiness):
    if self.can_move(speed):
        self.move(speed)
    else:
        self.turn(25)
    if random.randrange(10) < drunkenness:</pre>
        self.turn(random.uniform(-jumpiness, jumpiness))
def update():
    act(crab1, speed1, drunkenness1, jumpiness1)
    act(crab2, speed2, drunkenness2, jumpiness2)
```

Die Parameterliste für act wird, wie wir sehen, länger und länger. Zudem ist der Zusammenhang von Daten, die zusammen gehören, in dieser Version nur durch die an die Variablennamen angehängte Ziffer ersichtlich. Wie wäre es, wenn wir die Verhaltensparameter (speed, drunkenness, jumpiness) direkt in den Krabbenobjekten speichern könnten? Dann würde sich die Parameterliste wieder auf ein einziges zu übergebendes Krabben-Objekt verkürzen. Dies ist in Python problemlos möglich. Wir können einfach zusätzliche Attribute in crabl bzw. crabl durch Angabe eines Punktes und dann des Attributnamens speichern. In der act-Funktion kann man die Attributwerte dann verwenden:

```
crab1 = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab1.speed = 5
crab1.drunkenness = 7
crab1.jumpiness = 15
crab1.appear_on_stage(beach)
crab2 = GameObj("crab", (WIDTH * 0.75, HEIGHT * 0.15))
crab2.speed = 1
crab2.drunkenness = 1
crab2.jumpiness = 25
crab2.appear_on_stage(beach)
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
        self.turn(25)
    if random.randrange(10) < self.drunkenness:</pre>
        self.turn(random.uniform(-<u>self.jumpiness</u>, <u>self.jumpiness</u>))
def update():
    act(crab1)
    act(crab2)
```

5.7 Initialisierungsfunktion

Zur Initialisierung einer Krabbe müssen wir im Moment jeweils vier Zeilen Programmcode schreiben:

```
crab1 = GameObj("crab", (WIDTH / 2, HEIGHT / 2))
crab1.speed = 5
crab1.drunkenness = 7
crab1.jumpiness = 15
```

Für jede Krabbe ist der Code sehr ähnlich. Wir wollen auch hier das Prinzip der prozeduralen Zerlegung anwenden, und eine Initialisierungsfunktion programmieren.

```
def init(self, pos, speed, drunkenness, jumpiness):
    self.image = "crab"
    self.pos = pos
    self.speed = speed
    self.drunkenness = drunkenness
    self.jumpiness = jumpiness
```

Die neue Funktion bekommt als ersten Parameter das zu initialisierende Objekt. In Python hat sich für Funktionen, die einem Objekt zugeordnet sind (also **Methoden**), der Name self eingebürgert. Danach folgen viele weitere Parameter, die zur Initialisierung heran gezogen werden. Die Festlegung der Attributwerte des neuen Objekts erfolgt unter Rückgriff auf die aktuellen erhaltenen Parameterwerte. Der Attributwert self.image ist sowieso für alle Krabben gleich und wird immer auf denselben Wert crab festgelegt.

Wir führen die neuen Funktionen nun wie folgt aus:

```
crab1 = GameObj()
crab2 = GameObj()
init(crab1, (WIDTH / 2, HEIGHT / 2), 5, 7, 15)
init(crab2, (WIDTH * 0.75, HEIGHT * 0.15), 1, 1, 25)
crab1.appear_on_stage(beach)
crab2.appear_on_stage(beach)

def update():
    act(crab1)
    act(crab2)
```

Mit der neuen init-Funktion könnten wir nun auf sehr übersichtliche Weise viele Krabben erzeugen und mit jeweils individuellen Objektzuständen initialisieren.

5.8 Verhaltensfunktion im Krabbenobjekt speichern

Wir stellen uns vor, dass neben Krabben viele weitere Tierarten den Strand bevölkern. Für jede Tierart müssen wir ein anderes Verhalten implementieren. Jede Tierart bekommt eigene init- und act-Funktionen. Damit wir die Funktionen unterscheiden können, müssten wir sie entsprechend benennen, z. B. crab_init, worm_act, usw.:

```
def crab_init(self, pos, speed, drunkenness, jumpiness):
def crab_act(self):
def worm_ init(self, pos, ...):
def worm_act(self):
def lobster_ init(self, pos, ...):
def lobster_act(self):
crab = GameObj()
worm = GameObj()
lobster = GameObj()
crab_init(crab, (WIDTH / 2, HEIGHT / 2), 5, 7, 15)
worm_init(worm, ...)
lobster_init(lobster, ...)
crab.appear_on_stage(beach)
worm.appear_on_stage(beach)
lobster.appear_on_stage(beach)
def update():
   crab_act(crab)
    worm_act(worm)
    lobster_act(lobster)
```

Die Liste dieser Funktionen kann ganz schön lang werden. Außerdem gäbe es auch hier wieder das Problem, dass der Zusammenhang zwischen den Funktionen crab_init und crab_act nicht explizit angegeben ist, sondern nur über Namensgleichheit herausgefunden werden kann. Und: was würde wohl passieren, wenn wir versehentlich ein Wurm-Objekt an die crab_act-Funktion übergeben: crab_act(worm)?

Wir wollen dies in wenigen Minuten verbessern, indem wir neben den Verhaltensparametern auch noch das Verhalten selbst in den Spielobjekten speichern. Dann ist ein versehentliches Übergeben eines falschen Objekts ausgeschlossen.

Betrachten wir jedoch vorher noch einmal im Zusammenhang, was wir bisher haben:

```
def init(self, pos, speed, drunkenness, jumpiness):
    self.image = "crab"
    self.pos = pos
    self.speed = speed
    self.drunkenness = drunkenness
    self.jumpiness = jumpiness
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
       self.turn(25)
    if random.randrange(10) < self.drunkenness:</pre>
        self.turn(random.uniform(-self.jumpiness, self.jumpiness))
crab = GameObi()
init(crab, (WIDTH / 2, HEIGHT / 2), 5, 7, 15)
crab.appear_on_stage(beach)
def update():
    act(crab)
```

Wir wollen die beiden Zeilen zum Erzeugen eines Spielobjekts und zur anschließenden Initialisierung zusammenfassen. Statt

```
crab = GameObj()
init(crab, (WIDTH / 2, HEIGHT / 2), 5, 7, 15)
```

wollen wir schreiben:

```
crab = Crab((WIDTH / 2, HEIGHT / 2), 5, 7, 15)
```

Dazu schreiben wir eine Funktion namens Crab:27 28

```
def Crab(pos, speed, drunkenness, jumpiness):
    self = GameObj()

def init():
        self.image = "crab"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness

init()
    return self
```

Sie sehen, dass wir die init-Funktion verschoben haben, und zwar in die Crab-Funktion hinein. Der Vorteil ist, dass diese Funktion keine Namenskollisionen mit Würmern und Hummern mehr befürchten muss, denn die Funktion init ist nur innerhalb der Crab-Funktion bekannt. Man sagt auch, die Funktion init ist nur lokal in der Crab-Funktion bekannt. Wir führen init direkt auf einem neu erzeugten Objekt namens self aus. Das solchermaßen initialisierte Objekt geben wir anschließend mit einer neuen Anweisung return als Ergebnis des Crab-Funktion (als Rückgabewert) zurück.

Wir können die neue Crab-Funktion wie folgt nutzen:

```
crab = Crab((WIDTH / 2, HEIGHT / 2), 5, 7, 15)
```

Wenden wir uns nun der act-Funktion zu. Auch diese verschieben wir in die Crab-Funktion hinein:

```
def Crab(pos, speed, drunkenness, jumpiness):
    self = GameObj()

    def act():
        if self.can_move(self.speed):
            self.move(self.speed)
        else:
            self.turn(25)
        if random.randrange(10) < self.drunkenness:
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))

def init():
        self.act = act
        self.image = "crab"
            ... Rest wie oben ...

init()
    return self</pre>
```

In Python sind Funktionen auch einfach nur Objekte. Innerhalb der Crab-Funktion wird mit def act():... ein neues Funktionsobjekt namens act erzeugt. Das neue Funktionsobjekt speichern wir in der ersten Anweisung in init direkt im Krabbenobjekt durch die Zuweisung self.act = act. Stellen Sie sich das Objekt self als einen Container für ein Sammelsurium verschiedenster Dinge vor. In self findet man Koordinaten, einen Bilddateinamen, einen Geschwindigkeitswert, einen Trunkenheitswert, einen Sprunghaftigkeitswert, ja und nun neuerdings auch eine Verhaltensfunktion. Auf die Verhaltensfunktion

²⁷ Beachten Sie das große C am Anfang des neuen Funktionsnamens. Da Python Groß- und Kleinschreibung bei Objektnamen und Funktionsnamen unterscheidet, ist keine Verwechslung mit dem Objekt crab (klein geschriebenes c) möglich.

²⁸ Mit der Definition der Funktion Crab gehen wir einen kleinen Umweg. Erfahrene Python-Programmierer würden stattdessen direkt eine sog. Crab-Klasse programmieren. Weil wir noch nicht genau wissen, wie das geht, versuchen wir es erst mit dem Funktionskonzept. Wir werden sehen, dass wir damit schon erstaunlich weit kommen. Der Schritt zu dem Klassenkonzept ist dann am Ende nur noch ein ganz kleiner. S. auch Abschnitt 5.10.

greift man wie gewohnt mit der Punkt-Notation zu: crab.act. Will man die Verhaltensfunktion ausführen, hängt man hinten noch die beiden runden Klammern an: crab.act(). Will man dies nicht, lässt man die runden Klammern weg. Die init-Funktion will act nicht ausführen, sondern in self speichern. Deshalb verwendet sie keine runden Klammern am Ende.

Der Vorteil ist nun, dass eine Verwechselung, welche act-Funktion zu welchem Spielobjekt gehört, eigentlich ausgeschlossen ist:

5.9 Zerlegung durch Klassen

Wir haben im vorangegangenen Abschnitt die Initialisierung eines Spielobjekts und dessen Verhalten in einem Objekt gebündelt. Dann haben wir mehrere Objekte angelegt und agieren lassen. Alle diese Objekte haben eine gleiche Datenstruktur bestehend aus Position, Bilddateiname, Geschwindigkeit, Trunkenheit und Sprunghaftigkeit. Außerdem haben alle diese Objekte das gleiche Verhalten in Gestalt einer identischen Funktion act. Wenn mehrere Objekte gleiches Verhalten aufweisen, spricht man von einer Klasse von Objekten. In Python gibt es zum Definieren einer Klasse das Schlüsselwort class:

```
class Crab(GameObj):
   def __init__(self, pos, speed, drunkenness, jumpiness):
    self.image = "crab"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
crab1 = Crab((WIDTH / 2, HEIGHT / 2), 5, 7, 15)
crab1.appear_on_stage(beach)
crab2 = Crab((WIDTH * 0.75, HEIGHT * 0.15), 1, 1, 25)
crab2.appear_on_stage(beach)
def update():
    crab1.act()
    crab2.act()
```

Wir arbeiten uns gemeinsam durch den vorstehenden Code. In der ersten Zeile wird die Klasse Crab definiert²⁹. In Klammern dahinter geben wir an, von welchem Typ die Objekte der Klasse Crab sein sollen, nämlich GameObj. Die Angabe des Typs in Klammern entspricht in etwa der ersten Anweisung unserer alten Crab-Funktion in Abschnitt 5.8. Dort hatten wir ja geschrieben self = GameObj(...).

Als nächstes sehen wir eine Funktion mit dem Namen __init__. In Python werden Initialisierungsfunktionen von Klassen nicht init genannt, sondern __init__ mit zwei führenden und zwei angehängten Unterstrichen _. Außerdem bekommt __init__ als allerersten Parameter immer das

²⁹ Hier beenden wir also unseren obigen Umweg und tun das, was erfahrene Python-Programmierer direkt gemacht hätten. Die Crab-Funktion aus Abschnitt 5.8 war nur ein Zwischenschritt, den wir nun verwerfen. Überleben wird nur die nun neu zu schreibende Crab-Klasse.

Objekt self übergeben, das initialisiert werden soll. Alle Parameter der ursprünglichen Crab-Funktion aus Abschnitt 5.8 sind nun zur __init__-Funktion gewandert, denn dort werden sie ja eigentlich gebraucht. Wenn Sie genau hinsehen, ist der Funktionsrumpf von __init__ fast identisch mit der init-Funktion in Abschnitt 5.8. Lediglich die Anweisung self.act = act fehlt. Diese Anweisung müssen wir jetzt nicht mehr explizit hinschreiben. Python weiß, dass es sich um ein Objekt einer Klasse handelt. Alle Funktionen, die in der Klasse definiert sind, werden automatisch in jedem Objekt der Klasse als Attribut abgelegt. Das müssen wir jetzt nicht mehr selbst erledigen.

Auch die act-Funktion ist eine gute Bekannte aus Abschnitt 5.8. Einziger Unterschied: der erste Parameter heißt self.

In der <u>Crab-Funktion</u> aus Abschnitt 5.8 hatten wir ganz zum Schluss die <u>init-Funktion</u> explizit aufgerufen und dann das erzeugte und initialisierte Objekt mit <u>return</u> zurückgegeben. Diese beiden Anweisungen entfallen nun bei der neuen <u>class-Lösung</u>. Python ruft automatisch für jedes neu erzeugte Objekt die <u>__init__</u>-Funktion auf und liefert dann das neu erzeugte und frisch initialisierte Objekt als Rückgabewert zurück.

Wir lehnen uns zurück und betrachten unser Werk. Es war ein langer Weg:

- Zuerst hatten wir einzelne Anweisungen in eine update-Funktion geschrieben, die für alle Spielfiguren zuständig war.
- Dann hatten wir die update-Funktion in mehrere Funktionen zerlegt. Für jede Spielfigur schrieben wir eine eigene act-Funktion.
- Nachdem wir immer mehr Parameter (Geschwindigkeit, Trunkenheitswert, Sprunghaftigkeitswert) hinzufügten, wurde die Initialisierung einer Spielfigur so komplex, dass wir für jede Spielfigur eine eigene init-Funktion schrieben.
- Schließlich drückten wir den Zusammenhang zwischen den Daten einer Spielfigur und seinen zugehörigen act- und init-Funktionen durch einen Klassenrahmen aus. Der Klassenrahmen verhindert, dass wir irrtümlich z. B. die mit der Wurm-Initialisierungsmethode erzeugte Spielfigur an die Krabben-act-Methode übergeben.

Wir beschreiben noch einmal allgemein, was man in Python unter einer Klasse versteht. Eine Klasse in Python beginnt mit einem Klassennamen und dahinter in runden Klammern dem Typ der Klasse (man kann auch sagen, dem **Basistyp**) gefolgt von einem Doppelpunkt. Danach folgt eine __init__-Funktion, die die Aufgabe hat, ein Objekt der Klasse zu initialisieren. Das zu initialisierende Objekt wird als erster Parameter self übergeben. Die Initialisierung in __init__ betrifft die <u>Daten</u> des Objekts. Das <u>Verhalten</u> des Objekts folgt dann nach der __init__-Funktion durch weitere, einfach untereinander programmierte Funktionen. Jede dieser weiteren Funktionen bekommt als ersten Parameter das Objekt, dessen Verhalten programmiert werden soll, als Parameter self übergeben. Alle Funktionen einer Klasse heißen, wie wir schon in Abschnitt 3.6 gelernt hatten, **Methoden**. Will man eine Klasse nutzen und Objekte der Klasse erzeugen, schreibt man den Klassennamen und dahinter in runde Klammern alle Parameter der <u>__init__</u>-Methode auf. Den ersten self-Parameter muss man dabei überspringen. Für ein auf diese Weise erzeugtes Objekt kann man dessen Verhalten ausführen. Dazu notiert man den Objektnamen gefolgt von einem Punkt gefolgt vom Methodennamen und dahinter zwei runde Klammern. Zwischen die runden Klammern schreibt man eventuelle Parameter, die der Verhaltens-Methode übergeben werden. Auch hierbei überspringt man den self-Parameter. Falls die Methode außer self keine weiteren Parameter hat, bleibt beim Aufruf der Methode der Platz zwischen den runden Klammern leer.

5.10 Randbemerkung: Didaktik meets Technik

Das, was wir in Abschnitt 5.8 gemacht haben, nämlich Funktionen als Attribute in Objekten zu speichern, haben wir in Abschnitt 5.9 zum Klassenkonzept weiter entwickelt. Aus didaktischer Sicht ist dieser Übergang nahtlos. Aus Python-Sicht sind die in Objektattributen gespeicherten Funktionsobjekte von Abschnitt 5.8 etwas vollkommen anderes als die Klassenmethoden von Abschnitt 5.9. Erstens erzeugen die beiden Ansätze Funktionen in verschiedenen **Namensräumen** (Objekt vs. Klasse). Zweitens sind die Funktionen von Abschnitt 5.8 **unbound functions** und die von Abschnitt 5.9 **bound methods**. Den

zweiten Punkt kann man zwar durch eine kompliziertere Syntax angleichen³⁰, das hilft aber aus didaktischer Sicht kaum weiter, eher im Gegenteil. Das Vorgehen in Abschnitt 5.8 nennt man auch **monkey patching³¹**. Ein solches Vorgehen wird in der Praxis nur unter ganz bestimmten Umständen als akzeptabel angesehen, in die Anfänger- und auch Fortgeschrittenen-Spielprojekte in den seltensten Fällen geraten.

Abschnitt 5.8 ist lediglich eine "Übergangsstation". Die Intention und Argumentation von Abschnitt 5.8 sollten Sie verinnerlichen. Die vorläufige Lösung in Form des Programmcodes aus Abschnitt 5.8 werden Sie über kurz oder lang wieder vergessen. Und das ist gut so. Sie werden sich in neuen Spielprojekten an der Lösung aus Abschnitt 5.9 und den kommenden Kapiteln orientieren. In diesem einführenden Buch werden wir daher den o. g. technischen Unterschied nicht vertiefen.

³º Ein in das Thema "Patching" einführender Blog-Artikel mag hier interessant sein: https://tryolabs.com/blog/2013/07/05/run-time-method-patching-python/

³¹ https://en.wikipedia.org/wiki/Monkey_patch

6 Krabbeninvasion

6.1 Zufällige Startposition

Bisher haben wir jedes Spielobjekt mit genau definierten Startwerten erzeugt. Wir wollen die Krabben zufällig positionieren und auch sonstige Eigenschaften der Krabben zufällig wählen. Die folgende Hilfs-Funktion erzeugt eine solch zufällige Krabbe:

Konzeptionell ist hier wenig Neues zu lernen. Für jeden der benötigten Parameter der in der Klasse definierten <u>__init__</u>-Funktion übergeben wir einen sorgfältig gewählten Zufallswert. Beachten Sie, dass man den ersten Parameter self der <u>__init__</u>-Funktion nicht angeben darf. Dieser wird automatisch von Python zur Laufzeit ergänzt.

6.2 Krabbeninvasion

Wir wollen nun "absahnen". Dadurch, dass wir eine Crab-Klasse realisiert haben, ist es nun leicht, hunderte von Objekten dieser Klasse zu erzeugen und zu beherrschen.

In Python gibt es die Möglichkeit, mehrere Objekte in einer **Liste** zu speichern. Wie auf einer Einkaufsliste, deren Zeilen man vielleicht durchnummeriert hat, kann man nachher einzelne Elemente der Liste wiederfinden, indem man sich einfach nur die Zeilennummer merkt. Eine Liste wird in Python mit eckigen Klammern erzeugt:

```
all_crabs = [] # leere Liste erzeugen
for i in range(20):
    c = create_random_crab()
    all_crabs.append(c) # Einen Eintrag ans Ende der Liste anfügen
```

Der zu Beginn als leere Liste angelegte all_crabs-Name wird in der letzten Zeile verwendet, um mit append einen Eintrag ans Ende der Liste anzufügen. Der angefügte Eintrag c wird direkt davor durch Ausführung unserer eben geschriebenen Funktion create_random_crab erzeugt.

Was bedeutet die Zeile mit dem for? Diese Zeile leitet eine sog. Schleife ein. Eine Schleife umrahmt eine Folge von (eingerückt eingetippten) Anweisungen, die mehrfach ausgeführt werden sollen. In unserem Beispiel werden die dritte und die vierte Zeile insgesamt 20 Mal ausgeführt. Und zwar zuerst die dritte Zeile, dann die vierte Zeile, dann erneut die dritte Zeile, dann wieder die vierte Zeile, usw. Da in der dritten Zeile immer wieder ein neues Krabben-Objekt erzeugt wird, wird in der vierten Zeile 20 Mal nacheinander ein neues Krabbenobjekt an die Liste all_crabs angefügt. Am Ende besitzen wir eine Liste mit 20 zufällig erzeugten Krabben.

Die Schreibweise for i in range(20): ist etwas gewöhnungsbedürftig. Man liest das etwa wie "für einen Zähler namens i, der nacheinander mit den Werten o, 1, 2, ..., 19 belegt wird". Wir können ja einmal versuchen, den Wert des Zählers i auszugeben:

```
all_crabs = []
for i in range(20):
    print("Ich lege die ", i, "-te Krabbe an")
    c = create_random_crab()
    all_crabs.append(c)
```

Wenn wir dieses Programm ausführen, beobachten wir die folgende Consolenausgabe:

6.2 Krabbeninvasion 51

```
== RESTART: C:\Users\garmann\Documents\pgz\crab.py ========
Ich lege die 0 -te Krabbe an
Ich lege die 1 -te Krabbe an
Ich lege die 2 -te Krabbe an
Ich lege die 3 -te Krabbe an
Ich lege die 4 -te Krabbe an
Ich lege die 5 -te Krabbe an
Ich lege die 6 -te Krabbe an
Ich lege die 7 -te Krabbe an
Ich lege die 8 -te Krabbe an
Ich lege die 9 -te Krabbe an
Ich lege die 10 -te Krabbe an
Ich lege die 11 -te Krabbe an
Ich lege die 12 -te Krabbe an
Ich lege die 13 -te Krabbe an
Ich lege die 14 -te Krabbe an
Ich lege die 15 -te Krabbe an
Ich lege die 16 -te Krabbe an
Ich lege die 17 -te Krabbe an
Ich lege die 18 -te Krabbe an
Ich lege die 19 -te Krabbe an
```

Wie Sie sehen, fangen Programmierer beim Zählen immer mit 0 an. Wenn Sie lieber bei 1 starten wollen, programmieren Sie stattdessen:

```
all_crabs = []
for i in range(1, 21):
    print("Ich lege die ", i, "-te Krabbe an")
    c = create_random_crab()
    all_crabs.append(c)
```

Die Ausgabe beginnt dann bei 1 und endet mit 20 -te Krabbe. Die bei range (1, 21) angegebene 21 wird also nicht mehr mitgezählt.

Die vielen zufälligen Krabben wollen wir nun natürlich auch sehen. Deshalb müssen alle Krabben die Bühne betreten:

```
all_crabs = []
for i in range(20):
    c = create_random_crab()
    all_crabs.append(c)

for c in all_crabs:
    c.appear_on_stage(beach)
```

Hier sehen wir eine etwas andere Variante der for-Schleife. Die Durchläufe werden nicht durch einen Zahlbereich vorgegeben (range), sondern durch die Liste all_crabs. Die Anweisung for c in all_crabs: bedeutet: "für jedes Element in der Liste all_crabs tue das folgende:". Das folgende ist eine Anweisung c.appear_on_stage(beach), in der die Variable c genutzt wird. c nimmt nacheinander verschiedene Werte an. Beim ersten Durchlauf der Schleife referenziert c das o-te Krabbenobjekt der Liste, beim zweiten Durchlauf der Schleife referenziert c das 1-te Krabbenobjekt der Liste, usw.

Das Ergebnis ist in Abbildung 26 dargestellt.

52 6.3 "... und Action bitte"

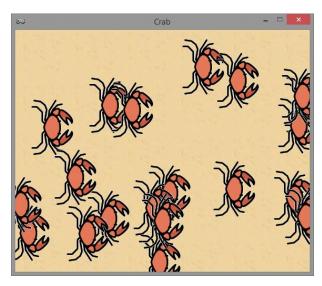


Abbildung 26: Krabbeninvasion

6.3 "... und Action bitte"

Alle Krabben sollen sich nun gemäß dem in der Klasse Crab definierten Verhalten bewegen³². Wir implementieren in der update-Funktion:

```
all_crabs = []
for dummy in range(20):
    c = create_random_crab()
    all_crabs.append(c)

for c in all_crabs:
    c.appear_on_stage(beach)

def update():
    for c in all_crabs:
        c.act()
```

Das Ergebnis sind zwanzig hektisch durcheinander über den Strand laufende Krabben. Abbildung 27 zeigt einen Beispiel-Screenshot.

Beachten Sie bitte, dass wir die for i in range(20): Zeile in for dummy in range(20): abgeändert haben. Dies ist eine verbreitete Art, auszudrücken, dass uns der Wert der Variablen in nicht wirklich interessiert. Wir wollen nur einen Anweisungsblock 20 Mal ausführen.

³² Sollten Sie das Beispiel des vorangegangenen Abschnitts ausprobiert haben, haben Sie vielleicht festgestellt, dass sich die Krabben schon bewegen. Wieso das so ist, werden wir später erläutern. Für den Moment implementieren wir eine eigene update-Funktion, weil wir dann sicher wissen, was diese tut. Später werden wir lernen, dass es die von der hshlib importierte "Standard"-update-Funktion ist, die für die zuvor schon stattfindende Bewegung verantwortlich ist.



Abbildung 27: Krabeninvasion in Bewegung

54 7.1 Worm-Klasse

7 Krabben fressen Würmer

7.1 Worm-Klasse

Wir erstellen eine neue Klasse Worm, mit der wir viele Würmer auf die Bühne befördern wollen. Die Würmer sollen sich zunächst ganz langsam in zufällige Richtungen bewegen.

Die neue Klasse worm besitzt eine __init__-Methode ohne konzeptionelle Überraschungen.

Die act-Methode benutzt die Zufallsfunktion random.uniform, um eine Zufallszahl zu erzeugen, die zwischen -0,5 und 0,5 liegt. Der jeweils resultierende Zufallswert wird auf self.pos[0] bzw. self.pos[1] aufaddiert. Dazu ist anzumerken, dass die eckigen Klammern den Zugriff auf einzelne Elemente des Tupels self.pos ermöglichen. An der Stelle self.pos[0] steht die x-Koordinate und an self.pos[1] die y-Koordinate.

In der drittletzten Zeile steht in der if-Bedingung der Aufruf einer neuen Methode self.stage.is_beyond_edge. Diese Methode ist in der pgzo in der Stage-Klasse realisiert. Sie erwartet eine Position und liefert einen Wahrheitswert True, wenn die Position außerhalb des Bühnenrandes liegt. Mit self.stage können wir direkt auf die Bühne zugreifen, auf der sich der Wurm, dessen act-Methode gerade ausgeführt wird, befindet. Schließlich führen wir mit not ein Schlüsselwort ein, mit dem man in Python einen Wahrheitswert "umdrehen" kann. Informatiker nennen das negieren. Wenn self.stage.is_beyond_edge(new_pos) den Wert False liefert, dann wird der eingerückte Block unterhalb der Bedingung ausgeführt, sonst nicht.

Nun erzeugen wir mehrere Würmer und genau eine Krabbe und lassen diese Objekte allesamt die Bühne betreten:

```
def create_random_worm():
    return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))

for dummy in range(20):
    w = create_random_worm()
    w.appear_on_stage(beach)

c = Crab((WIDTH / 2, HEIGHT / 2), 5, 7, 15)
    c.appear_on_stage(beach)
```

Wie Sie erkennen können, haben wir uns entschieden, die Worm-Objekte nicht in einer Liste abzulegen, sondern nur auf die Bühne zu schicken. Als letztes lassen wir alle auf der Bühne versammelten Objekte ihr in act programmiertes Verhalten zeigen. Wir machen das etwas anders, als eben, indem wir eine neue Methode der Stage verwenden, die alle auf der Bühne versammelten Objekte liefert:

```
def update():
    for o in beach.get_game_objects():
        o.act()
```

Diese update-Funktion wird nun sowohl die zwanzig Worm-Objekte als auch das eine Crab-Objekt mit je einem act-Aufruf versorgen. Ein Szenenbild zeigt Abbildung 28.

7.2 Würmer fressen 55

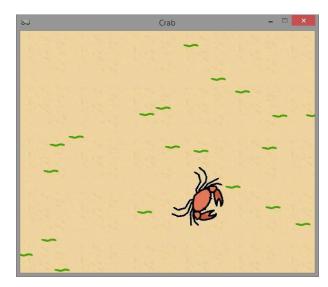


Abbildung 28: Eine Krabbe und viele Würmer

Dass wir nicht eine eigene Liste benutzt haben wie in Abschnitt 6.3 liegt daran, dass wir nun gleich daran gehen wollen, Spielobjekte von der Bühne zu entfernen. Die for-Schleife in update bearbeitet ein Spielobjekt nach dem anderen und ruft dessen act-Methode auf. Was passiert nun, wenn der act-Aufruf eines Spielobjekts A dazu führt, dass ein anderes Spielobjekt B die Bühne verlässt. Z. B. weil A das Spielobjekt B gefressen hat. Wenn sich die for-Schleife zum Spielobjekt B vorgearbeitet hat, wollen wir verhindern, dass jetzt noch die act-Methode von B ausgeführt wird. Denn B ist ja gar nicht mehr präsent.

Wenn wir eine eigene Liste wie in Abschnitt 6.3 benutzen würden, müssten wir so etwas wie das hier programmieren:

```
def update():
    crab.act()
    for w in all_worms:
       if beach.is_on_stage(w):
          w.act()
```

Weil die zusätzliche if-Anweisung etwas länger und komplizierter ist, haben wir stattdessen die Methode beach.get_game_objects() benutzt. Diese liefert immer ein aktuelles Bild aller auf der Bühne anwesenden Spielobjekte. Zwischenzeitlich gefressene Objekte werden nicht mehr geliefert.

7.2 Würmer fressen

Die Krabbe soll jeden Wurm, dem sie begegnet, fressen.

Wir programmieren in der act-Methode der Klasse Crab wie folgt:

Um ein Aufeinandertreffen der Krabbe mit einem Wurm festzustellen, kann man die in Pygame Zero eingebaute und bereits bekannte Funktion colliderect genutzt werden. Nachteil dieser Funktion ist, dass sie durch Verwendung von Hüllrechtecken auch dann Begegnung attestiert, wenn eigentlich noch ein paar Pixel Platz zwischen Krabbe und Wurm ist. Wir haben stattdessen die in der pgzo realisierte Funktion overlaps benutzt, die eine exakte Kollision berechnet.

56 7.2 Würmer fressen

Die if-Anweisung am Ende der act-Methode prüft also, ob die Krabbe (self) mit dem durch worm referenzierten Wurm kollidiert. Falls ja, wird der worm von der Bühne gewiesen (worm.leave_stage()) und ein kurzer Sound abgespielt (sonds.blop.play()). Die for-Schleife durchwandert ähnlich wie in Abschnitt 7.1 alle auf der Bühne versammelten Objekte. Anders ist, dass wir mit self.stage wieder direkt auf die Bühne zugreifen, auf der sich die Krabbe gerade befindet. Und außerdem ist der Parameter ganz am Ende der Zeile neu: self.stage.get_game_objects(worm). worm (groß geschrieben) ist eine Klasse. Die Methode get_game_objects liefert hier nicht alle auf der Bühne versammelten Objekte, sondern nur diejenigen, die der Klasse worm angehören.

Wenn wir das Programm nun laufen lassen, können wir der Krabbe dabei zusehen, wie sie langsam aber sicher den Strand abräumt. Nach einer Weile, deren Länge vom Zufall abhängt, ist der letzte Wurm gefressen und die Krabbe irrt weiter auf dem leeren Strand umher.

Was wird wohl passieren, wenn wir den Parameter Worm weglassen? Also:

```
for worm in self.stage.get_game_objects():
    if worm.overlaps(self):
        worm.leave_stage()
        sounds.blop.play()
```

Dann wird die Krabbe eine Überlappung (overlaps) mit sich selbst feststellen und sich selbst von der Bühne entfernen ©. Das geht so schnell beim Start des Spiels, dass es so aussieht, als sei nie eine Krabbe vorhanden gewesen. Die Selbstentfernung der Krabbe ist nur einem kurzen "Blop" akustisch festzustellen.

8 Hummer fressen Krabben

8.1 Hummer hinzufügen

Wir ergänzen eine weitere Spielfigur: einen Hummer. Hummer sollen in unserem Spiel den Feind der Krabbe darstellen. Das Verhalten der Hummer soll dem der Krabben gleichen, nur dass Hummer halt keine Würmer, sondern Krabben fressen. Wir können die neue Klasse Lobster recht einfach zunächst als Kopie der Klasse Crab anlegen und dann entsprechend anpassen. Die Stellen, die wir angepasst haben, sind im Folgenden hervorgehoben dargestellt:

```
class Lobster(GameObj):
   def __init__(self, pos, speed, drunkenness, jumpiness):
        self.image = "lobster"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
   def act(self):
        if self.can_move(self.speed):
           self.move(self.speed)
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        for crab in self.stage.get_game_objects(Crab):
            if crab.overlaps(self):
                crab.leave_stage()
                sounds.au.play()
```

Wir schreiben eine eigene Funktion, um einen zufälligen Hummer zu erzeugen. In dem Spiel wird es später darum gehen, dass der Spieler die Krabbe steuert und dabei versuchen muss, dem Hummer auszuweichen. Für verschiedene Spiellevel schwebt uns vor, Hummer mit verschiedenen Stärken zu erzeugen. Aus diesem Grund bekommt die folgende Funktion einen Parameter strength. Der zu übergebende Wert muss zwischen o und 1 liegen. O bedeutet schwach, 1 bedeutet stark:

Zunächst haben wir uns entschieden, den Hummer in der oberen Bildhälfte zu erzeugen. Die y-Koordinate liegt zwischen 0% und 40% der Gesamthöhe des Fensters. Die Krabbe werden wir gleich in der unteren Bildhälfte platzieren, damit man zu Beginn des Spiels wenigstens eine Chance hat und nicht sofort mit einem Hummer kollidiert.

Die letzten drei Werte für speed, drunkenness und jumpiness werden abhängig vom übergebenen strength-Wert linear zwischen 1 und einem oberen Maximalwert interpoliert.

Den neu erzeugten Hummer legen wir in einer Variablen result ab. Dies tun wir, weil wir den Hummer noch um einen zufälligen Winkel drehen wollen, bevor wir ihn mit return zurückgeben.

Wir platzieren zwei Hummer auf der Bühne. Der neue Code ist im Folgenden hervorgehoben dargestellt:

58 8.2 Tastatursteuerung

```
crab = Crab((WIDTH / 2, HEIGHT * 0.7), 5, 7, 15)
crab.appear_on_stage(beach)

for dummy in range(20):
    w = create_random_worm()
    w.appear_on_stage(beach)

for dummy in range(2):
    lobster = create_random_lobster(0.2)
    lobster.appear_on_stage(beach)

def update():
    for o in beach.get_game_objects():
        o.act()
```

Wenn wir das Spiel starten, können wir beobachten, wie die Krabbe Würmer vertilgt und dabei von den beiden Hummern gejagt wird. Manchmal ist die Krabbe erfolgreich und verspeist alle Würmer, bevor sie selbst gefangen wird. Ein Schnappschuss ist in Abbildung 29 dargestellt.

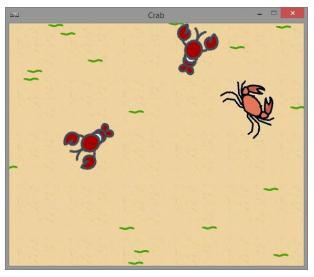


Abbildung 29: Schnmappschuss der von Hummern gejagten Krabbe, die schon einige Würmer erwischt hat

8.2 Tastatursteuerung

Wir wollen die Krabbe so umbauen, dass sie vom Spieler über die Tastatur gesteuert werden kann. Die linken und rechten Pfeiltasten sollen eine Drehung der Krabbe um die eigene Achse bewirken. Die oberen und unteren Pfeiltasten dienen der Beschleunigung oder der Verlangsamung der aktuellen Fortbewegungsgeschwindigkeit.

Zuerst einmal müssen wir jedoch den Code ausbauen, der für die automatischen Wendemanöver am Bühnenrand, die Trunkenheit und die Sprunghaftigkeit verantwortlich ist. Wenn die Krabbe am Bühnenrand anstößt, soll sie einfach stehen bleiben. Der folgende Code zeigt die Streichungen und Hinzufügungen hervorgehoben an:

8.2 Tastatursteuerung 59

```
class Crab(GameObi):
   def __init__(self, pos, speed, drunkenness, jumpiness):
        self.image = "crab
        self.pos = pos
        self.speed = speed 0 # Start at speed 0
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
   def act(self):
       if self.can_move(self.speed):
           self.move(self.speed)
        else:
            self.turn(25)
            self.speed = 0
        if random.randrange(10) < self.drunkenness:
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        for worm in self.stage.get_game_objects(Worm):
            if worm.overlaps(self):
                worm.leave_stage()
                sounds.blop.play()
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
```

Im unteren Bereich der neuen act-Methode ist die Tastatursteuerung untergebracht. Links- und Rechtsdrehung müssten selbsterklärend sein. Neu ist für die Beschleunigung (Pfeil nach oben) der Aufruf der Methode self.speed_up. Diese Methode in der Klasse GameObj erhöht den Attributwert von self.speed um einen kleinen Wert. Falls die maximale Geschwindigkeit erreicht ist, tut self.speed_up nichts. Die Methode self.slow_down tut das Gegenteil: sie verringert die Geschwindigkeit bis zu einer minimalen negativen Geschwindigkeit. Eine negative Geschwindigkeit lässt die Krabbe rückwärts laufen.

Wir können uns einmal die Implementierung von speed_up und slow_down ansehen, indem wir die Datei pgzo.py inspizieren:

```
def speed_up(self):
    self.speed += 0.1
    if self.speed > self.MAX_SPEED: self.speed = self.MAX_SPEED

def slow_down(self):
    self.speed -= 0.1
    if self.speed < self.MIN_SPEED: self.speed = self.MIN_SPEED</pre>
```

An dieser Implementierung können wir zwei neue Dinge über Python lernen. Erstens ist es offenbar erlaubt, hinter dem Doppelpunkt der if-Bedingung direkt in der gleichen Zeile weiter zu schreiben. Diese Schreibweise bietet sich an, wenn bei Zutreffen der Bedingung sowieso nur eine Anweisung ausgeführt werden soll. Zweitens fällt uns die Anweisung self.speed += 0.1 auf. Dies ist tatsächlich eine Kurzschreibweise für self.speed = self.speed + 1. Entsprechend steht self.speed -= 0.1 für self.speed = self.speed - 1.33

Für einen ungestörten Spielgenuss müssen wir die Erzeugung der Krabbe noch anpassen, denn wir haben ja aus der __init__-Methode einige Parameter entfernt. Die neue hierzu benötigte Anweisung ist:

```
crab = Crab((WIDTH / 2, HEIGHT * 0.7))
```

Dies platziert die Krabbe in der unteren Fensterhälfte.

So, das Zocken kann beginnen. Viel Vergnügen!

³³ Für erfahrene Leser, die sich bereits mit anderen Programmiersprachen auskennen, sei hier erwähnt, dass Python keine "++"- und keine "--"- Operatoren unterstützt.

8.3 Das vollständige bisherige Programm

Das vollständige bisherige Programm drucken wir hier einmal ab. Sie können selbst im Programm an den <u>hervorgehobenen</u> Stellen Änderungen vornehmen, um bspw. die Spielstärke und Anzahl der Gegner oder die Anzahl der Würmer zu ändern.

```
import random
import pgzrun
from pgzo import *
WIDTH = 560
                # screen width
HEIGHT = 460 # screen height
TITLE = "Crab" # window title
beach = Stage()
beach.background_image = "sand"
beach.show()
class Crab(GameObj):
    def __init__(self, pos):
        self.image = "crab"
        self.pos = pos
        self.speed = 0
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
            self.speed = 0
        for worm in self.stage.get_game_objects(Worm):
            if worm.overlaps(self):
                worm.leave_stage()
                sounds.blop.play()
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
class Lobster(GameObj):
    def __init__(self, pos, speed, drunkenness, jumpiness):
    self.image = "lobster"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        for crab in self.stage.get_game_objects(Crab):
            if crab.overlaps(self):
                crab.leave_stage()
                sounds.au.play()
```

```
class Worm(GameObj):
    def __init__(self, pos):
    self.image = "worm"
        self.pos = pos
    def act(self):
        new_pos = (self.pos[0] + random.uniform(-0.5, 0.5),
                   self.pos[1] + random.uniform(-0.5, 0.5))
        if not self.stage.is_beyond_edge(new_pos):
            self.pos = new_pos
def create_random_lobster(strength):
    result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                     1 + strength * 4,
                     1 + strength * 9,
                     1 + strength * 24 )
    result.turn(random.randrange(360))
    return result
def create_random_worm():
    return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
crab = Crab((WIDTH / 2, HEIGHT * 0.7))
crab.appear_on_stage(beach)
for dummy in range(20):
    w = create_random_worm()
    w.appear_on_stage(beach)
for dummy in range(\frac{2}{2}):
    lobster = create_random_lobster(0.2) # strength 0.2
    lobster.appear_on_stage(beach)
def update():
    for o in beach.get_game_objects():
        o.act()
pgzrun.go()
```

9 Spielende

9.1 Die update- und draw-Funktionen

In unserem Spiel, dessen letzter Stand in Abschnitt 8.3 abgedruckt ist, haben wir keine draw-Funktion, jedoch eine update-Funktion geschrieben. Die draw-Funktion hatten wir gleich zu Beginn weggelassen, weil wir gelernt hatten, dass die aktive Bühne selbst für das Zeichnen sorgt. Dies gilt nun aber ebenfalls für die update-Funktion. Auch die update-Funktion wird automatisch von der aktiven Bühne realisiert, und zwar so, dass alle auf der Bühne anwesenden Spielobjekte mit einem Aufruf ihrer act-Methode versorgt werden. Wir können unsere eigene update-Funktion, die ja nichts anderes macht, einfach entfernen und das Spiel funktioniert weiterhin. Die pgzo importiert alles Notwendige.

9.2 Bühnenmechanik

Bisher haben wir es gut im Griff, wenn wir Spielmechanik einbauen wollen, die eine ganz bestimmte Spielfigur betrifft. Wir können dann den benötigten Programmcode einfach in die act-Methode der Klasse der betreffenden Spielfigur schreiben. Was machen wir jedoch mit Spielmechanik, die eher die gesamte Bühne betrifft? Ein Beispiel ist die Erkennung des Spielendes. Dazu müssen wir prüfen, ob auf der Bühne überhaupt noch eine Krabbe bzw. überhaupt noch ein Wurm zugegen ist. Und falls nicht, ist das Spiel zu Ende.

Wir erstellen zunächst die folgende Funktion:

```
def beach_update():
    defeated = beach.count_game_objects(Crab) == 0
    victorious = beach.count_game_objects(worm) == 0
    if defeated or victorious: # gameover
        sys.exit()
```

In dieser Funktion werden mehrere Variablen erzeugt, die einen Wahrheitswert besitzen. Die erste Zeile defeated = ... belegt die Variable defeated mit dem Wert, der aus dem Vergleich beach.count_game_objects(Crab) == 0 ermittelt wird. Der Vergleich zählt die Anzahl der Krabben auf der beach-Bühne. Wenn diese 0 ist, dann erhält defeated den Wert True, sonst erhält defeated den Wert False.

Ganz entsprechend bekommt victorious den Wert True genau dann, wenn kein Wurm mehr auf der Bühne verweilt.

Schließlich verknüpfen wir die beiden Werte defeated und victorious mit einem logischen Oder (or). Das Spiel ist zu Ende, wenn wir verloren haben (defeated == True) oder wenn wir gewonnen haben (victorious == True). Tatsächlich ist die Bedingung hinter dem if auch dann True, wenn beide Variablen defeated UND victorious den Wert True besitzen. Das wird in unserem Spiel aber kaum gleichzeitig vorkommen, bzw. es müsste ein großer Zufall eintreten, dass die Krabbe exakt in dem Moment den letzten Wurm frisst, in dem sie selbst vom Hummer gefressen wird.

In Python gibt es mehrere Operatoren für Wahrheitswerte. Man nennt diese Operatoren auch **boolesche Operatoren** oder **logische Operatoren**. Es gibt drei (naheliegende) Operatoren: and, or und not.

Am Ende unserer beach_update-Funktion prüfen wir in einer if-Anweisung, ob wenigstens einer der beiden Werte defeated und victorious wahr ist. Falls ja, beenden wir das Spiel mit sys.exit().

Damit diese Funktion zur Ausführung kommt, müssen wir die Funktion in der beach-Bühne als Attribut speichern. Dies geht wie folgt:

```
beach = Stage()
beach.background_image = "sand"
def beach_update():
    defeated = beach.count_game_objects(Crab) == 0
    victorious = beach.count_game_objects(Worm) == 0
    if defeated or victorious: # gameover
        sys.exit()
beach.update = beach_update
beach.show()
```

Beachten Sie: links vom Gleichheitszeichen steht ein Punkt zwischen beach und update. D. h. links vom Gleichheitszeichen wird ein Attribut des beach-Objekts referenziert. Rechts vom Gleichheitszeichen steht ein Unterstrich zwischen beach und update. D. g. rechts vom Gleichheitszeichen wird die unmittelbar zuvor definierte Funktion beach_update referenziert.

Durch die oben <u>hervorgehobene</u> Zuweisung erfährt die <u>beach</u>-Bühne und damit die <u>pgzo</u> von unserer kleinen <u>beach_update</u>-Funktion und führt diese etwa 60 Mal in der Sekunde aus.

9.3 Einen Text anzeigen

Der harte Spielabbruch ist nicht besonders schön. Wir würden gerne auf dem Bildschirm anzeigen, ob das Spiel gewonnen oder verloren wurde.

Die Anzeige einer Meldung erfolgt, wie wir schon wissen, in der einer draw-Funktion. Wir können ähnlich, wie wir es eben mit der beach_update-Funktion gemacht haben, eine weitere Funktion beach_draw erstellen und der Bühne bekanntgeben:

```
def beach_draw():
    if beach.count_game_objects(Crab) == 0:
        screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25))
    if beach.count_game_objects(Worm) == 0:
        screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25))
beach.draw = beach_draw
```

Wir haben die von Pygame Zero angebotene Funktion screen.draw.text() benutzt. Diese Funktion unterstützt verschiedene Schriftarten, -farben, -größen, usw. Eine Beschreibung vieler Möglichkeiten der Funktion findet man in der Online-Dokumentation³⁴ der entsprechenden Funktion von Pygame Zero.

Damit wir überhaupt etwas von dem neuen Text sehen können, bauen wir übergangsweise einen Teil der beach_update-Funktion zurück:

```
def beach_update():
    defeated = beach.count_game_objects(Crab) == 0
    victorious = beach.count_game_objects(Worm) == 0
    if defeated or victorious: # gameover
        #sys.exit()
```

Wir haben den <code>sys.exit()</code>-Befehl durch ein voran gestelltes <code>#-Zeichen</code> einfach zu einem Kommentar umfunktioniert. Dadurch wird der Befehl deaktiviert, aber er steht noch im Programm und kann leicht wieder vom Programmierer aktiviert werden. Leider ist das Programm so nicht übersetzbar. Der Grund: hinter einer <code>if</code>-Bedingung muss hinter dem Doppelpunkt immer ein Anweisungsblock oder eine einzelne Anweisung stehen. Wir können das Problem aus zwei Arten lösen:

```
def beach_update():
    defeated = beach.count_game_objects(Crab) == 0
    victorious = beach.count_game_objects(Worm) == 0
    #if defeated or victorious: # gameover
    # sys.exit()
```

Nun ist die komplette if-Anweisung **auskommentiert** und alles in Ordnung. Häufig findet man allerdings auch die folgende Variante:

³⁴ https://pygame-zero.readthedocs.io/en/stable/ptext.html

```
def beach_update():
    defeated = beach.count_game_objects(Crab) == 0
    victorious = beach.count_game_objects(Worm) == 0
    if defeated or victorious:  # gameover
        #sys.exit()
    pass
```

Der syntaktisch notwendige Anweisungsblock wurde nun einfach durch Aufruf der in Python eingebauten pass-Anweisung umgesetzt. Die Anweisung pass tut zur Laufzeit einfach nichts, macht aber den Python-Interpreter glücklich ©.

Wir lassen unser neues Programm laufen und verlieren das Spiel mutwillig. Das Ergebnis zeigt Abbildung 30. Wir sind nicht zufrieden. Den Text kann man in weißer Farbe kaum lesen. Auch die Schriftgröße und Schriftart sind verbesserungswürdig. Wir ersetzen die Anweisung durch die folgende:

Das Ergebnis ist in Abbildung 31 dargestellt.

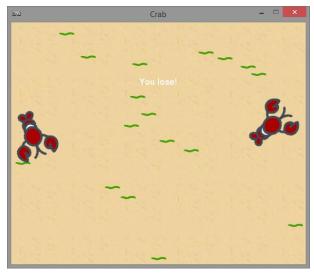


Abbildung 30: Die noch unschöne Meldung über ein verlorenes Spiel



Abbildung 31: Die schönere Meldung über ein verlorenes Spiel

9.4 Eine Beach-Klasse 65

So weit so gut. Nicht so gut gelungen ist, dass die beach_draw-Funktion fast dieselben Anweisungen noch einmal ausführt, die schon in der beach_update-Funktion ausgeführt wurden. Das ist eine Verschwendung von CPU-Leistung, die man sich in Computerspielen normalerweise nicht leisten darf. Wir wollen daher in der beach_update-Funktion die ermittelten Werte für defeated und victorious so zwischen speichern, dass die beach_draw-Funktion auf die Werte einfach direkt zugreifen kann. Im Moment kann beach_draw nämlich nicht auf die in beach_update definierten Variablen defeated und victorious zugreifen. Man kann sich das so vorstellen, dass diese beiden Variablen, die in beach_update in einer Zuweisung definiert wurden, Privateigentum der Funktion beach_update.

Eine einfache Lösung besteht darin, keine lokalen Variablen defeated und victorious zu verwenden, sondern stattdessen Attribute im beach-Objekt zu erzeugen:

```
beach = Stage()
beach.background_image = "sand"
def beach_update(self):
    self.defeated = self.count_game_objects(Crab) == 0
    self.victorious = self.count_game_objects(worm) == 0
   #if self.defeated or self.victorious: # gameover
         sys.exit()
beach.update = beach_update
def beach_draw(self):
    if self.defeated:
        screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                         color="brown", fontsize=60, fontname="zachary")
    if self.victorious:
        screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                         color="brown", fontsize=60, fontname="zachary")
beach.draw = beach_draw
beach.show()
```

9.4 Eine Beach-Klasse

Die Verwendung des Parameters self in den beiden Funktionen beach_draw und beach_update hat Sie vielleicht auf die Idee gebracht, dass die Zuordnung dieser beiden Funktionen zum beach-Objekt doch eigentlich genauso funktionieren müsste, wie wir das bei der Einführung der Crab-Klasse gemacht haben. Und genau so ist es. Wir wollen den obigen Code für das beach-Objekt nun in eine Beach-Klasse überführen, von der wir dann ein Objekt beach erzeugen wollen.

Klassen sind insbesondere dann interessant und wichtig, wenn man viele Objekte einer Klasse anlegen will. So war es ja bei unseren Spielfiguren. Hier haben wir es mit der Strandbühne zu tun, von der wir eigentlich nur eine benötigen. Dennoch kann es auch in so einem Fall sinnvoll sein, eine Klasse zu erstellen. Eine Klasse bildet einen Rahmen um die zusammen gehörenden Daten und Funktionen. Das alleine rechtfertigt bereits eine Klasse.

Wenn wir genau hinsehen, hat sich bereits einiges an Code angesammelt, der zu unserem beach-Objekt gehört. Insbesondere der Initialisierungscode, der die Hummer-, Würmer- und Krabben-Objekte erzeugt, gehört in eine Initialisierungsfunktion der Klasse Beach:

66 9.4 Eine Beach-Klasse

```
import random
import pgzrun
from pgzo import *
WIDTH = 560
              # screen width
HEIGHT = 460 # screen height
TITLE = "Crab" # window title
class Beach(Stage):
   def __init__(self):
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(20):
            w = Beach._create_random_worm()
            w.appear_on_stage(self)
        for dummy in range(2):
            lobster = Beach._create_random_lobster(0.2) # strength 0.2
            lobster.appear_on_stage(self)
   @staticmethod
   def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                         1 + strength * 4,
                         1 + strength * 9,
                         1 + strength * 24 )
        result.turn(random.randrange(360))
        return result
   @staticmethod
   def _create_random_worm():
        return Worm( (random.randrange(WIDTH), random.randrange(HEIGHT)) )
   def update(self):
        self.defeated = self.count_game_objects(Crab) == 0
        self.victorious = self.count_game_objects(Worm) == 0
        #if self.defeated or self.victorious: # gameover
            sys.exit()
   def draw(self):
        if self.defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
        if self.victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
class Crab(GameObj):
   ... wie in Abschnitt 8.3 ...
class Lobster(GameObj):
    ... wie in Abschnitt 8.3 ...
class Worm(GameObj):
   ... wie in Abschnitt 8.3 ...
beach = Beach()
beach.show()
pgzrun.go()
```

Die Klasse Beach ist eine Unterklasse von Stage. Deshalb steht class Beach(Stage): in der ersten Zeile der Klasse. Die Methoden draw und update sind fast identische Kopien der vorherigen beach_draw- und beach_update-Funktionen. Nur der Name wurde jeweils geändert.

9.5 Effizienz

Die __init__-Methode nimmt den gesamten Code aus Abschnitt 8.3 auf, der sich mit dem Erzeugen von Spielfiguren beschäftigte. Anders als dort wird nun der self-Parameter der __init__-Methode benutzt, um auf das Bühnenobjekt zu verweisen.

Interessant die die beiden Hilfsfunktionen aus Abschnitt 8.3, die einen einzelnen Hummer bzw. einen einzelnen Wurm erzeugten: create_random_lobster und create_random_worm. Die Funktion create_random_worm benötigt keinerlei Informationen von außerhalb der Funktion. Deshalb hatten wir diese Funktion ohne Parameter ausgestattet. Allerdings müssen eigentlich alle Methoden einer Klasse als ersten Parameter self definieren. Für create_random_worm ist das aber überflüssig, denn die Funktion erledigt ihren Job auch ohne Kenntnis des Bühnenobjekts. Für solche Fälle hat Python sog. statische Methoden eingeführt. Eine statische Methode hat keinen self-Parameter, kann aber andere Parameter besitzen, wie wir gut an der Methode _create_random_lobster beobachten können. Damit Python weiß, dass es sich um eine Methode ohne self-Parameter handelt, schreibt man vor den Methodenkopf die Zeile @staticmethod. Diese Zeile ist ein sog. Decorator, der die Funktion mit der Information "statische Methode" dekoriert.

Dass wir die beiden _create_random_...-Methoden mit einem Unterstrich haben beginnen lassen, ist eine Python-Konvention für sog. private Elemente einer Klasse. Da diese beiden Methoden ausschließlich für die Verwendung innerhalb der Klasse Beach vorgesehen sind, kann man dies durch den führenden Unterstrich kenntlich machen³⁵. Niemand verhindert jedoch tatsächlich, dass auch Programmteile außerhalb der Beach-Klasse die beiden Methoden aufrufen. Wie gesagt: es handelt sich um eine Konvention.

Um statische Methoden aufzurufen, ist es üblich, nicht self._create_random_...(...) zu schreiben, sondern Beach._create_random_...(...) Dies drückt nämlich klar lesbar aus, dass die _create_random_...-Methode keine Informationen des self-Objekts nutzt.

Die Klasse Beach ist eine Klasse und noch kein Objekt. Ein Objekt erzeugen wir ganz bewusst erst ganz am Ende des Programms, nachdem alle anderen Klassen definiert wurden. Denn: Die Anweisung Beach() wird zur Laufzeit wie ein Funktionsaufruf unmittelbar ausgeführt. Wenn die Zeile beach = Beach() oberhalb der Crab-, worm- und Lobster-Klassen stehen würde, würde die __init__-Methode der Beach-Klasse aufgerufen, bevor der Python-Interpreter den Rest der Datei crab.py lesen konnte. In der __init__-Methode müssen die Crab-, worm- und Lobster-Klassen aber bekannt sein, da ja Objekte dieser Klassen angelegt werden. Der Python-Interpreter verarbeitet eine Datei von oben nach unten. Alles was unter der aktuell verarbeiteten Zeile steht, hat der Python-Interpreter noch nicht gelesen und kann er nicht kennen. In so einem Fall meldet der Python-Interpreter dann eine Fehlermeldung der folgenden Art: NameError: name 'Crab' is not defined.

9.5 Effizienz

Die update-Methode der Klasse Beach untersucht 60 Mal in der Sekunde, ob es noch Würmer bzw. Krabben auf der Bühne gibt. Das kostet durchaus CPU-Leistung. Eigentlich ist es nicht nötig, dass diese Aufgabe 60 Mal in der Sekunde ausgeführt wird. Es reicht eigentlich, dass wir den Moment abpassen, in dem bspw. die Krabbe einen Wurm von der Bühne tilgt. In diesem Moment ist es sinnvoll, zu prüfen, ob noch Würmer übrig sind.

Wir verschieben daher die Anweisung von der Klasse Beach in die update-Methode der Klasse Crab, die den Wert es Attributs victorious berechnet. Wir betrachten zunächst die Änderungen in der Methode update der Klasse Beach:

35 Es gibt auch normale (nicht-statische) private Methoden. Dass hier gerade die beiden statischen Methoden privat sind, ist eher Zufall.

68 9.6 Kapselung

```
def update(self):
    self.defeated = self.count_game_objects(Crab) == 0
    self.victorious = self.count_game_objects(Worm) == 0
    #if self.defeated or self.victorious: # gameover
    # sys.exit()
```

In der update-Methode haben wir die Anweisung gestrichen, die wir nach Crab verschieben wollen. Die act-Methode in Crab ändern wir wie folgt:

```
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.speed = 0
    for worm in self.stage.get_game_objects(Worm):
        if worm.overlaps(self):
            worm.leave_stage()
            sounds.blop.play()
            self.stage.victorious = self.stage.count_game_objects(worm) == 0
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
```

D. h. wir haben einfach eine Anweisung verschoben. Die neue Anweisung in Crab mussten wir etwas umformulieren, da wir mit self.stage einen anderen Zugriff auf die Strandbühne benötigen, als in der Original-Anweisung. Ansonsten ist nichts verändert worden. Die neue Anweisung in der act-Methode in Crab wird nur dann ausgeführt, wenn die if-Bedingung worm.overlaps(self) wahr ist. D. h. nur in dem Fall, dass gerade ein Wurm die Bühne verlassen hat. Das spart CPU-Leistung.

Wir aktivieren zu Testzwecken die gameover-if-Anweisung, indem wir die Kommentarzeichen am Anfang der beiden betreffenden Zeilen entfernen.

```
def update(self):
    self.defeated = self.count_game_objects(Crab) == 0
    if self.defeated or self.victorious: # gameover
        sys.exit()
```

Das so gestartete Spiel bricht sofort mit einer Fehlermeldung ab:

```
if self.defeated or self.victorious: # gameover
AttributeError: 'Beach' object has no attribute 'victorious'
```

Die Ursache ist, dass die if-Bedingung auf ein Attribut victorious lesend zugreift, das noch gar nicht existiert. Das Attribut wird ja nur dann mit einem Wert belegt, wenn ein Wurm gefressen wird. Aber das ist zu Beginn des Spiels, wenn die update-Methode der Strandbühne erstmals ausgeführt wird, noch nicht der Fall. Wir müssen dafür sorgen, dass die in der update-Methode der Klasse Beach gelesenen Attribute auch wirklich existieren. Das erledigen wir in der __init__-Methode von Beach:

```
def __init__(self):
    self.defeated = False
    self.victorious = False
    self.background_image = "sand"
    ... Rest wie bisher ...
```

Nun funktioniert es wieder.

9.6 Kapselung

Das Programm ist insgesamt etwas effizienter geworden. Allerdings: es ist leider auch etwas verworrener geworden. Schauen wir uns einmal die Klasse Beach genauer an. In der Klasse steht in der __init__-Methode die Intialisierung des Attributs victorious auf den Wert False und in der Methode update wird

9.6 Kapselung 69

dieser Wert gelesen. Es gibt in der gesamten Klasse Beach keinen Hinweis darauf, dass sich an diesem Wert etwas ändert. Dazu müssen wir erst in die update-Methode der Klasse Crab sehen.

Man wünscht sich normalerweise Programmcode, der übersichtlich ist. Übersichtlich ist er unter anderem dann, wenn alles, was zusammen gehört, auch in einer Klasse steht. Der schreibende Zugriff von außen (Klasse Crab) auf den Zustand der Strandbühne ist unerwünscht.

In unserem Programm ist es ja die Strandbühne, die darüber Buch führt, ob das Spiel gewonnen oder verloren ist. Die Strandbühne ist sozusagen der Schiedsrichter, der einen Zettel in der Hemdtasche mit sich führt, auf dem er sich notiert: victorious = False und defeated = False. Wenn nun die Krabbe dem Schiedsrichter ohne zu fragen in die Hemdtasche greift und einfach auf den Zettel victorious = True schreibt, ist das mindestens überraschend, wenn nicht sogar unhöflich.

Wir würden den Zugriff auf den Spielzustand, als den Zugriff auf die Attribute victorious und defeated gerne vor unberechtigtem Zugriff schützen. In der Programmierung nennt man das **Kapselung**. In Python gibt es keinen sicheren Weg der Kapselung, nur Konventionen. Wir werden der Konvention folgen und die beiden Attribute mit einem führenden Unterstrich beginnen lassen. Damit signalisieren wir der Krabbe: Finger weg, das ist mein **privater Zustand**:

```
class Beach(Stage):
    def __init__(self):
                                                       Beachten Sie den neuen Unterstrich vor den beiden
        self._defeated = False
                                                                  initialisierten Attributen
        self._victorious = False
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(1):
            w = Beach._create_random_worm()
            w.appear_on_stage(self)
        for dummy in range(2):
            lobster = Beach._create_random_lobster(0.2) # strength 0.2
            lobster.appear_on_stage(self)
   @staticmethod
   def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                         1 + strength * 4,
                         1 + strength * 9,
                         1 + strength * 24 )
        result.turn(random.randrange(360))
        return result
   @staticmethod
   def create random worm():
        return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
   def update(self):
                                                                                     Hier auch mit Unterstrich
        self._defeated = self.count_game_objects(Crab) == 0
        if self._defeated or self._victorious:
            sys.exit()
   def draw(self):
                                             Hier auch mit Unterstrich
        if self._defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
        if self._victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
```

Eine unhöfliche Krabbe würde nun in ihrer act-Methode auch einfach einen Unterstrich ergänzen. Eine höfliche Krabbe bittet den Schiedsrichter durch Aufruf einer noch zu schreibenden Methode, den Spielstand zu ändern, statt es einfach selbst zu tun. Wir ergänzen in Beach eine weitere Methode:

70 9.6 Kapselung

```
def detect_victory(self):
    self._victorious = self.count_game_objects(Worm) == 0
```

In Crab korrigieren wir die act-Methode:

```
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.speed = 0
    for worm in self.stage.get_game_objects(Worm):
        if worm.overlaps(self):
            worm.leave_stage()
            sounds.blop.play()
            self.stage.detect_victory()
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
```

Durch diese Änderung ist die Klasse Beach in sich kohärenter geworden. Alle Operationen, die den Spielzustand betreffen, sind in der Klasse Beach versammelt. Außerdem hat sich die Bindung zwischen Crab und Beach etwas gelockert – man sagt auch Crab und Beach unterliegen nun einer loseren Kopplung als vorher. Die Klasse Crab Crab muss nun selbst eigentlich gar nicht wissen, wie genau der Spielzustand vom Strand-Schiedsrichter verwaltet wird. Crab hat nur die Aufgabe, dem Schiedsrichter im richtigen Moment einen Tipp zu geben: "Jetzt lohnt es sich, den Spielzustand neu zu bewerten. Ich habe nämlich gerade einen Wurm gefressen".

In der unhöflichen Krabbenversion hatten wir geschrieben:

```
self.stage.victorious = self.stage.count_game_objects(worm) == 0
```

Solche Ketten von mehreren Objektnamen bzw. Funktionsnamen, die durch Punkte getrennt sind, gelten als schlechter Programmierstil. Die Krabbe greift hierbei nämlich auf Daten zu, die ihr gar nicht selbst gehören, sondern der Strandbühne. Die unhöfliche Krabbe hält sich nicht an das sog. **Gesetz von Demeter**, das besagt: Objekte sollen nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren. In unserem Fall: die Krabbe darf mit der Strandbühne kommunizieren, aber nicht direkt mit dem Objekt victorious. Das Gesetz von Demeter ist ein wichtiges Gesetz der objektorientierten Programmierung.

Wenn wir uns die neue Methode update in Crab genau ansehen, erkennen wir noch eine weitere Stelle, in der das Gesetz von Demeter verletzt ist:

In den ersten drei Zeilen lässt sich die Krabbe von der Bühne Wurm-Objekte liefern, mit denen sie dann direkt kommuniziert. Stattdessen sollten wir die direkte Kommunikation mit dem Wurm lieber an die Bühne delegieren. In der Klasse Beach programmieren wir dafür eine neue Methode:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(Worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
            self._victorious = self.count_game_objects(Worm) == 0
```

Diese rufen wir aus der update-Methode von Crab heraus aus:

```
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.speed = 0
    self.stage.take_out_neighbouring_worms(self)
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
```

D. h. die Krabbe übergibt eine Referenz auf sich selbst an die neue Methode take_out_neighbouring_worms und überlässt es nun der Bühne, die benachbarten Würmer zu finden und zu entfernen. Dass nun auch die Bühne den blop-Sound abspielt, bot sich hier an, ist aber vielleicht sowieso eine gute Idee.

Die Methode detect_victory haben wir übrigens wieder entfernt. Sie ist in der neuen Version überflüssig geworden.

9.7 Effiziente und wartbare Detektion einer Niederlage

Die neuen Ideen werden wir nun ebenfalls auf den Lobster und dessen Bestreben übertragen, der Krabbe eine Niederlage beizubringen. Dazu entfernen wir im Sinne einer besseren Effizienz zunächst die noch die in der update-Methode von Beach verbliebene Zeile zur Detektion einer Niederlage:

Dann ergänzen wir eine neue Methode in Beach:

```
def take_out_neighbouring_crab(self, lobster):
    for c in self.get_game_objects(Crab):
        if c.overlaps(lobster):
            c.leave_stage()
            sounds.au.play()
        # There is only one crab on the beach, so we set
        # _defeated=True if we found at least one overlapping crab:
        self._defeated = True
```

Die Methode ähnelt der Methode, die benachbarte Würmer entfernt. Im Falle eines gefressenen Wurms hatten wir mit self.count_game_objects(worm) == 0 eine Untersuchung vorgenommen, ob das Spiel denn nun gewonnen sei. Im Falle der gefressenen Krabbe benötigen wir keine umständliche Untersuchung, ob das Spiel nun verloren ist. Wir können einfach self._defeated auf True setzen.

Die neue Methode nutzen wir in Lobster:

```
def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.turn(25)
    if random.randrange(10) < self.drunkenness:
        self.turn(random.uniform(-self.jumpiness, self.jumpiness))
    self.stage.take_out_neighbouring_crab(self)</pre>
```

Die neue Variante spart nun CPU-Leistung ein und ist gleichzeitig vielleicht sogar übersichtlicher als die Version am Ende von Abschnitt 9.4. Grundsätzlich ist es eine gute Idee, beim Eintreten eines Spielobjektbezogenen Ereignisses gleich und sofort alle möglichen Folgen abzuprüfen. Tut man dies erst später, gibt es keinen Anlass mehr dafür und es bleibt einem meistens nur, dies in der update-Funktion der Bühne zu erledigen, die 60 Mal pro Sekunde ausgeführt wird.

Wir zeigen das ganze Programm noch einmal im Zusammenhang. Dabei haben wir die Beendigung des Programms (sys.exit()) erst einmal wieder auskommentiert. Um eine bessere Version des Spielendes kümmern wir uns im folgenden Abschnitt.

```
import random
import pgzrun
from pgzo import *
WIDTH = 560
               # screen width
HEIGHT = 460
             # screen height
TITLE = "Crab" # window title
class Beach(Stage):
   def __init__(self):
        self._defeated = False
        self._victorious = False
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(20):
           w = Beach._create_random_worm()
           w.appear_on_stage(self)
        for dummy in range(2):
            lobster = Beach._create_random_lobster(0.2) # strength 0.2
            lobster.appear_on_stage(self)
   def take_out_neighbouring_worms(self, crab):
        for w in self.get_game_objects(Worm):
            if w.overlaps(crab):
                w.leave_stage()
                sounds.blop.play()
                self._victorious = self.count_game_objects(Worm) == 0
   def take_out_neighbouring_crab(self, lobster):
        for c in self.get_game_objects(Crab):
            if c.overlaps(lobster):
                c.leave_stage()
                sounds.au.play()
                # There is only one crab on the beach, so we set
                # _defeated=True if we found at least one overlapping crab:
                self._defeated = True
   @staticmethod
   def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                        1 + strength * 4,
                         1 + strength * 9,
                         1 + strength * 24 )
        result.turn(random.randrange(360))
        return result
   @staticmethod
   def _create_random_worm():
        return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
   def update(self):
        if self._defeated or self._victorious: # gameover
           #sys.exit()
           pass
   def draw(self):
        if self._defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
        if self. victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
```

9.8 clock 73

```
class Crab(GameObj):
    def __init__(self, pos):
        self.image = "crab"
        self.pos = pos
        self.speed = 0
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
class Lobster(GameObj):
   def __init__(self, pos, speed, drunkenness, jumpiness):
    self.image = "lobster"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
        else:
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        self.stage.take_out_neighbouring_crab(self)
class Worm(GameObj):
   def __init__(self, pos):
    self.image = "worm"
        self.pos = pos
    def act(self):
        new_pos = (self.pos[0] + random.uniform(-0.5, 0.5),
                    self.pos[1] + random.uniform(-0.5, 0.5))
        if not self.stage.is_beyond_edge(new_pos):
            self.pos = new_pos
beach = Beach()
beach.show()
pgzrun.go()
```

9.8 clock

Wir hatten den <code>sys.exit()</code>-Befehl auskommentiert. Nun wollen wir das Spiel aber tatsächlich beenden. Allerdings nicht abrupt, sondern nach einer angemessenen Pause von etwa 4 Sekunden. Für die verzögerte Ausführung von Anweisungen gibt es in Pygame Zero das <code>clock</code>-Objekt. Dazu kommen wir gleich.

Zunächst wollen wir jedoch Vorkehrungen treffen, dass innerhalb dieser 4 Sekunden ein bereits gewonnenes Spiel nicht doch noch verloren wird. Dies lösen wir, indem wir im Falle eines Siegs sofort alle Hummer von der Bühne werfen. Dann können diese nämlich keinen Schaden mehr anrichten:

74 9.8 clock

Die Methode leave_all der Klasse Stage verweist alle Spielobjekte der als Parameter angegebenen Klasse von der Bühne.

Nun wollen wir sowohl im Falle eines Sieges als auch im Falle einer Niederlage das Spiel verzögert beenden:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(Worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
            self._victorious = self.count_game_objects(Worm) == 0
            if self._victorious:
                self.leave_all(Lobster)
                self.schedule_gameover()
def take_out_neighbouring_crab(self, lobster):
    for c in self.get_game_objects(Crab):
        if c.overlaps(lobster):
            c.leave_stage()
            sounds.au.play()
            # There is only one crab on the beach, so we set
            # _defeated=True if we found at least one overlapping crab:
            self. defeated = True
            self.schedule_gameover()
def schedule_gameover(self):
    clock.schedule_unique(sys.exit, 4.0)
```

Die Methode schedule_gameover nutzt das clock-Objekt und dessen schedule_unique-Methode. Der erste Parameter ist die Funktion, die verzögert ausgeführt werden soll, der zweite Parameter gibt die Sekunden der Verzögerung an. Mit _unique ist gemeint, dass die angegebene Funktion sys.exit einmal ausgeführt werden soll (und nicht etwa wiederholt alle 4 Sekunden). Mehr Details zu schedule_unique gibt es in der Online-Dokumentation³⁶ von Pygame Zero.

Die update-Methode der Beach-Klasse ist nun vollkommen überflüssig geworden und können wir entfernen (was wir auch tun).

Die Verwendung des clock-Objekts birgt einige Tücken. Wenn man den Aufruf der schedule_unique-Methode an einer Stelle programmiert, wo er wieder und wieder aus dem Gameloop heraus aufgerufen wird, dann verlängert jeder Aufruf die Frist. Wir könnten auf die Idee kommen, die Methode take_out_neighbouring_worms wie folgt abzuändern:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
        self._victorious = self.count_game_objects(worm) == 0
    if self._victorious:
        self.leave_all(Lobster)
        self.schedule_gameover()
```

Die Idee hinter dieser Änderung ist eine Effizienzverbesserung. Die Detektion eines Sieges (Würmer zählen) kostet CPU-Leistung. Da reicht es doch, das einmal am Ende der Methode take_out_neighbouring_worms zu machen, anstatt es bei jedem gefressenen Wurm zu wiederholen. Wir

³⁶ http://pygame-zero.readthedocs.io/en/stable/builtins.html#clock

haben daher den gesamten Code zur Siegdetektion und die Reaktion darauf aus der in der for-Schleife geschachtelten if-Anweisung herausgezogen. Dadurch wird die Siegdetektion nur einmal nach vollständig abgearbeiteter for-Schleife durchgeführt. Das ist effizienter, aber nun funktioniert das Programmende nicht mehr. Das Fenster schließt sich nach dem letzten Wurm gar nicht. Nicht sofort, nicht nach 4 Sekunden, nie. Die Ursache hatten wir schon erwähnt. Durch unsere Änderung wird, nachdem der letzte Wurm den Strand verlassen hat, die Methode self.schedule_gameover() fortwährend und immer wieder 60 Mal pro Sekunde ausgeführt. Denn der Wert von self._victorious ist True und bleibt True. Und dadurch wird der Aufruf von sys.exit() immer wieder neu für 4 Sekunden in die Zukunft verschoben. Wir schieben sozusagen den geplanten Aufruf immer wieder vor uns her und erreichen ihn nie. Wie der Esel, der die einen Meter vor ihm baumelnde Möhre nie erreichen wird, werden auch wir den stets 4 Sekunden vor uns baumelnden sys.exit()-Aufruf nie erleben.

Wir korrigieren dies wie folgt:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
        if not self._victorious and self.count_game_objects(worm) == 0:
            self._victorious = True
        self.leave_all(Lobster)
        self.schedule_gameover()
```

Dadurch, dass wir in die Bedingung der if-Anweisung den Ausdruck not self._victorious aufnehmen, wird die gesamte Bedingung nur True, so lange das Spiel noch nicht gewonnen ist. Genau in dem Moment, in dem der letzte Wurm vertilgt wurde, gelangen wir das erste Mal in den Anweisungsteil der if-Anweisung. Dort setzen wir self._victorious = True und bewirken hierdurch, dass die if-Bedingung bei dem in wenigen Millisekunden stattfindenden, nächsten Gameloop-Durchlauf False ist und False bleibt. Im Endeffekt wird also self.schedule_gameover() genau einmal ausgeführt. Wir haben uns elegant aus der misslichen Esel-Lage befreit ©.

Erwähnenswert ist noch, wie die Bedingung not self._victorious and self.count_game_objects(worm) == 0 ausgewertet wird. Wirkt das not auf den gesamten Ausdruck oder nur auf den ersten Teil. Die Regel: not bindet stärker als and und and bindet stärker als or. Ein paar Beispiele zur Illustration:

Ausdruck	ist gleichbedeutend mit						
not a and b	(not a) and b						
a and b or c and	(a and b) or (c and d)						
a and not b or c	(a and (not b)) or c						
not a or b and c	(not a) or (b and c)						

Die Regeln sind vergleichbar mit der aus dem Mathematikunterricht bekannten Regel "Punktrechnung vor Strichrechnung". Das and entspricht der Multiplikation, das or der Addition. Und das not entspricht der Verkehrung einer Zahl ins Negative durch ein voran gestelltes Minuszeichen.

9.9 Neustart statt Spielende

Vielleicht wollen Sie nach einem Sieg lieber ein neues Spiel wagen? Kein Problem. Wir können ja einfach eine andere Funktion zur Ausführung in 4 Sekunden planen. Einzige Voraussetzung: die Funktion darf keine Parameter verlangen. Wir planen die Ausführung einer eigenen Methode restart ein:

9.10 Zusammenfassung

```
def restart(self):
    if self._victorious:
        self.leave_all()
        self.__init__()
    elif self._defeated:
        sys.exit()
    else:
        raise Exception("restart on unfinished game detected!")

def schedule_gameover(self):
    clock.schedule_unique(self.restart, 4.0)
```

Die Methode restart prüft zunächst, ob es sich um einen Sieg oder eine Niederlage handelt. Hier wird ein neues Konstrukt verwendet: elif. Dieses bedeutet so viel wie "else if", d. h. die hinter elif stehende Bedingung wird nur ausgewertet, wenn die erste if-Bedingung unwahr war. Und dann ist das elif wie ein ganz normales if zu verarbeiten. Man kann kaskadierend sehr viele elif-Anweisungen untereinander schachteln. Am Ende kann dann optional noch ein else-Zweig folgen. So auch in unserem Fall. Im Fall, dass weder Sieg noch Niederlage vorliegt, erzeugen wir einen Programmabbruch. Denn da ist ja offenbar etwas nicht mit rechten Dingen zugegangen. Die Anweisung raise erzeugte eine sogenannte Exception – einen Ausnahmefall. Die zum Fehler gehörende Nachricht geben wir als Text hinter dem erzeugten Exception-Objekt an.

Falls alles mit rechten Dingen zugegangen ist, wird im Falle einer Niederlage einfach sys.exit() aufgerufen, um das Programm zu beenden. Im Falle eines Sieges bereinigen wir die Bühne um alle verbliebenen Akteure (leave_all) und initialisieren die Bühne neu (__init__). Der __init__-Aufruf bevölkert die Bühne wie gehabt mit einer neuen Krabbe und mehreren neuen Würmern und Hummern.

Um einmal zu prüfen, wie sich die raise-Anweisung auswirkt, programmieren wir vorübergehend einen verbotenen Aufruf der restart-Methode:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
            <u>self.restart() # nur um einmal die Auswirkungen von raise zu testen</u>
    if not self._victorious and self.count_game_objects(Worm) == 0:
        self._victorious = True
        self.leave_all(Lobster)
        self.schedule_gameover()
```

Wir haben einen Programmierfehler eingebaut. Schon beim ersten gefressenen Wurm rufen wir direkt self.restart() auf. Zur Laufzeit bricht das Programm beim ersten gefressenen Wurm mit der folgenden Meldung auf der Console ab:

```
raise Exception("restart on unfinished game detected!")
Exception: restart on unfinished game detected!
```

Nachdem wir nun die Auswirkungen erfahren haben und getestet haben, dass die Methode restart erfolgreich prüfen kann, wenn etwas nicht mit rechten Dingen zugeht, werfen wir die zuletzt hinzugefügte Programmzeile wieder hinaus.

9.10 Zusammenfassung

Wir haben in diesem Kapitel mehrfach Code zwischen Methoden hin und her geschoben, ohne die Funktion tatsächlich zu verändern. Unser Ziel bei den Verschiebereien war, die Effizienz und die Übersichtlichkeit des Programms zu verbessern. Solche Veränderungen, die keine Funktionsänderungen bewirken, sondern eher für unsichtbare Qualitätsverbesserungen sorgen, nennt man **Refactoring**.

In diesem Abschnitt haben wir eine Beach-Klasse eingeführt. Dieses neue Konzept hat uns zum Thema der Kapselung gebracht. Wir haben diskutiert, welche Klasse für welche Daten verantwortlich ist und diese "besitzt". Wir haben gelernt, dass privater Besitz einer Python-Klasse konventionsgemäß mit einem einleitenden Unterstrich gekennzeichnet wird. Und wir haben gelernt, dass es höflich ist, den Besitz anderer Klassen zu respektieren, auch wenn wir wissen, dass der Python-Interpreter auf einer solchen

9.10 Zusammenfassung 77

Höflichkeit nicht besteht. D. h. wenn wir Code schreiben, der direkt auf in fremden Klassen gekapselte Daten zugreift, wird der Code funktionieren, aber er widerspricht der Konvention.

Wir haben versucht, die Effizienz unseres Programms zu verbessern, indem wir Programmcode von der Bühne in einzelne Spielobjekte verschoben haben. Das war zwar der Effizienz zuträglich, nicht jedoch der Kapselung. Am Ende haben wir eine Version geschaffen, in der beides vorhanden ist: Effizienz und Kapselung. Dabei haben wir ein neues Gesetz berücksichtigt: das **Gesetz von Demeter**. Wir haben zudem darauf geachtet, dass jede Klasse **kohärent** ist, d. h. in sich stimmig und zusammenhängend. Salopp formuliert meint Kohärenz: "alles drin, nicht zu wenig und nicht zu viel". Kohärenz ist die Schwester der **losen Kopplung**, die besagt, dass eine Klasse möglichst lockere Bindungen mit anderen Klassen eingehen soll. Salopp formuliert besagt die lose Kopplung, dass eine Klasse "nichts von dem Besitz einer anderen Klasse weiß, was sie nichts angeht".

Zwischendurch haben wir bemerkt, dass ein Unterschied zwischen sog. lokalen Variablen einerseits und Objekt-Attributen andererseits besteht. Lokale Variablen sind nur innerhalb einer Funktion verwendbar. Objekt-Attribute werden als Zustand in dem Objekt dauerhaft gespeichert. Die in Objekt-Attributen gespeicherten Daten können daher von mehreren Methoden derselben Klasse gemeinschaftlich genutzt werden. Nicht zuletzt wurden sog. statische Methoden eingeführt, um zur Klasse gehörende Hilfsroutinen zu programmieren und dabei klar auszudrücken, dass diese unabhängig von den in den Objekt-Attributen gespeicherten Daten funktionieren.

Nebenbei haben wir logische Operatoren eingeführt und die elif-Anweisung eingesetzt. Eine neue Anweisung pass wurde alternativ oder begleitend zu auskommentierten Anweisungen genutzt. Wir wissen, wie man ein laufendes Programm mit sys.exit() beendet und wir haben das clock-Objekt eingesetzt, um zukünftig auszuführende Anweisungen zu planen. Zudem wissen wir, wie man Text in verschiedenen Größen und Schriftarten auf der Bühne ausgeben kann. Dieses Wissen wird uns im nächsten Kapitel helfen, in dem wir uns als erstes den aktuellen Punktestand vornehmen. Ganz zum Schluss haben wir in das Exception-Konzept hinein geschnuppert, um ein Programm im Ausnahmefall mit einer Fehlermeldung abzubrechen.

78 10.1 Score

10 Spielstand

10.1 Score

Wir möchten die Anzahl bereits gefressener Würmer auf dem Fenster anzeigen. Dazu bietet sich die Methode draw in der Klasse Beach an. Nehmen wir an, wir hätten den aktuellen Score bereits in der Strandbühne in einem Attribut gespeichert. Dann ist die folgende Methode geeignet, den Score anzuzeigen:

Die neue Anweisung übergibt als ersten Parameter den Wert "Score: " + str(self._score) an die Methode screen.draw.text. Es handelt sich dabei um einen Ausdruck, der von Python zur Laufzeit ausgewertet wird, bevor das berechnete Ergebnis an die Methode screen.draw.text übergeben wird. Mit einem + Zeichen lassen sich zwei Zeichenketten in Python zu einer längeren Zeichenkette verknüpfen. Wichtig: sowohl der linke als auch der rechte Operand müssen Zeichenketten sein³⁷. Da self._score ein Zahlwert sein wird, wandeln wir diesen zuerst durch str(self._score) in eine Zeichenkette um.

Fehlt noch das Attribut <u>score</u>, das wir in der Initialisierungsroutine der Strandbühne erst einmal auf einen Anfangswert setzen:

```
class Beach(Stage):
    def __init__(self):
        self._defeated = False
        self._victorious = False
        <u>self._score = 0</u>
        self.background_image = "sand"
        ... Rest wie bisher ...
```

Wir haben uns für ein Attribut mit führendem Unterstrich entschieden, um zu signalisieren, dass die Strandbühne den Punktestand im Sinne der Kapselung für ihren Privatbesitz hält.

Abbildung 32 zeigt, wie das Ergebnis aussieht.



Abbildung 32: Anzeige des Punktestands

³⁷ Darin unterscheidet sich Pythons String-Verkettungs-Operator von entsprechenden Operatoren anderer Programmiersprachen. Z. B. kann man in Java auch eine Zeichenkette und eine Zahl verketten. In Python muss man Zahl-Werte vorher in einen String umwandeln.

10.2 Umwandlung zwischen Zeichenketten und Zahlen

Genau genommen handelt es sich bei str(self._score) gar nicht um eine Umwandlungsfunktion. Alle Zeichenketten in einem Python-Programm sind Objekte. Diese Objekte sind Objekte der in Python eingebauten Klasse namens str. Wenn wir schreiben str(17), dann erzeugt Python ein str-Objekt und ruft anschließend dessen __init__-Methode auf. Die __init__-Methode von str untersucht den erhaltenen Parameter und entscheidet dann, da es sich um einen Zahlwert handelt, aus dem Zahlwert 17 die Zeichenfolge "17" zu generieren.

Man kann übrigens auch Zeichenketten in Zahlen "umwandeln". Probieren Sie doch einmal folgendes in der IDLE-Shell aus:

```
>>> text = "16"
>>> zahl = 5
>>> print(text + str(zahl))
165
>>> print(int(text) + zahl)
21
```

Beim ersten print wurden zwei Zeichenketten mit + verknüpft. Beim zweiten print wurde der text vorher in eine ganze Zahl "umgewandelt" und dann zwei Zahlen addiert.

Entsprechendes geht auch mit float-Zahlen. Dabei denken wir daran, dass Fließkommazahlen immer mit einem Dezimalpunkt (kein Komma) dargestellt werden:

```
>>> pi = float("3.1415927")
>>> r = 4
>>> print( pi * r**2 )
50.2654832
```

Die Variable pi bekommt hier den float-Wert 3.1415927, da die Zeichenkette explizit in ein float-Objekt konvertiert wird. Mit dem float-Wert in pi kann man dann die Kreisfläche eines Kreises mit Radius 5 ausrechnen. Hier haben wir Ihnen auch gleich einen weiteren von Python unterstützten Operator verraten: ** bezeichnet die mathematische Potenz. Der Ausdruck r**2 wird zur Laufzeit als das Quadrat des in r gespeicherten Wertes ausgewertet.

Wo wir schon bei neuen Operatoren sind:

```
>>> zwoelf = "12"
>>> fuenf = 5
>>> print( zwoelf * fuenf )
1212121212
>>> print( int(zwoelf) * fuenf )
60
```

Gibt man dem Multiplikationsoperator * als linken Operanden eine Zeichenkette, so wird diese mehrfach aneinander gehängt. Das Ergebnis ist wieder eine Zeichenkette. Wandelt man den linken Operanden hingegen vorher in eine Zahl um, wird ein Produkt zweier Zahlen ausgerechnet.

Man kann den Typ eines Objekts übrigens auch einfach mit type(...) abfragen:

```
>>> euler = 2.7182818
>>> zwei = 2
>>> text = "hello"
>>> print(type(euler))
<class 'float'>
>>> print(type(zwei))
<class 'int'>
>>> print(type(text))
<class 'str'>
```

Wahrheitswerte in Python sind Objekte des Typs bool:

```
>>> python_rocks = True
>>> print(type(python_rocks))
<class 'bool'>
```

10.3 Score bei jedem gefressenen Wurm erhöhen

Den Score-Wert nun bei jedem gefressenen Wurm um 1 zu erhöhen ist nicht schwer:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(worm):
        if w.overlaps(crab):
            w.leave_stage()
            sounds.blop.play()
            self._score += 1

if not self._victorious and self.count_game_objects(worm) == 0:
        self._victorious = True
        self.leave_all(Lobster)
        self.schedule_gameover()
```

Wir haben die Kurzschreibweise gewählt. Gleichbedeutend, nur etwas länger, hätten wir schreiben können:

```
self._score = self._score + 1
```

Das war es schon. Nun wird die Anzahl der gefressenen Würmer links oben angezeigt.

10.4 Spiellevel

Für den Fall eines gewonnen Spiels hatten wir einen Neustart des Spiels programmiert. Wir wäre es, wenn bei jedem gewonnenen Spiel das Spiel etwas schwerer würde. Z. B. könnten wir die Anzahl der Hummer erhöhen. Wir wollen ganz leicht mit einem einzigen Hummer beginnen:

```
class Beach(Stage):
    def __init__(self):
        self._defeated = False
        self._victorious = False
        self._score = 0
        self._level = 1
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(20):
            w = Beach._create_random_worm()
            w.appear_on_stage(self)
        for dummy in range(self._level):
            lobster = Beach._create_random_lobster(0.2)
                                                          # strength 0.2
            lobster.appear_on_stage(self)
```

Im vorstehenden Code haben wir ein neues Zustandsattribut self._level hinzugefügt. Dieses bekommt zu Beginn den Wert 1. In der for-Schleife, in der wir die Hummer generieren, haben wir als obere Grenze genau diesen Wert self._level angegeben. D. h. beim ersten Spielstart sollten wir genau einen Hummer als Gegner bekommen. Und das funktioniert auch.

Nun programmieren wir die Erhöhung des Spiellevels um 1:

10.4 Spiellevel 81

```
def restart(self):
    if self._victorious:
        self.leave_all()
        <u>self._level += 1</u>
        self.__init__()
    elif self.__defeated:
        sys.exit()
    else:
        raise Exception("restart on unfinished game detected!")
```

Kurz bevor der nächste Level beginnt, erhöhen wir den Wert des Zustandsattributes um 1. Nur: es funktioniert nicht. Wenn wir mit der Krabbe den Strand einmal abgeräumt haben, kommt im nächsten Level wieder nur ein Hummer.

Wir geben uns den Spiellevel einmal unterhalb des Scores aus:

Das Ergebnis ist in Abbildung 33 dargestellt. Und die Anzeige belegt, dass nach der ersten Runde, weiterhin "Level: 1" angezeigt wird. D. h., der Wert des Zustandsattributes self._level scheint sich nicht zu ändern.



Abbildung 33: Anzeige des Spiellevels

Tatsächlich ändert sich der Wert des Zustandsattributes self._level auf den Wert 2, aber nur ganz kurz. Unmittelbar nach der Erhöhung des Wertes rufen wir nämlich self.__init__() auf. Und die Initialisierungroutine überschreibt das Zustandsattribut wieder mit dem Startwert 1.

Man kann dieses Problem auf verschiedene Weise lösen. Man könnte z. B. vor dem Aufruf von self.__init__() den aktuellen Level-Wert in einer lokalen Variablen sichern und nach Beendigung von self.__init__() den gesicherten Wert reaktivieren:

```
def restart(self):
    if self._victorious:
        self.leave_all()
        memorize_level = self._level
        self._init__()
        self._level = memorize_level + 1
    elif self._defeated:
        sys.exit()
    else:
        raise Exception("restart on unfinished game detected!")
```

Nun wird die Levelanzeige tatsächlich brav nach jedem abgeräumten Strand um 1 hochgezählt. Die Anzahl der Hummer bleibt aber bei 1, weil die <u>__init__</u>-Methode verständlicherweise von dem gesicherten Wert memorize_level nichts mitbekommt.

So funktioniert es also offenbar nicht. Wir nehmen noch einmal die alte Variante von restart:

```
def restart(self):
    if self._victorious:
        self.leave_all()
        <u>self._level += 1</u>
        self.__init__()
    elif self._defeated:
        sys.exit()
    else:
        raise Exception("restart on unfinished game detected!")
```

Und nun programmieren wir in der <u>__init__</u>-Methode eine Ergänzung, die verhindert, dass ein bereits vorhandener Levelwert irrtümlich überschrieben wird:

```
def __init__(self):
    self._defeated = False
    self._victorious = False
    self._score = 0
    if not hasattr(self, "_level"):
        self._level = 1
    self.background_image = "sand"

    crab = Crab((WIDTH / 2, HEIGHT * 0.7))
    crab.appear_on_stage(self)

for dummy in range(20):
    w = Beach._create_random_worm()
    w.appear_on_stage(self)

for dummy in range(self._level):
    lobster = Beach._create_random_lobster(0.2) # strength 0.2
    lobster.appear_on_stage(self)
```

Mit der in Python eingebauten Funktion hasattr(object, name) kann man überprüfen, ob das als erster object-Parameter übergebene Objekt ein Attribut besitzt, das wie der zweite name-Parameter benannt ist. Wenn wir das Spiel komplett neu starten, gibt es in dem Strandbühnenobjekt noch kein _level-Attribut. In diesem Fall liefert hasattr den Wert False. Da wir den Wert mit not negiert haben, wird in diesem Fall die Initialisierung self._level = 1 ausgeführt. Wenn die Krabbe den Strand einmal erfolgreich abgeräumt hat, erhöht restart den self._level-Wert auf 2. Während der direkt anschließenden __init__-Ausführung liefert der hasattr-Aufruf den Wert True, denn es gibt das Attribut ja bereits. In diesem Fall wird die Initialisierung von self._level also übersprungen. Die for-Schleife weiter unten findet den Wert 2 in self._level vor und erzeugt zwei Hummer. Es funktioniert!

10.5 Das vollständige bisherige Programm

Erneut wollen wir das vollständige bisherige Programm abdrucken:

```
import random
import pgzrun
from pgzo import *
```

```
WIDTH = 560
                # screen width
HEIGHT = 460
                # screen height
TITLE = "Crab" # window title
class Beach(Stage):
    def __init__(self):
        self._defeated = False
        self._victorious = False
        self._score = 0
        if not hasattr(self, "_level"):
            self._level = 1
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(20):
            w = Beach._create_random_worm()
            w.appear_on_stage(self)
        for dummy in range(self._level):
            lobster = Beach._create_random_lobster(0.2) # strength 0.2
            lobster.appear_on_stage(self)
    def take_out_neighbouring_worms(self, crab):
        for w in self.get_game_objects(Worm):
            if w.overlaps(crab):
                w.leave_stage()
                sounds.blop.play()
                self._score += 1
        if not self._victorious and self.count_game_objects(Worm) == 0:
            self._victorious = True
            self.leave_all(Lobster)
            self.schedule_gameover()
    def take_out_neighbouring_crab(self, lobster):
        for c in self.get_game_objects(Crab):
            if c.overlaps(lobster):
                c.leave_stage()
                sounds.au.play()
                # There is only one crab on the beach, so we set
                # _defeated=True if we found at least one overlapping crab:
                self._defeated = True
                self.schedule_gameover()
    def restart(self):
        if self._victorious:
            self.leave_all()
            self._level += 1
        self.__init__()
elif self._defeated:
            sys.exit()
        else:
            raise Exception("restart on unfinished game detected!")
    def schedule_gameover(self):
        clock.schedule_unique(self.restart, 4.0)
    @staticmethod
    def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                         1 + strength * 4,
                         1 + strength * 9,
                         1 + strength * 24)
        result.turn(random.randrange(360))
        return result
    @staticmethod
    def _create_random_worm():
        return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
```

```
def draw(self):
        screen.draw.text("Score: " + str(self._score), topleft= (10,10),
                          color="black", fontsize=20, fontname="zachary")
        screen.draw.text("Level: " + str(self._level), topleft= (10,35),
                         color="black", fontsize=20, fontname="zachary")
        if self._defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                              color="brown", fontsize=60, fontname="zachary")
        if self._victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                              color="brown", fontsize=60, fontname="zachary")
class Crab(GameObj):
    def __init__(self, pos):
        self.image = "crab"
        self.pos = pos
        self.speed = 0
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
        else:
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
class Lobster(GameObj):
    def __init__(self, pos, speed, drunkenness, jumpiness):
    self.image = "lobster"
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
    def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
        else:
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        self.stage.take_out_neighbouring_crab(self)
class Worm(GameObj):
    def __init__(self, pos):
        self.image = "worm"
        self.pos = pos
    def act(self):
        new_pos = (self.pos[0] + random.uniform(-0.5, 0.5),
                   self.pos[1] + random.uniform(-0.5, 0.5))
        if not self.stage.is_beyond_edge(new_pos):
            self.pos = new_pos
beach = Beach()
beach.show()
pgzrun.go()
```

11.1 Daumenkino 85

11 Animierte Spielobjekte

11.1 Daumenkino

Unsere Spielfiguren bewegen sich zurzeit regungslos über den Bildschirm. Wir wollen die Akteure etwas in Bewegung versetzen. Man sagt auch, wir wollen die Spielfiguren **animieren**. Ziel ist es, die Bewegung etwas natürlicher wirken zu lassen. Es soll so aussehen, als würden Hummer und Krabbe ihre Beine bewegen. Und bei den Würmern soll es aussehen, als würden diese sich eben wie Würmer fortbewegen.

Um Animationen zu erstellen braucht man mehrere Bilder ein und derselben Figur, die man wie ein Daumenkino nacheinander anzeigt. Abbildung 34 zeigt sechs Bilder mit verschiedenen Beinstellungen. Wir wollen umlaufend jedes der sechs Bilder anzeigen und dann wieder von vorne beginnen, um so ein Bewegen der Beine zu simulieren.



Abbildung 34: Sechs leicht abgewandelte Bilder der Krabbe mit verschiedenen Stellungen der Beine

Wir kopieren die sechs Bilder crab0.png, crab1.png, crab2.png, crab3.png, crab4.png, crab5.png in das images-Verzeichnis. Das dort bereits abgelegte Bild crab.png benötigen wir nicht mehr, denn es ist identisch mit crab0.png.

11.2 Konstanten

Wir werden gleich häufig testen wollen, wie eine Krabbe in ihrer Bewegung aussieht. Dazu brauchen wir Ruhe und können störende Hummer nicht gebrauchen. Wir machen uns das Testen ein bisschen leichter, indem wir die Anzahl der Gegner im ersten Spiellevel auf o setzen. Hierzu bereiten wir die Klasse Beach in der __init__-Methode so vor, dass die Anzahl der erzeugten Gegner im ersten Level in einer sog. Konstanten definiert werden kann. Eine Konstante ist ein globaler Bezeichner, der auf einen unveränderlichen Wert verweist:

86 11.2 Konstanten

```
import random
import pgzrun
from pgzo import *
WIDTH = 560
               # screen width
HEIGHT = 460
             # screen height
TITLE = "Crab" # window title
START_WITH_LOBSTERS = 1
class Beach(Stage):
   def __init__(self):
       self._defeated = False
       self._victorious = False
       self. score = 0
       if not hasattr(self, "_level"):
           self. level = 1
       self.background_image = "sand"
       crab = Crab((WIDTH / 2, HEIGHT * 0.7))
       crab.appear_on_stage(self)
       for dummy in range(20):
           w = Beach._create_random_worm()
           w.appear_on_stage(self)
       num_lobsters = START_WITH_LOBSTERS + self._level - 1
       for dummy in range(num_lobsters):
            lobster = Beach._create_random_lobster(0.2) # strength 0.2
            lobster.appear_on_stage(self)
```

Im vorstehenden Codeabschnitt lassen wir die for-Schleife für die Hummererzeugung mehrfach durchlaufen. Die Anzahl der Durchläufe berechnen wir unmittelbar davor, indem wir die Variable num_lobsters mit einem Wert belegen. Die Anzahl der Hummer ergibt sich aus dem aktuellen Spiellevel zzgl. Der Anzahl der Hummer im ersten Spiellevel abzgl. 1. Da wir START_WITH_LOBSTERS mit dem Wert 1 initialisieren, verändert sich unser Spiel nicht: wir starten mit einem Hummer und bekommen in jedem Level einen weiteren Hummer dazu.

Das besondere an START_WITH_LOBSTERS ist, dass wir diesen Bezeichner nicht als lokale Variable, sondern ganz oben im Programm definiert haben. Dieser Bezeichner ist für das gesamte Programm sichtbar. Da der Bezeichner ganz oben steht, ist es leicht, diesen zu finden und zu ändern, wenn wir die Gegnerzahl z. B. für Tests verändern wollen. START_WITH_LOBSTERS ist ein globaler Bezeichner.

Es wird i. a. empfohlen, globale Bezeichner zu vermeiden. Ausnahme: **globale Konstanten**. Wenn START_WITH_LOBSTERS eine Variable wäre, könnte im Prinzip jede Funktion in unserem Programm den Wert verändern. Der Vorteil, dass wir über diesen Wert die Anzahl der Start-Hummer übersichtlich an einer Stelle regeln können, wäre dahin. In Python werden Konstanten konventionsgemäß mit Großbuchstaben geschrieben. Tatsächlich verhindert der Python-Interpreter nicht, dass irgendeine Funktion den Wert verändert. Immerhin macht es Python einer solch unhöflichen Funktion aber immerhin schwer. Wir versuchen einmal vorübergehend, unhöflich zu sein:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(Worm):
        if w.overlaps(crab):
            START_WITH_LOBSTERS = 5  # unhöflich, aber in dieser Form wirkungslos
            w.leave_stage()
            sounds.blop.play()
            self._score += 1
    if not self._victorious and self.count_game_objects(Worm) == 0:
            self._victorious = True
            self.leave_all(Lobster)
            self.schedule_gameover()
```

Wir versuchen, die Anzahl der Hummer beim Fressen eines Wurms zu manipulieren, und erwarten nun, dass beim Start des nächsten Spiellevels nicht zwei, sondern sechs Hummer erscheinen. Dem ist aber nicht so. Es erscheinen brav zwei Hummer. Der Grund: Python erfordert bei schreibendem Zugriff auf

11.3 Bildzähler 87

globale Bezeichner, dass explizit davor deklariert wird, dass es sich um einen globalen Bezeichner handelt. Lässt man die Deklaration weg, geht Python davon aus, dass START_WITH_LOBSTERS eine neue lokale Variable in take_out_neighbouring_worms ist, die mit dem globalen Bezeichner nichts zu tun hat. Da wir nichts deklariert haben, hat die Zuweisung auf den Wert 5 keine Auswirkungen auf den globalen Bezeichner.

Manchmal mag es notwendig sein, globale Bezeichner in ihrem Wert zu verändern. Deshalb sei hier verraten, wie man dies in Python bewerkstelligt. Gleich dazu aber eine Warnung: in 99% aller Fälle, brauchen Sie das nicht. Veränderung von Werten globaler Bezeichner ist ein Zeichen schlechten Programmierstils:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(worm):
        if w.overlaps(crab):
            global START_WITH_LOBSTERS
            START_WITH_LOBSTERS = 5  # unhöflich, und in dieser Form wirksam
            w.leave_stage()
            sounds.blop.play()
            self._score += 1
    if not self._victorious and self.count_game_objects(worm) == 0:
            self._victorious = True
            self.leave_all(Lobster)
            self.schedule_gameover()
```

So, nachdem wir das kurz getestet haben, bauen wir die beiden unhöflichen Zeilen aus take_out_neighbouring_worms schnell wieder aus.

Wir wollen noch zwei weitere globale Konstanten definieren, um die Schwierigkeit des Spiels zu steuern:

```
# "Schaltzentrale" für die Schwierigkeit des Spiels:
START_WITH_LOBSTERS = 1
START_LOBSTER_STRENGTH = 0.2
START_WITH_WORMS = 20
class Beach(Stage):
    def __init__(self):
        self._defeated = False
        self._victorious = False
        self._score = 0
        if not hasattr(self, "_level"):
            self._level = 1
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(<a href="START_WITH_WORMS">START_WITH_WORMS</a>):
            w = Beach._create_random_worm()
            w.appear_on_stage(self)
        num_lobsters = START_WITH_LOBSTERS + self._level - 1
        for dummy in range(num_lobsters):
            lobster = Beach._create_random_lobster(START_LOBSTER_STRENGTH)
            lobster.appear_on_stage(self)
```

Der Vorteil dieser Variante ist, dass wir nun an einer Stell am Anfang des Programms eine Art "Schaltzentrale" installiert haben, an der wir über die Schwierigkeit des ersten Levels entscheiden können. Im Moment ändert sich die Anzahl der Würmer pro Level nicht. Das können Sie, wenn Sie Lust haben, noch in unser Programm einbauen.

11.3 Bildzähler

Das erste Bild ist das Bild mit der Ziffer o. Wir programmieren einen neuen Zähler image_cnt in der Klasse Crab, der immer dann hochgezählt wird, wenn das nächste Bild angezeigt werden soll:

88 11.4 Streckenzähler

```
class Crab(GameObj):

def __init__(self, pos):
    self.image = "crab0"
    self.image_cnt = 0
    self.pos = pos
    self.speed = 0
```

Im ersten Versuch programmieren wir in der act-Methode das Hochzählen des Zählers. Bei jedem Aufruf der act-Methode soll der Zähler erhöht und das Bild ausgetauscht werden:

```
def act(self):
    self.switch_image()
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.speed = 0
    self.stage.take_out_neighbouring_worms(self)
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
```

Wir haben uns entschieden, die Funktion des Hochzählens und des Bildaustauschens in eine separate Funktion zu verschieben (Stichwort: prozedurale Zerlegung). Die separate Funktion könnten wir wie folgt programmieren:

```
def switch_image(self):
    self.image_cnt += 1
    if self.image_cnt == 6:
        self.image_cnt = 0
    self.image = "crab" + str(self.image_cnt)
```

Der Zähler wird um 1 erhöht. Wenn der Zähler den maximal gültigen Index überschritten hat (if-Anweisung), setzen wir ihn wieder auf o. Schließlich "basteln" wir den Namen des Bildes aus dem Text "crab" und dem in eine Zeichenkette konvertierten Zählerwert (str(self.image_cnt)) zusammen.

Bevor wir das Programm starten, manipulieren wir in unsere "Schaltzentrale" die Schwierigkeit. Die Anzahl der Hummer im ersten Level setzen wir auf o:

```
START_WITH_LOBSTERS = 0
```

Wenn wir dieses Programm starten, sehen wir eine nervös zitternde Krabbe, die sogar dann zittert, wenn wir sie gar nicht bewegen. Ein Teilerfolg also. Wir haben offenbar erfolgreich einen Bildaustausch programmiert. Nur findet dieser zu häufig statt.

11.4 Streckenzähler

Abhängig von der Geschwindigkeit der Krabbe muss der Bildzähler häufiger oder seltener hochgezählt werden. Wir müssen also mitzählen, wie weit die Krabbe seit dem letzten Bildaustausch gelaufen ist. Wir implementieren also einen weiteren Zähler traveled, diesmal für die zurück gelegte Strecke:

```
class Crab(GameObj):

    def __init__(self, pos):
        self.image = "crab0"
        self.image_cnt = 0
        self.pos = pos
        self.speed = 0
        self.traveled = 0
```

Der neue Zähler muss in der act-Methode erhöht werden, und zwar immer dann, wenn die Krabbe move aufruft. Bei dieser Implementierung gehen wir davon aus, dass eine Drehung (turn) nicht als Fortbewegung im Sinne der Animation gilt:

11.5 Innere Werte

```
def act(self):
    self.switch_image()
    if self.can_move(self.speed):
        self.move(self.speed)
        self.traveled += abs(self.speed)

else:
        self.speed = 0
    self.stage.take_out_neighbouring_worms(self)
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
```

Beachten Sie, dass wir nicht einfach den Wert von self.speed addiert haben, sondern den Absolutbetrag davon. Da unsere Krabbe auch rückwärts laufen kann und wir sie dabei genauso animieren wollen, müssen wir auf negative Werte von self.speed vorbereitet sein. Die in Python eingebaute Funktion abs liefert für einen negativen self.speed-Wert den entsprechend negierten, also positiven Wert. Wenn self.speed sowieso schon positiv ist, liefert abs den positiven Wert.

Wir müssen nun in der neuen Methode switch_image prüfen, ob die bisher zurück gelegte Strecke ausreicht, um einen nächsten Bildwechsel zu rechtfertigen:

```
def switch_image(self):
    if self.traveled > 5:
    self.image_cnt += 1
    if self.image_cnt == 6:
        self.image_cnt = 0
    self.image = "crab" + str(self.image_cnt)
    self.traveled -= 5
```

Wir prüfen, ob die Krabbe mindestens 5 Pixel weit gelaufen ist. Falls ja, führen wir den nächsten Bildwechsel durch. Dann am Ende reduzieren wir die bisher zurück gelegte Strecke um den Wert 5. Damit "verbrauchen" wir gewissermaßen je 5 Längeneinheiten der zurück gelegten Strecke für jeden Bildwechsel. Wir gehen davon aus, dass die Krabbe nie mehr als 5 Längeneinheiten pro Gameloop-Zyklus zurücklegt. Insofern können wir sicher sein, dass nach dem Verlassen der Methode switch_image der Wert von self.traveled wieder kleiner als 5 ist.

Das Programm funktioniert. Die Krabbe bewegt ihre Beine, und zwar sowohl im Vorwärtsgang als auch im Rückwärtsgang.

11.5 Innere Werte

Wir wollen die "inneren Werte" unseres Programms noch etwas verbessern. Das Hochzählen des Bildzählers und die if-Anweisung zur Prüfung, ob wir zu weit hochgezählt haben, kann man etwas eleganter programmieren. Statt:

```
self.image_cnt += 1
if self.image_cnt == 6:
    self.image_cnt = 0
```

einfach:

```
self.image_cnt = (self.image_cnt + 1) % 6
```

Hier haben wir einen neuen Operator benutzt, den Modulo-Operator 🖔. Dieser Operator dividiert den linken Operanden durch den rechten Operanden und liefert als Ergebnis den Rest der Division. In der folgenden Tabelle sind einige Beispielwerte angegeben, die bei Berechnungen mit dem Modulo-Operator entstehen:

90 11.5 Innere Werte

Bei einem Wert von a =	0	1	2	3	4	5	6	7	8	•••
Ergibt sich für a % 6 der	0	1	2	3	4	5	0	1	2	

Zum zweiten gefällt uns nicht, dass unser Programm an mehreren Stellen die gleiche Zahl 5 enthält:

```
def switch_image(self):
    if self.traveled > 5
        self.image_cnt = (self.image_cnt + 1) % 6
        self.image = "crab" + str(self.image_cnt)
        self.traveled -= 5
```

An beiden Stellen besitzt diese Zahl dieselbe Bedeutung, nämlich die Strecke, die eine Krabbe zurück legen muss, bevor ein Bildwechsel stattfinden soll. Solche Zahlwerte (Literale), die an mehreren Stellen im Programm vorkommen, und dabei immer die gleiche Bedeutung haben, nennt man **magische Zahlen**. Magische Zahlen haben zwei Nachteile. Erstens können wir uns vielleicht in zwei Wochen nicht mehr so genau an die Bedeutung der Zahl 5 erinnern. Das ließe sich noch wie folgt beheben:

```
def switch_image(self):
    # Nach je 5 zurückgelegten Längeneinheiten erfolgt ein Bildwechsel
    if self.traveled > 5:
        self.image_cnt = (self.image_cnt + 1) % 6
        self.image = "crab" + str(self.image_cnt)
        self.traveled -= 5
```

Zweitens ist es umständlich, wenn wir die Notwendigkeit verspüren, den Wert zu ändern. Vielleicht haben wir eine neue Bildfolge in einem Zeichenprogramm erstellt, die vielleicht aus 12 Bildern besteht. Dann ist die Strecke pro Bildwechsel vermutlich etwas geringer. Wir müssten nun an mehreren Stellen im Programm die Zahl 5 durch eine andere Zahl austauschen. Wir müssen dabei sehr gut aufpassen, keine Stelle zu übersehen.

Besser wäre es, die Zahl von vorneherein nur einmal im Programm zu haben:

```
def switch_image(self):
    travel_distance_between_image_flips = 5
    if self.traveled > travel_distance_between_image_flips:
        self.image_cnt = (self.image_cnt + 1) % 6
        self.image = "crab" + str(self.image_cnt)
        self.traveled -= travel_distance_between_image_flips
```

Der Kommentar ist jetzt sogar überflüssig geworden. Eine Änderung der Zahl 5 ist nun an genau einer Stelle möglich.

Die Einführung einer Variablen für die magische Zahl ist eine Variante des Programmierprinzipien Single Source Of Truth (SSOT) und Don't Repeat Yourself (DRY). Es wird i. a. als schädlich für die Wartbarkeit von Programmen angesehen, wenn gleiche Daten oder gleiche Funktionen mehrfach an verschiedenen Stellen implementiert werden. Es soll einen einzigen "Quell der Wahrheit" geben (bei uns ist der Quell der Wahrheit nun die Zeile travel_distance_between_image_flips = 5) und es soll keine Wiederholungen geben.

Gibt es noch weitere Stellen in unserem Programm, an denen wir SSOT und DRY verletzen? Ja, der Name der Bilddatei "crab" steht an zwei Stellen: einmal in der __init__-Methode und einmal in der __init__-Methode. Wir korrigieren dies zunächst wie folgt:

11.5 Innere Werte

```
class Crab(GameObj):
   def __init__(self, pos):
        self.IMAGE_PREFIX = "crab"
        self.image = self.IMAGE_PREFIX + "0"
       self.image\_cnt = 0
        self.pos = pos
        self.speed = 0
        self.traveled = 0
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
   def switch_image(self):
        travel_distance_between_image_flips = 5
        if self.traveled > travel_distance_between_image_flips:
            self.image_cnt = (self.image_cnt + 1) % 6
            self.image = self.IMAGE_PREFIX + str(self.image_cnt)
            self.traveled -= travel_distance_between_image_flips
```

Da sich die Wiederholung auf zwei verschiedene Methoden erstreckte, reicht eine lokale Variable nicht mehr aus. Wir haben uns daher zunächst für ein Objekt-Attribut self. IMAGE_PREFIX entschieden, in dem wir den Namen der Bilddatei ablegen.

Nehmen wir für einen Moment an, dass es nicht eine Krabbe, sondern hunderte Krabben in einem Spiel gäbe. Jede Krabbe würde dann das Attribut self.IMAGE_PREFIX speichern. Das wäre eine unnötige Speicherplatzverschwendung. Eigentlich würde es reichen, den Bilddateinamen nur einmal pro Klasse zu speichern, statt einmal pro Objekt. Dafür gibt es in Python und auch in vielen anderen Programmiersprachen eine Möglichkeit. Wir verschieben die Initialisierung des Attributs IMAGE_PREFIX einfach "eine Ebene nach außerhalb" in den Klassenrahmen. Dabei lassen wir das self. weg, denn die Initialisierung betrifft ja kein einzelnes Objekt mehr:

92 11.5 Innere Werte

```
class Crab(GameObj):
    IMAGE_PREFIX = "crab"
    def __init__(self, pos):
        self.IMAGE_PREFIX = "crab"
        self.image = selfCrab.IMAGE_PREFIX + "0"
        self.image_cnt = 0
        self.pos = pos
        self.speed = 0
        self.traveled = 0
    def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
    def switch_image(self):
        travel_distance_between_image_flips = 5
        if self.traveled > travel_distance_between_image_flips:
            self.image_cnt = (self.image_cnt + 1) % 6
            self.image = selfCrab.IMAGE_PREFIX + str(self.image_cnt)
            self.traveled -= travel_distance_between_image_flips
```

Wir haben soeben ein sog. **Klassenattribut** geschaffen (auch **statisches Attribut** genannt). Der Zugriff auf das Klassenattribut erfolgt nun durch Voranstellung des Klassennamens³⁸ Crab.IMAGE_PREFIX.

Mit dem neuen Wissen können wir auch die lokale Variable travel_distance_between_image_flips in ein Klassenattribut umwandeln. Denn: es ist unnötig, den Interpreter bei jeder Ausführung der switch_image-Methode immer wieder eine neue lokale Variable anlegen zu lassen. Das reicht, wenn wir das einmal pro Klasse machen:

³⁸ Tatsächlich würde auch weiterhin self. IMAGE_PREFIX funktionieren. Es wird jedoch i. a. davon abgeraten, auf Klassenattribute mittels zuzugreifen, da sonst der Eindruck entstehen könnte, es handele sich um ein Objektattribut.

11.5 Innere Werte

```
class Crab(GameObj):
   IMAGE_PREFIX = "crab'
    IMAGE\_COUNT = 6
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
   def __init__(self, pos):
        self.image = Crab.IMAGE_PREFIX + "0"
        self.image\_index = 0
        self.pos = pos
        self.speed = 0
        self.traveled = 0
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
        else:
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
   def switch_image(self):
        if self.traveled > <u>Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS</u>:
            self.image_index = (self.image_index + 1) % Crab.IMAGE_COUNT
            self.image = Crab.IMAGE_PREFIX + str(self.image_index)
            self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Das neue Klassenattribut TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS ist eine Konstante, deshalb haben wir sie in Großbuchstaben geschrieben.

Wir haben die Gelegenheit genutzt, und noch eine weitere Klassenkonstante IMAGE_COUNT für die Anzahl der Bilder eingeführt. Vor dem Hintergrund des DRY-Prinzips wäre das nicht notwendig gewesen, denn die Zahl 6 kam nur einmal im ursprünglichen Code vor. Es gibt aber natürlich weitere Prinzipien. Eines davon ist, selbsterklärenden Code zu schreiben. Die Zahl 6 erklärt wenig. Der Name der Konstante IMAGE_COUNT erklärt vieles.

Um Verwechselungen zwischen der Klassenkonstante IMAGE_COUNT und dem Objektattribut self.image_cnt zu vermeiden, haben wir das Objektattribut in self.image_index umbenannt. Dies repräsentiert auch besser den Zustand des Krabbenobjekts ("diese Krabbe verweist gerade auf das Bild mit dem Index 3"). Ein Zähler repräsentiert eher den Prozess ("diese Krabbe ist im Zählprozess gerade an der Stelle 3"). Objektattribute repräsentieren den Objektzustand und sollten dies in ihrer Namensgebung auch reflektieren.

Wir sind noch nicht ganz fertig mit der Verbesserung der inneren Werte. Wir haben noch an einer weiteren Stelle das DRY-Prinzip verletzt: sowohl die __init__-Methode als auch die _switch_image_Methode implementieren eine Anweisung zur Zusammensetzung des Bilddateinamenprefix und des Bildindex. Das Wissen darüber, wie aus dem Prefix und dem Index ein gültiger Dateiname wird, ist derzeit also an zwei Stellen im Programm programmiert. Write Everything Twice (WET) ist das Negativ-Prinzip, das wir hier versehentlich beherzigt haben. Das wollen wir ändern:

94 11.5 Innere Werte

```
class Crab(GameObj):
    IMAGE_PREFIX = "crab"
    IMAGE\_COUNT = 6
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
    def __init__(self, pos):
        self.image_index = 0
        self._set_image()
        self.pos = pos
        self.speed = 0
        self.traveled = 0
    def _set_image(self):
        self.image = Crab.IMAGE_PREFIX + str(self.image_index)
    def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
    def switch_image(self):
        if self.traveled > Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self.image_index = (self.image_index + 1) % Crab.IMAGE_COUNT
            self._set_image()
            self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Eine neue Methode _set_image ist nun alleinige Trägerin des Wissens darüber, wie Prefix und Index zu einem Dateinamen verknüpft werden. An den beiden ursprünglichen Stellen steht nun ein einfacher Aufruf der neuen Methode.

Damit unser Programm funktioniert, mussten wir in der __init__-Methode die Anweisung zur Initialisierung von self.image_index an den Anfang der Methode stellen. Sonst führt die Ausführung von self._set_image() zu der Fehlermeldung, dass das Objektattribut image_index nicht existiert. Methoden, die nur dann funktionieren, wenn man vorher vorbereitende Arbeiten erledigt hat, sind durchaus riskant im Einsatz. Wir ändern die neue Methode noch einmal, indem wir ihr einen Parameter überreichen. Das macht den Einsatz sicherer:

11.5 Innere Werte

```
class Crab(GameObj):
   IMAGE_PREFIX = "crab"
   IMAGE\_COUNT = 6
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
   def __init__(self, pos):
        self.image_index = 0
        self._set_image_index(0)
       self.pos = pos
        self.speed = 0
        self.traveled = 0
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Crab.IMAGE_PREFIX + str(self.image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
           self.traveled += abs(self.speed)
        else:
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
       if keyboard.left:
           self.turn(10)
        if keyboard.right:
           self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
   def switch_image(self):
        if self.traveled > Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self.image_index = (self.image_index + 1) % Crab.IMAGE_COUNT
            self._set_image_index((self.image_index + 1) % Crab.IMAGE_COUNT)
            self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Wie Sie erkennen können, haben wir die Methode umbenannt in <u>_set_image_index</u>. Dieser Name passt besser zu dem Parameter, der ja ein Index ist.

11.5.1 Für Fortgeschrittene: Properties

Statt einer privaten _set_...-Methode könnten wir alternativ eine sog. **Property** für image_index realisieren:

96 11.6 Hummer animieren

```
class Crab(GameObj):
   IMAGE_PREFIX = "crab"
   IMAGE\_COUNT = 6
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
   def __init__(self, pos):
        self.image_index = 0
                                # initialize image by index via setter property
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0
        self.traveled = 0
   @property
   def image_index(self):
        return self._image_index
   @image_index.setter
   def image_index(self, image_index):
        self._image_index = image_index
        self.image = Crab.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
           self.move(self.speed)
           self.traveled += abs(self.speed)
        else:
           self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
           self.turn(-10)
        if keyboard.up:
           self.speed_up()
        if keyboard.down:
           self.slow_down()
   def switch_image(self):
        if self.traveled > Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            # The following line effectively induces an image flip via the above setter property:
           self.image_index = (self.image_index + 1) % Crab.IMAGE_COUNT
            self._set_image_index((self.image_index + 1) % Crab.IMAGE_COUNT)
           self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Properties erlauben es, bei einer Zuweisung eines Objektattributs eine Funktion auszulösen, die über die eigentliche Zuweisung hinausgeht. Der Vorteil der Property ist, dass man nicht plötzlich die Syntax eines _set_image_index-Funktionsaufrufs benutzen muss. Man kann weiter mit Zuweisungen arbeiten.

Mehr zu Properties finden Sie z. B. in einem guten einführenden Python-Buch (s. etwa Fußnote 7 auf Seite 6).

11.6 Hummer animieren

Ähnlich, wie wir das für die Krabbe gemacht haben, wollen wir nun den Hummer animieren. Dabei können wir viel von der Krabbe abgucken. Zunächst kopieren wir die Bilder lobster0.png und lobster1.png in das images-Verzeichnis (vgl. Abbildung 35). Die Datei lobster.png ist mit lobster0.png identisch und kann entfernt werden.





Abbildung 35: Zwei leicht abgewandelte Bilder des Hummers

Dann verändern wir den Programmcode der Lobster-Klasse:

11.6 Hummer animieren 97

```
class Lobster(GameObj):
   IMAGE_PREFIX = "lobster"
   IMAGE\_COUNT = 2
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 7
   def __init__(self, pos, speed, drunkenness, jumpiness):
        self._set_image_index(0)
        self.pos = pos
        self.speed = speed
        self.traveled = 0
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Lobster.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        self.stage.take_out_neighbouring_crab(self)
   def switch image(self):
        if self.traveled > Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Lobster.IMAGE_COUNT)
            self.traveled -= Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Wir setzen die globale Konstante START_WITH_LOBSTERS = 1 und testen. Das funktioniert auch so weit. Unterschiede bestehen i. w. beim Dateinamen "lobster" und bei der Anzahl der Bilder (2 statt 6) und der Entfernung zwischen zwei Bildwechseln (7 statt 5). Die viele Code-Kopiererei lässt allerdings ein ungutes Gefühl zurück. Wenn wir ab jetzt irgendwelche Änderungen an der Animationslogik durchführen wollen, müssen wir das immer in beiden Klassen Crab und Lobster durchführen. Das DRY-Prinzip ist hier deutlich verletzt.

Es gäbe nun verschiedene Möglichkeiten, das Prinzip wieder einzuhalten. Denkbar wäre, dass die beiden Klassen Crab und Lobster von einer gemeinsamen Basisklasse erben, die die Animationslogik aufnimmt. Da wir hier in diesem einführenden Buch keine breite Einführung in die Vererbung planen, belassen wir es jedoch bei dem unguten Gefühl.

Für Fortgeschrittene unter Ihnen wollen wir es uns aber nicht nehmen lassen, einen spannenden anderen Ausweg zu beschreiten, der sehr typisch für dynamisch getypte Sprache wie Python ist: wir schreiben eine Funktion animate, die eine Spielfigur als Parameter entgegennimmt, und die die Spielfigur während der Laufzeit um die benötigten zusätzlichen Methoden erweitert. Diesen Ausweg beschreiten wir im nun folgenden Abschnitt.

11.6.1 Für Fortgeschrittene: ein Objekt und dessen Klasse zur Laufzeit dynamisch verändern

In diesem Abschnitt gehen wir so vor, dass ich Ihnen den fertigen Code zuerst zeige, und danach gehen wir den Code Schritt für Schritt durch:

98 11.6 Hummer animieren

```
def animate(gobj, get_image_name_func, image_count, travel_distance_between_image_flips):
   cls = type(gobj)
   if not hasattr(cls, "move") or not callable(cls.move):
        raise Exception("first parameter's type must provide 'move' method")
   # first prepare the class, if not done already
   # overwrite cls's 'move' method with a new one thereby
   # renaming the existing method to 'old_move'.
   # First check, if we have overwritten already in the past:
   if not hasattr(cls, "old_move"):
        # Not overwritten yet.
        # Let's first rename the current move method:
        cls.old_move = cls.move
        # define a new method:
        def new_move(self, distance):
            self.old_move(distance) # call old move method
            # now after moving we perform image flip if needed:
            self.traveled += abs(distance)
            if self.traveled > travel_distance_between_image_flips:
                self._set_image_index((self.image_index + 1) % image_count)
                self.traveled -= travel_distance_between_image_flips
        # overwrite with the new method
        cls.move = new_move
        # add an additional helper method:
        def _set_image_index(self, image_index):
            self.image_index = image_index
            self.image = get_image_name_func(image_index)
        cls._set_image_index = _set_image_index
   gobj.traveled = 0
   gobj._set_image_index(0)
class Crab(GameObj):
   def __init__(self, pos):
       self.image = "crab"
animate(self, lambda i: "crab" + str(i), 6, 5)
       self.pos = pos
        self.speed = 0
   def act(self):
        if self.can_move(self.speed):
            self.move(self.speed)
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
```

Wir vergleichen zuerst einmal die Klasse Crab mit der Klasse Crab wie sie aussah kurz bevor wir die Animation begonnen hatten (abgedruckt in Abschnitt 10.5). Wir haben den Code der Klasse Crab nur in einer Zeile geändert:

```
self.image = "crab"
animate(self, lambda i: "crab" + str(i), 6, 5)
```

Die Bedeutung des Aufrufs der animate-Funktion ist:

- animiere dieses Objekt (self),
- verwende dabei Dateinamen, die durch die folgende Funktion berechnet werden: lambda i: "crab" + str(i),
- verwende insgesamt 6 Bilder,
- wechsle das Bild alle 5 Pixellängeneinheiten.

11.6 Hummer animieren 99

Der Parameterwert lambda i: "crab" + str(i) ist eine anonyme Funktion mit Parameter i und Rückgabewert "crab" + str(i). Mehr zu Lambdas finden Sie z. B. im offiziellen Tutorial³⁹ zu Python. Wir übergeben also ein Funktionsobjekt an die animate-Funktion. Jene wird irgendwann, wenn sie einen Dateinamen generieren möchte, unser übergebenes Funktionsobjekt ausführen. Dies ist eine elegante Möglichkeit, nicht nur Daten, sondern auch Funktionalität an die animate-Funktion zu übergeben.

Wenden wir uns der animate-Funktion zu, die auf den ersten Blick unübersichtlich wirkt. Die Aufgabe von animate ist, das als Parameter erhaltene Objekt gobj mit zusätzlicher Funktionalität auszustatten. Und das soll zur Laufzeit passieren! Die animate-Funktion erwartet als Parameter ein Objekt, das eine move-Methode besitzt. Crab- und Lobster-Objekte erfüllen diese Bedingung, denn diese beiden Klassen sind ja vom Typ Gameobj und darin ist eine move-Methode implementiert. Die ersten drei Zeilen der animate-Funktion prüfen ganz einfach, ob die Erwartung zutrifft. Mit hasattr(cls, "move") wird sichergestellt, dass die Klasse des gobj-Objekts ein Attribut namens "move" besitzt. Mit callable(cls.move) wird sichergestellt, dass es sich bei diesem Attribut um ein Ding handelt, das man aufrufen kann, oder anders formuliert, dass es sich bei diesem Attribut um eine Funktion handelt.

Wenn die ersten drei Zeilen erfolgreich und ohne Exception abgearbeitet wurden, will die animate-Funktion nun mehrere Dinge mit dem gobj-Objekt anstellen:

- 1. Neudefinition der move-Methode des gobj-Objekts. Die alte move-Methode wird in old_move umbenannt.
- 2. Hinzufügen einer neuen _set_image_index-Methode zum gobj-Objekt.
- 3. Hinzufügen eines Streckenzählers traveled zum gobj-Objekt
- 4. Initialisierung des gobj-Objekts durch Aufruf der soeben hinzugefügten _set_image_index-Methode.

Die beiden erstgenannten Schritte betreffen die <u>Klasse</u> des **gobj**-Objekts, da sich alle Objekte derselben Klasse ja die Methoden teilen⁴⁰. Die beiden letztgenannten Schritt betreffen das **gobj**-<u>Objekt</u>.

Falls es mehrere zu animierende Objekte derselben Klasse gibt (bei Lobster wird das der Fall sein), muss animate die beiden erstgenannten Schritte nur einmal beim ersten Lobster-Objekt durchführen. Daher prüft animate mit if not hasattr(cls, "old_move"):, ob die Klasse, der gobj angehört, bereits im Sinne der beiden erstgenannten Schritte präpariert ist. Falls ja, geht es direkt mit dem dritten Schritt weiter. Falls nein, befasst sich animate nun mit der Präparation im Sinne der beiden erstgenannten Schritte. Dabei definiert animate zwei lokale Funktionen new_move und _set_image_index und weist diese als Funktionsobjekte den entsprechenden Klassen-Attributen cls.move und cls._set_image_index zu. Vorher wurde die alte cls.move-Methode noch vor dem Überschreiben gerettet, indem die Referenz auf die alte cls.move-Methode nach cls.old_move kopiert wurde. Der Inhalt der beiden neuen, lokal definierten Funktionen, ähnelt dem Code, den wir ursprünglich direkt in die Klassen Crab und Lobster geschrieben hatten. Zunächst betrachten wir:

Der einzige Unterschied zur ursprünglich direkt in den Klassen Crab und Lobster implementierten Methode besteht im Aufruf von get_image_name_func. Hier versteckt sich also der Aufruf der als Parameter übergebenen anonymen Funktion. Die zweite lokal definierte Funktion ist:

³⁹ https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions

⁴⁰ In Python ist es zwar auch möglich, dass einzelne Objekte Methoden der Klasse ersetzen, aber das wollen wir hier nicht weiter vertiefen.

100 11.7 Würmer animieren

```
def new_move(self, distance):
    self.old_move(distance) # call old move method
    # now after moving we perform image flip if needed:
    self.traveled += abs(distance)
    if self.traveled > travel_distance_between_image_flips:
        self._set_image_index((self.image_index + 1) % image_count)
        self.traveled -= travel_distance_between_image_flips
```

In dieser new_move-Funktion stehen all die Anweisungen, die in der ursprünglichen Version in den act-Methoden von Crab und Lobster implementiert waren. Die Reihenfolge der Anweisungen ist hier etwas anders, aber das hätten wir in den act-Methoden von Crab und Lobster theoretisch in der gleichen Reihenfolge implementieren können. Interessant ist der Aufruf self.old_move(distance), der dem Aufruf self.move(self.speed) in der alten act-Methode von Crab und Lobster entpricht. Und tatsächlich entspricht er diesem, denn wir hatten die move-Methode ja kurz zuvor in old_move umbenannt.

Die beiden ersten Schritte hat animate nun erfolgreich erledigt. Bleiben die Schritte 3 und 4:

```
gobj.traveled = 0
gobj._set_image_index(0)
```

Diese beiden Anweisungen hatten wir in der ursprünglichen Version in der __init__-Methode der Klassen Crab und Lobster realisiert.

Wir beenden hier die Diskussion der animate-Funktion. Diese ist nun ein mächtiges Werkzeug, um eine GameObj-Klasse, die nur ein statisches Spielfigurbild unterstützt, durch eine einzige Zeile in eine Klasse zu verwandeln, die eine animierte Spielfigur unterstützt. Wir setzen die neue animate-Funktion nun ebenfalls in Lobster ein:

```
class Lobster(GameObj):
    def __init__(self, pos, speed, drunkenness, jumpiness):
        animate(self, lambda i: "lobster" + str(i), 2, 7)
        self.pos = pos
        self.speed = speed
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness

def act(self):
    if self.can_move(self.speed):
        self.move(self.speed)
    else:
        self.turn(25)
    if random.randrange(10) < self.drunkenness:
        self.turn(random.uniform(-self.jumpiness, self.jumpiness))
    self.stage.take_out_neighbouring_crab(self)</pre>
```

11.7 Würmer animieren

Die Würmer werden wir (ohne den Einsatz der in Abschnitt 11.6.1 eingeführten animate-Funktion) animieren. Würmer sollen sich nur nach links und nach rechts bewegen dürfen. Zu Beginn beim Erzeugen und initialisieren eines Wurms entscheiden wir zufällig über die initiale Bewegungsrichtung des Wurms:

```
self.speed = 0.5
if random.randrange(2) == 0:
    self.speed = -self.speed
```

Ansonsten können wir die Initialisierung des Wurms wie in der Klasse Lobster gestalten:

```
class Worm(GameObj):
    IMAGE_PREFIX = "worm"
    IMAGE_COUNT = 2
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 3

def __init__(self, pos):
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0.5
        if random.randrange(2) == 0:
              self.speed = -self.speed
        self.traveled = 0

def __set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Worm.IMAGE_PREFIX + str(image_index)
```

Wir haben lediglich zwei Bilder zur Animation zur Verfügung, die wir alle 3 Pixel-Schritte "flippen" wollen. Die Bilder worm0.png und worm1.png kopieren wir in das images-Verzeichnis (vgl. Abbildung 36), die alte worm.png-Datei entfernen wir, da sie mit worm0.png übereinstimmt.

Abbildung 36: Zwei leicht abgewandelte Bilder des Wurms

Die act-Methode ähnelt der act-Methode von Lobster:

```
def act(self):
    self.switch_image()
    if self.can_move(self.speed):
        self.move(self.speed)
        self.traveled += abs(self.speed)
    else:
        self.speed = -self.speed
    if random.randrange(100) < 10:
        self.speed += random.uniform(-1, 1) / 50</pre>
```

Ein Unterschied zum Lobster besteht hier in der Reaktion auf die Erreichung des Bühnenrands. Statt eine Drehbewegung zu vollführen (turn), legen wir einfach den Rückwärtsgang ein:

```
self.speed = -self.speed
```

Außerdem besitzen Würmer keine drunkenness, sondern sie bewegen sich schnurgerade auf einer Linie. Allerdings wollen wir die Geschwindigkeit der Würmer variabel halten. Manchmal werden Würmer schneller, manchmal langsamer. Die folgende Anweisung modifiziert die Geschwindigkeit um einen zufälligen negativen oder positiven Wert:

```
self.speed += random.uniform(-1, 1) / 50
```

Zum Schluss kommt noch die von Lobster kopierte und leicht angepasste switch_image-Methode, in der wir an den Stellen des Konstantenzugriffs nur den Klassennamen Lobster durch worm ersetzen mussten:

```
def switch_image(self):
    if self.traveled > Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
        self._set_image_index((self.image_index + 1) % Worm.IMAGE_COUNT)
        self.traveled -= Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

Fertig. Die Würmer wandern in leicht unterschiedlicher Geschwindigkeit in horizontaler Richtung und werden dabei etwas animiert.

11.8 Das vollständige bisherige Programm

Erneut wollen wir das vollständige bisherige Programm abdrucken:

```
import random
import pgzrun
from pgzo import *
```

```
WIDTH = 560
               # screen width
               # screen height
HEIGHT = 460
TITLE = "Crab" # window title
# "Schaltzentrale" für die Schwierigkeit des Spiels:
START_WITH_LOBSTERS = 1
START_LOBSTER_STRENGTH = 0.2
START_WITH_WORMS = 20
class Beach(Stage):
   def __init__(self):
        self._defeated = False
        self._victorious = False
        self._score = 0
        if not hasattr(self, "_level"):
           self.\_level = 1
        self.background_image = "sand"
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(START_WITH_WORMS):
           w = Beach._create_random_worm()
           w.appear_on_stage(self)
        num_lobsters = START_WITH_LOBSTERS + self._level - 1
        for dummy in range(num_lobsters):
            lobster = Beach._create_random_lobster(START_LOBSTER_STRENGTH)
            lobster.appear_on_stage(self)
   def take_out_neighbouring_worms(self, crab):
        for w in self.get_game_objects(Worm):
            if w.overlaps(crab):
                w.leave_stage()
                sounds.blop.play()
                self._score += 1
        if not self._victorious and self.count_game_objects(Worm) == 0:
           self._victorious = True
            self.leave_all(Lobster)
           self.schedule_gameover()
   def take_out_neighbouring_crab(self, lobster):
        for c in self.get_game_objects(Crab):
           if c.overlaps(lobster):
                c.leave_stage()
                sounds.au.play()
                # There is only one crab on the beach, so we set
                # _defeated=True if we found at least one overlapping crab:
                self._defeated = True
                self.schedule_gameover()
   def restart(self):
        if self._victorious:
           self.leave_all()
            self._level += 1
           self.__init__()
        elif self._defeated:
           sys.exit()
        else:
            raise Exception("restart on unfinished game detected!")
   def schedule_gameover(self):
        clock.schedule_unique(self.restart, 4.0)
   @staticmethod
   def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                         1 + strength * 4,
                         1 + strength * 9,
                         1 + strength * 24 )
```

```
result.turn(random.randrange(360))
        return result
   @staticmethod
   def _create_random_worm():
        return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
   def draw(self):
        screen.draw.text("Score: " + str(self._score), topleft= (10,10),
                         color="black", fontsize=20, fontname="zachary")
        screen.draw.text("Level: " + str(self._level), topleft= (10,35),
                         color="black", fontsize=20, fontname="zachary")
       if self._defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
       if self. victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
class Crab(GameObj):
   IMAGE_PREFIX = "crab"
    IMAGE\_COUNT = 6
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
   def __init__(self, pos):
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0
        self.traveled = 0
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Crab.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
       else:
            self.speed = 0
       self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
           self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
   def switch_image(self):
       if self.traveled > Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Crab.IMAGE_COUNT)
            self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
class Lobster(GameObj):
   IMAGE_PREFIX = "lobster"
   IMAGE\_COUNT = 2
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 7
   def __init__(self, pos, speed, drunkenness, jumpiness):
        self._set_image_index(0)
        self.pos = pos
        self.speed = speed
        self.traveled = 0
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
   def _set_image_index(self, image_index):
```

```
self.image_index = image_index
        self.image = Lobster.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
        else:
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        self.stage.take_out_neighbouring_crab(self)
   def switch_image(self):
        if self.traveled > Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Lobster.IMAGE_COUNT)
            self.traveled -= Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
class Worm(GameObj):
   IMAGE_PREFIX = "worm"
   IMAGE\_COUNT = 2
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 3
   def __init__(self, pos):
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0.5
        if random.randrange(2) == 0:
            self.speed = -self.speed
        self.traveled = 0
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Worm.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.speed = -self.speed
        if random.randrange(100) < 10:</pre>
            self.speed += random.uniform(-1, 1) / 50
   def switch_image(self):
        if self.traveled > Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % worm.IMAGE_COUNT)
            self.traveled -= Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
beach = Beach()
beach.show()
pgzrun.go()
```

12 Die Krabbe wird unverwundbar

Wir wollen ein Power-Up in unser Spiel einbauen. Wenn der Spieler die Leertaste betätigt, soll die Krabbe für 3 Sekunden lang unverwundbar sein. In dieser Zeit soll ein "Schutzschild" angezeigt werden. Insgesamt hat die Krabbe pro Spiellevel Energiereserven für 10 Mal je 3 Sekunden Unverwundbarkeit. Danach muss der Spiellevel ohne Power-Up geschafft werden. Zu Beginn des nächsten Spiellevels gibt es wieder volle Energie.

12.1 Schutzschild-Grafiken erstellen

Wir beginnen damit, halbtransparente Grafiken für den Schutzschild und für die Energieanzeige zu erstellen. Hierzu ist das Werkzeug Inkscape⁴¹ eine gute Hilfe. Sollten Sie sich nicht die Mühe machen wollen, es selbst mit Inkscape einmal auszuprobieren, finden Sie das Ergebnis in den Dateien shield.png und energy_block.png.

In Inkscape erzeugt man zunächst eine Kreisscheibe. Dazu ist links ein Werkzeug verfügbar (s. Abbildung 37 links). Die Kreisscheibe kann man dann einfach in den Zeichenbereich zeichnen. Das sieht dann etwa so aus wie in Abbildung 37 mittig. Um die Kreisscheibe wirklich rund zu bekommen, korrigieren wir die Größenangaben am oberen Bildrand. Wir setzen z. B. Breite und Höhe auf den Wert 100 Pixel (s. Abbildung 37 rechts). Diese Kreisscheibe färben wir nun bläulich halbtransparent ein. Dazu wählen wir den Menüpunkt "Objekt" > "Füllung und Kontur". Im auf der rechten Seite erscheinenden Dialog selektieren wir den Reiter "Füllen" (s. Abbildung 38 links). Hier können wir z. B. die Schieberegler benutzen oder auch einfach direkt die Farbe 48d1e193 in das Eingabefeld "RGBA:" eintippen.

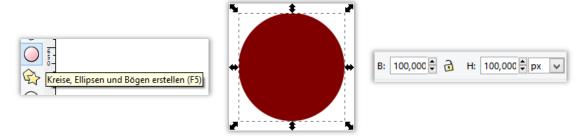


Abbildung 37: Kreisscheibenwerkzeug (links). Gezeichnete Kreisscheibe (mittig). Größenangaben (rechts)

⁴¹ https://inkscape.org

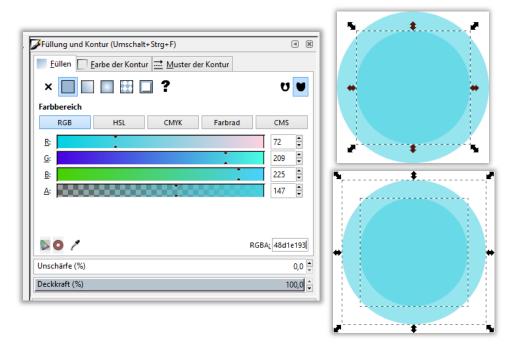


Abbildung 38: Halbtransparent einfärben (links). Verkleinertes Kreisscheiben-Duplikat (oben rechts). Beide Kreisscheiben selektiert (unten rechts)

Aus der Kreisscheibe wollen wir nun einen Ring machen. Dazu selektieren wir die Kreisscheibe im Zeichenbereich (einmal anklicken) und duplizieren Sie mit der Tastenkombination Strg-D. Nun liegen zwei Kreisscheiben übereinander. Und nun halten wir die beiden Tasten Strg-Shift gedrückt und verkleinern die oben liegende Kreisscheibe, in dem wir an einem der vier Eck-Doppelpfeile mit der Maus ziehen. Das sieht jetzt etwa so aus wie in Abbildung 38 oben rechts.

Schließlich selektieren wir beide Kreisscheiben gleichzeitig (z. B. bei gehaltener Shift-Taste die andere Kreisscheibe ebenfalls mit der linken Maustaste anklicken, s. Abbildung 38 unten rechts). Der Menüpunkt "Pfad" > "Differenz" erzeugt nun den gewünschten Ring (s. Abbildung 39 links oben). Diesen müssen wir nun noch im PNG-Format exportieren. Der Menüpunkt "Datei" > "PNG-Bild exportieren" öffnet rechts einen Dialog, in dem wir den Dateipfad "…/images/shield.png" eingeben und die Breite und Höhe auf 150 x 150 Pixel setzen (vgl. Abbildung 39 rechts). Mit Betätigung des Buttons wird die PNG-Datei erstellt.

Nun wollen wir noch einen Block zur Anzeige der verbleibenden Energie erzeugen. An anderer Stelle auf dem Zeichenbereich erstellen wir ein gefülltes Rechteck, das wir in der gleichen Farbe wie die Kreisscheiben einfärben. Die Größe passen wir auf 8 Pixel Breite und 20 Pixel Höhe an (vgl. Abbildung 39 links unten). So lange das Rechteck noch selektiert ist, wählen wir wieder den Menüpunkt "Datei" > "PNG-Bild exportieren". Dort geben wir den Dateipfad ".../images/energy_block.png" ein und achten auf die Bildgröße 8 Pixel breit und 20 Pixel hoch. Auch hier betätigen wir abschließend — Exportieren. Hiermit haben wir die beiden benötigten Dateien erfolgreich erstellt.

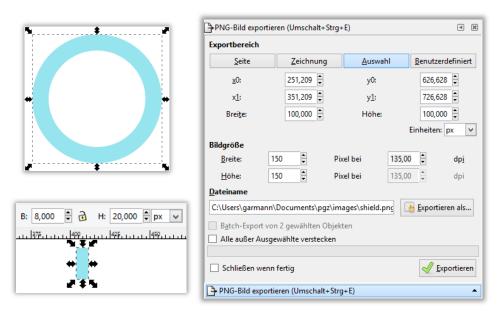


Abbildung 39: Ring (links oben). PNG-Export (rechts). Block (links unten)

12.2 Schutzschild anzeigen

Zur Anzeige des Schutzschildes bedienen wir uns der Actor-Klasse von Pygame Zero, da diese am einfachsten zu handhaben ist. In unserer Crab-Klasse schreiben wir eine neue draw-Methode⁴²:

```
def draw(self):
    shield = Actor("shield", self.pos)
    shield.draw()
```

Wenn wir das Programm ausführen, können wir den Schutzschild sehen (s. Abbildung 40). Der Schutzschild wandert immer mit der Krabbe mit. So soll es sein.



Abbildung 40: Schutzschildanzeige

⁴² Wie Sie sehen können, programmieren wir in die draw-Methode von Crab nur die zusätzlichen Anweisungen, die über das Bild der Krabbe hinaus gezeichnet werden müssen. Leser mit Hintergrundwissen über Vererbung in objektorientierten Programmiersprachen werden vielleicht einwenden, dass in unserer draw-Methode ein Aufruf der Superklassen-draw-Methode fehlt. Wenn Sie nicht wissen, was eine Superklassen-Methode ist, können Sie die Lektüre dieser Fußnote beenden. Ansonsten kommt hier eine kurze Erläuterung. Die pgzo-Bibliothek will die Implementierung von Subklassen so einfach wie möglich machen. Aufrufe von Superklassen-Methoden und insbesondere der Superklassen-__init___-Methode sind syntaktisch schwierig und fehleranfällig. Zudem ist die Bedeutung von super() schwer durchschaubar, wenn Subklassen mit Mehrfachvererbung eingesetzt werden. Um dieser Problematik von vornherein aus dem Weg zu gehen, ruft die Klasse Stage für jede auf der Bühne anwesende Spielfigur sowohl die Superklassen draw-Methode auf und zusätzlich (falls vorhanden) auch noch die Subklassen-draw-Methode. Würden wir selbst den Aufruf super().draw() in unserer draw-Methode ergänzen, würde es noch funktionieren, nur würde die Superklassen-draw-Methode eben zwei Mal ausgeführt.

Der Schutzschild soll selbstverständlich nur dann angezeigt werden, wenn der Spieler die Leertaste betätigt hat. Wir ergänzen ein Objektattribut shielded:

```
class Crab(GameObj):
    IMAGE_PREFIX = "crab"
    IMAGE_COUNT = 6
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5

def __init__(self, pos):
    self._set_image_index(0)
    self.pos = pos
    self.speed = 0
    self.traveled = 0
    self.shielded = False
```

Das neue Attribut wird in der act-Methode in dem Moment auf True gesetzt, wenn der Spieler die Leertaste betätigt:

```
def act(self):
    ... Anfang unverändert ...
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
    if keyboard.space and not self.shielded:
        self.shielded = True
        clock.schedule_unique(self.unshield, 3)
```

Wie wir sehen, haben wir in die Bedingung ein and not self.shielded ergänzt. Der Grund liegt darin, dass wir in der letzten Zeile mit clock das Ende des Schutzschilds in genau 3 Sekunden einplanen wollen. Wie bereits in Abschnitt 9.8 ausgeführt, müssen wir verhindern, dass uns das Esel-Möhren-Prinzip ereilt. Die Anweisung clock.schedule_unique(self.unshield, 3) darf nur einmal ausgeführt werden. Dies erreichen wir durch genau diese Zusatzbedingung and not self.shielded.

Die Methode unshield, die in der clock-Anweisung verwendet wird, gibt es noch gar nicht. Diese schreiben wir jetzt:

```
def unshield(self):
    self.shielded = False
```

Das war einfach. Wenn 3 Sekunden abgelaufen sind, ruft die Laufzeitumgebung automatisch unshield auf. Dort wird der Schutzschild deaktiviert.

Nun müssen wir in draw noch abfragen, ob der Schutzschild derzeit aktiv ist, und nur dann den Schutzschild zeichnen:

```
def draw(self):
    if self.shielded:
        shield = Actor("shield", self.pos)
        shield.draw()
```

Wenn wir das nun ausprobieren, erscheint der Schutzschild genau in dem Moment, indem der Spieler der Leertaste betätigt. 3 Sekunden später verschwindet der Schutzschild wieder.

12.3 Ein schützender Schutzschild

Noch hat der Schutzschild keine schützende Wirkung. Dies wollen wir nun ergänzen. In der Beach-Methode take_out_neighbouring_crab erweitern wir die Bedingung der if-Anweisung wie folgt:

12.4 Energie 109

Mit dieser erweiterten Bedingung wird die Krabbe nicht angetastet, so lange der Wert von c.shielded True ist. Zum Beweis verweisen wir auf die in Abbildung 41 dargestellte Spielsituation.



Abbildung 41: Ein wirkungsvoller Schutzschild

Dass Hummer und Krabbe sich überlappen, ist noch nicht perfekt, Man könnte sich wünschen, dass der Hummer vom Schutzschild abprallt. Oder der Hummer könnte zerstört werden. Oder, oder oder. Ihren eigenen Ideen sind an dieser Stelle keine Grenzen gesetzt. Setzen Sie diese bei Interesse direkt in die Tat um!

12.4 Energie

Der Spieler soll maximal 10 Mal pro Level einen Schutzschild erhalten. Wir installieren in der Crab-Klasse einen "Energie"-Wert, der zu Beginn auf 10 initialisiert wird:

```
class Crab(GameObj):
    IMAGE_PREFIX = "crab"
    IMAGE_COUNT = 6
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5

def __init__(self, pos):
    self._set_image_index(0)
    self.pos = pos
    self.speed = 0
    self.traveled = 0
    self.shielded = False
    self.shield_energy = 10
```

Jede Betätigung eines neuen Schutzschildes soll diesen Wert um 1 verringern:

110 12.4 Energie

```
def act(self):
    ... Anfang unverändert ...
    if keyboard.left:
        self.turn(10)
    if keyboard.right:
        self.turn(-10)
    if keyboard.up:
        self.speed_up()
    if keyboard.down:
        self.slow_down()
    if keyboard.space and not self.shielded and self.shield_energy > 0:
        self.shielded = True
        self.shield_energy -= 1
        clock.schedule_unique(self.unshield, 3)
```

In der if-Bedingung haben wir am Ende ein and self.shield_energy > 0 ergänzt. Dadurch erreichen wir, dass die Leertaste wirkungslos bleibt, wenn der Energiespeicher erschöpft ist.

Wir können die neue Funktion schon testen. Nach dem 10-ten Schutzschild gibt es keinen 11-ten mehr.

Schöner wird es nun, wenn wir den aktuellen Energie-Wert anzeigen. Wir verwenden dazu die Bilddatei energy_block.png. Wir werden in der oberen rechten Ecke des Fensters genau die Anzahl an Blöcken zeichnen, die im Objektattribut shield_energy gespeichert ist. Der Block wird wieder als Actor realisiert:

```
def draw(self):
    if self.shielded:
        shield = Actor("shield", self.pos)
        shield.draw()

        energy_block = Actor("energy_block")
```

Nun werden wir eine for-Schleife programmieren, die genauso oft durch läuft wie durch das Objektattribut shield_energy vorgegeben ist. In der for-Schleife werden wir eine x-Koordinate immer wieder um einen festen Pixelwert verringern. Wir beginnen mit x am rechten Fensterrand:

```
x = WIDTH - 20
y = 20
for dummy in range(self.shield_energy):
    energy_block.topright = (x, y)
    energy_block.draw()
    x -= 10
```

Der Actor wird mit seiner rechten oberen Ecke positioniert und gezeichnet. Dann verringern wir x um den Wert 20. Das ganze wiederholen wir immer wieder, d. h. wir zeichnen denselben Actor mehrfach auf das Fenster.

Das Ergebnis sieht aus wie in Abbildung 42. In der dargestellten Spielsituation hatte ich bereits zwei Mal den Schutzschild aktiviert. Es verbleiben noch acht Blöcke.

12.4 Energie 111



Abbildung 42: Energieanzeige

Der Schutzschild ist damit fertig. Im nächsten Kapitel werden wir eine weitere Bühne für den Startbildschirm erstellen Dort werden wir demonstrieren, wie Mauseingaben verarbeitet werden können.

13 Startbühne

13.1 Eine Klasse für die Startbühne

Wir beginnen mit einer sehr einfachen Version der Startbühne:

Auf der Startbühne gibt es keine Spielfiguren. Wir implementieren eine einfache draw-Methode, die den Titel des Spiels ausgibt. Das Hintergrundbild start.png hatten wir bereits in Abschnitt 4.3 an geeigneter Stelle abgelegt.

Die neue Startbühne soll die erste Bühne sein, die der Spiele zu sehen bekommt. Ganz am Ende unseres Programms ersetzen wir daher die dort stehenden Anweisungen wie folgt:

```
beach = Beach()
beach.show()
s = Start()
s.show()

pgzrun.go()
```

Wenn wir unser Programm starten, erscheint die neue Startbühne (s. Abbildung 43).



Abbildung 43: Startbühne

13.2 Das Spiel mit der Leertaste starten

Die Startbühne soll in ihrer update-Methode zunächst ganz einfach auf die Leertaste reagieren. Wenn die Leertaste betätigt wird, soll das Spiel beginnen. Wir implementieren die Tastaturereignisverarbeitung wie gewohnt:

```
def update(self):
    if keyboard.space:
        b = Beach()
        b.show()
```

Falls die if-Bedingung zutrifft, wird eine Objekt der Klasse Beach erzeugt: b = Beach(). Danach rufen wir b.show() auf, um die neu erzeugte Strandbühne anzuzeigen.

Das funktioniert so weit. Allerdings beginnt das Spiel sofort mit einem aktivierten Schutzschild. Offenbar ist die betätigte Leertaste nicht nur von der Startbühne sondern auch noch von der Strandbühne interpretiert worden.

Das Problem können wir auf verschiedene Arten lösen. Erstens: wir benutzen auf der Startseite eine andere Taste, z. B. die Return-Taste. Da return ein Schlüsselwort in Python ist, erfolgt der Zugriff in diesem Fall etwas anders:

```
def update(self):
   if keyboard['return']:
    ...
```

Eine zweite Lösung verzögert den Start einfach um einige Millisekunden:

Diese Variante ist nicht so schön, da i. a. davon abgeraten wird, die Spielmechanik künstlich zu verzögern. Aber vielleicht sind zwei Zehntelsekunden nicht so schlimm.

Eine dritte Lösung greift auf eine private Methode des von Pygame Zero definierten keyboard-Objekts zu. Bevor die Strandbühne angezeigt wird, melden wir dem keyboard-Objekt, dass die Leertaste wieder losgelassen wurde. Wenn dies auch nicht stimmt, so gaukeln wir es dem Pygame Zero keyboard zumindest erfolgreich vor.

```
def update(self):
    if keyboard.space:
        keyboard._release(pygame.K_SPACE)
    b = Beach()
    b.show()
```

Eine vierte Lösung setzt die von Pygame Zero unterstützten Ereignis-Handler ein. Zusammen mit der pgzo-Bibliothek verwendet man diese wie folgt:

In dieser vierten Lösung "horchen" wir sozusagen auf das Loslassen der Leertaste. D. h. wir können in diesem Moment sicher sein, dass die Leertaste nicht mehr gedrückt ist. In diesem Moment wechseln wir in die Strandbühne. Dort ist nun zu Beginn sicher kein Schutzschild aktiviert.

In der Darstellung des Abschnitts Abbildung 13 auf Seite 20 sind lediglich Aufrufe von update und draw vorgesehen. Tatsächlich ruft Pygame Zero bei Benutzereingaben weitere Funktionen in unserem Programm auf. Wenn bspw. eine Taste gedrückt wird, wird on_key_down ausgeführt. Die weiteren von Pygame Zero unterstützten Funktionen sind in der Online-Dokumentation⁴³ beschrieben.

Beim Einsatz der pgzo programmieren wir keine globalen Ereignis-Handler, sondern wir schreiben je Bühne einen eigenen Satz von Ereignis-Handlern als Methoden in der entsprechenden Bühnenklasse. Unterschied: jede dieser Methoden bekommt als ersten Parameter das Bühnenobjekt self übergeben. Die weiteren Parameter entsprechen den von Pygame Zero normalerweise an die globalen Ereignis-Handler übergebenen Parameter.

13.2.1 Kurzer Einschub: Mausereignis-Handler

Als ein Beispiel für einen Mausereignis-Handler programmieren wir in der Krabben-Klasse einen Notnagel. Wenn man mit der Maus ins Fenster klickt, springt die Krabbe direkt dorthin:

```
class Crab(GameObj):
    ... alles wie gehabt ...

    def on_mouse_down(self, pos, button):
        if button == mouse.LEFT:
            self.pos = pos
            self.speed = 0
```

Wie Sie sehen, haben wir auch die aktuelle Geschwindigkeit auf o gesetzt. Es ist also möglich, die Spielmechanik einer Spielfigur auf mehrere Methoden aufzuteilen. In der update-Methode und in allen Ereignis-Handlern zusammen findet die gesamte Spielmechanik statt. Manchmal ist es verwirrend, so viele Methoden zu schreiben. Wir empfehlen, so lange es keine Probleme wie etwa das obige mit der Leertaste gibt, dass die Spielmechanik in update implementiert wird. Ereignis-Handler sind zwar in komplexen Programmen, insb. in grafischen Benutzeroberflächen, an der Tagesordnung. In einem Computerspiel können Ereignis-Handler jedoch die Programmierung und vor allem die Fehlersuche unnötig erschweren.

Da das Spiel mit der neuen on_mouse_down-Methode langweilig wird, entfernen wir die Methode wieder.

13.3 Das Spiel mit der Maus starten

Schlussendlich wollen wir auf der Startbühne auch noch die Möglichkeit haben, das Spiel mit der Maus zu starten. Wir bieten dem Spieler dazu eine Schaltfläche an (vgl. Abbildung 44). Dazu programmieren wir:

⁴³ http://pygame-zero.readthedocs.io/en/stable/hooks.html

```
class Start(Stage):
   def __init__(self):
        self.background_image = "start"
        self.start_button_rect= Rect(WIDTH * 0.35, HEIGHT * 0.8, WIDTH * 0.3, HEIGHT * 0.1)
   def draw(self):
        screen.draw.text("The Crab Game", center = (WIDTH//2, HEIGHT * 0.25),
                         color="brown", fontsize=60, fontname="zachary")
        screen.draw.textbox("Start", self.start_button_rect,
                            color="black", fontname="zachary")
   def update(self):
        if mouse_state.left and self.start_button_rect.collidepoint(mouse_state.pos):
            self.start_game()
   def on_key_up(self, key):
        if key == keys.SPACE:
            self.start_game()
   def start_game(self):
        b = Beach()
        b.show()
```

Das Rechteck start_button_rect dient gleichzeitig als Textbox-Begrenzung und als Bereich, in dem ein Mausklick als gültig erkannt wird (collidepoint).



Abbildung 44: Startbildschirm mit Schaltfläche

Problematisch an dieser Lösung ist, dass man möglicherweise die Schaltfläche nicht als solche erkennt. Wir wollen den Text dynamisch gestalten. Wenn die Maus über die Start-Schaltfläche bewegt wird, soll der Text etwas "angehoben" werden. Dazu führen wir ein Attribut des Startbühnen-Objekts ein, das anzeigt, ob sich die Maus derzeit über der Start-Schaltfläche befindet.

```
class Start(Stage):
    def __init__(self):
        self.background_image = "start"
        self.start_button_rect= Rect(WIDTH * 0.35, HEIGHT * 0.8, WIDTH * 0.3, HEIGHT * 0.1)
        self.hover = False
```

Zu Beginn ist der Wert False. Wir werden den Wert dieses Attributs gleich in der update-Methode in dem Moment auf True setzen, in dem die Maus über die Startschaltfläche geschoben wird. Vorher programmieren wir jedoch in draw das "Anheben" des Start-Textes:

Durch einen Schatten unterhalb der Schrift ist die Start-Schaltfläche gut zu erkennen (s. Abbildung 45). Einen Schatten ergänzt man zum einen durch Angabe einer Schattengröße als (x,y)-Tupel. Außerdem gibt man die Schattenfarbe an. Es gibt verschiedene Möglichkeiten, die Farbe festzulegen. Wir haben hier die hexadezimale Schreibweise⁴⁴ gewählt.



Abbildung 45: Angehobene Start-Schaltfläche

Im Fall, dass die Maus nicht über der Schaltfläche verweilt, wird der else-Zweig der in draw implentierten if-Anweisung ausgeführt. Die beiden Zuweisungen bergen ein neues Python-Schlüsselwort None. None bezeichnet ein nicht vorhandenes Objekt. Die beiden Zuweisungen drücken also aus, dass keine Schattenposition und auch keine Schattenfarbe vorhanden sind. None ist etwas anderes als 0. Z. B. würde 0 die Farbe Schwarz bezeichnen. None bezeichnet eine nicht vorhandene Farbe.

None ist das Python-Wort für ein in vielen Programmiersprachen und Anwendungen vorhandene Konzept, das das "Nichts" beschreibt. Dieses braucht man häufig. Eine Excelttabelle mit leeren Zellen besitzt viele "Nichts"-Einträge. Eine Temperatur-Messreihe hat ein paar "Nichts"-Lücken, weil die technischen Geräte an einigen Tagen versagt haben. Grafisch kann man das Ergebnis der Anweisung shadow = None wir in en None-Wert wie in Abbildung 46 darstellen.



Abbildung 46: Darstellung des Ergebnisses der Anweisung shadow = None

Wir wenden uns nun der update-Methode zu:

```
def update(self):
    if mouse_state.moved:
        self.hover = self.start_button_rect.collidepoint(mouse_state.pos)
    if mouse_state.left and self.hover:
        self.start_game()

def on_key_up(self, key):
    if key == keys.SPACE:
        self.start_game()

def start_game(self):
    b = Beach()
    b.show()
```

In der pgzo ist ein Objekt mouse_state eingebaut, welches verschiedene Attribute zur Abfrage anbietet. Hier prüfen wir mit mouse_state.moved, ob sich die Maus seit der letzten Abfrage bewegt hat. In diesem Fall prüfen wir die Position der Maus relativ zum Rechteck rund um die Start-Schaltfläche. Wenn collidepoint True liefert, wird das hover-Attribut True. Dieses steuert dann auch gleich die darauf

⁴⁴ Eine Anleitung zu diesem Format einer Farbdefinition finden Sie z. B. im folgenden Wikipedia-Artikel: https://de.wikipedia.org/wiki/Hexadezimale_Farbdefinition

folgende if-Anweisung, die zum Start der Spiels führt, wenn gleichzeitig die linke Maustaste gedrückt wird.

13.4 Nach Spielende wieder zurück zur Startbühne

Wenn das Spiel verloren ist, soll wieder die Startbühne gezeigt werden. Wir müssen dazu in der entsprechenden Routine restart in der Beach-Klasse wieder die Startbühne anzeigen. An dieser Stelle kann man tatsächlich einmal über globale Variablen nachdenken. Wir haben es insgesamt mit genau zwei Bühnen zu tun. Diese beiden Bühnen wollen wir gleich zu Beginn des Spiels als Objekte so anlegen, dass man von allen Stellen des Programms darauf zugreifen kann.

Wir programmieren daher ganz am Ende unseres Programms:

```
start_stage = Start()
beach_stage = Beach()
start_stage.show()

pgzrun.go()
```

Nun müssen wir die Stelle des Spielstarts so abwandeln, dass wir die bereits existierende Strandbühne auf ihren Anfangszustand setzen und dann anzeigen:

```
class Start(Stage):
    ... Rest unverändert ...

def start_game(self):
    beach_stage.reset_game()
    beach_stage.show()
```

Die neue Methode reset_game existiert noch nicht und soll die alte __init__-Methode der Strandbühne ablösen. Wir brauchen jetzt nämlich zwei Initialisierungsarten für die Strandbühne: eine vollständige Initialisierung des Spiels inkl. Rücksetzung des Spiellevels auf 1 und eine Initialisierung des Zustandes der Strandbühne auf einen Zustand, dass ein neuer Spiellevel starten kann.

Die Strandbühne wollen wir wie folgt umorganisieren, indem wir die <u>__init__</u>-Methode fast komplett leer räumen und die ursprünglich dort enthaltenen Anweisungen auf zwei neue Methoden verteilen:

```
class Beach(Stage):
   def __init__(self):
        self.background_image = "sand"
   def reset_game(self):
        self._level = 1
        self._score = 0
        self.prepare_beach()
   def prepare_beach(self):
        self._defeated = False
        self._victorious = False
        self.leave_all()
        crab = Crab((WIDTH / 2, HEIGHT * 0.7))
        crab.appear_on_stage(self)
        for dummy in range(START_WITH_WORMS):
           w = Beach._create_random_worm()
           w.appear_on_stage(self)
        num_lobsters = START_WITH_LOBSTERS + self._level - 1
        for dummy in range(num_lobsters):
            lobster = Beach._create_random_lobster(START_LOBSTER_STRENGTH)
            lobster.appear_on_stage(self)
```

Es gibt nun zwei neue Initialisierungsmethoden: reset_game und prepare_beach:

- prepare_beach nimmt den Löwenanteil der bisherigen __init__-Methode auf. In prepare_beach wird der Strand mit Spielfiguren bevölkert. Um sicher zu gehen, dass der Strand am Anfang leer ist, wird ganz zu Beginn der Methode self.leave_all() ausgeführt. Diese Anweisung haben wir sozusagen aus restart nach prepare_beach verschoben. Zwar wird self.leave_all() nun auch unnötigerweise ganz am Anfang beim allerersten Start der Strandbühne ausgeführt, aber das schadet ja nicht wirklich.
- Die reset_game-Methode ist als Ersatz für die alte __init__-Methode gedacht. Da es unüblich ist, private Methoden wie die __init__-Methode von außerhalb der Klasse aufzurufen, haben wir uns entschieden, mit reset_game eine öffentliche Schnittstelle anzubieten, um der Startbühne zu ermöglichen, das Spiel zurückzusetzen. Anders als bisher haben wir uns entschieden, den Score nicht nach jedem gewonnenen Level auf 0 zu setzen. Man kann nun also über mehrere Levels hinweg Punkte sammeln. Deshalb wird der Score nur einmal in reset_game auf 0 initialisiert und nicht in prepare_beach. Mit dem Zurücksetzen des Spiels ist auch eine Vorbereitung der Spielfiguren auf dem Strand verbunden. Deshalb rufen wir am Ende von reset_game die prepare_beach-Methode auf.

Schließlich haben wir die Initialisierung des Hintergrundbildes in <u>__init__</u> belassen, weil sich dieses ja sowieso nie ändert. Das reicht also, dass wir das einmal beim ersten Spielstart initialisieren.

Bleibt noch eine Abwandlung der restart-Methode in Beach zu leisten:

13.5 Ein Quit-Button

```
def restart(self):
    if self._victorious:
        self.leave_all()
        self._level += 1
        self._init_()
        self.prepare_beach()
    elif self._defeated:
        sys.exit()
        start_stage.show()
    else:
        raise Exception("restart on unfinished game detected!")

... restliche Methoden unverändert ...
```

Nach einem verlorenen Spiel wird wieder die Startbühne angezeigt (start_stage.show()). Dies erfolgt durch einen Zugriff auf die globale Variable start_stage. Nach einem gewonnen Level, erhöhen wir den Levelzähler um 1 und bereiten den Strand für den nächsten Level vor (self.prepare_beach()).

13.5 Ein Quit-Button

Nach jedem verlorenen Spiel gelangen wir nun zurück zur Startbühne. Wir wollen dem Spieler die Möglichkeit geben, das Spiel mit einer "Quit"-Schaltfläche zu verlassen. Wir nutzen die Gelegenheit, um die bisher direkt in der Startbühne implementierte Logik für die Start-Schaltfläche in ein separates Spielfigur-Objekt zu verschieben. Jede Schaltfläche (**Button**) soll eine Instanz der folgenden neuen Klasse Button sein. Da Buttons nicht verschoben werden sollen und auch nichts mit Himmelskörpern zu tun haben, geben wir als Basistyp der Klasse direkt den Typ GameObj in der ersten Zeile an:

```
class Button(GameObj):
    def __init__(self, image_normal, image_hover, pos, action):
        self.pos = pos
        self.image = image_normal
        self.image_normal = image_normal
        self.image_hover = image_hover
        self.action = action
        self.hover = False
```

Ein Button besitzt zwei Bilder: eines für die Anzeige im Normalzustand und eines, das angezeigt wird, wenn sich die Maus über dem Button befindet. Die beiden Bilder speichern wir in der __init__-Methode als Objekt-Attribute genauso wie den schon bekannten Zustand hover. Als letzter Parameter der __init__-Methode wird ein Funktionsobjekt action erwartet. Auch dieses speichern wir als Objekt-Attribut.

Wir implementieren in Button keine update-Methode, sondern stattdessen zwei Ereignis-Handler:

```
def on_mouse_move(self, pos):
    self.hover = self.collidepoint(mouse_state.pos)
    if self.hover:
        self.image = self.image_hover
    else:
        self.image = self.image_normal

def on_mouse_down(self, pos, button):
    if button == mouse.LEFT and self.hover:
        self.action()
```

Die beiden verwendeten Ereignis-Handler sind auf der Pygame Zero Projekt-Seite⁴⁵ dokumentiert. Zur Bestimmung der Mausposition verwenden wir die Methode collidepoint aus der Klasse Actor, die auch von GameObj angeboten wird. Je nach Ausgang der Positionsbestimmung wechseln wir das Bild gegen das jeweils andere Bild aus.

Die Startbühne können wir nun deutlich verschlanken, da wir einige der alten Inhalte in Button implementiert haben. Eine update-Methode benötigen wir nicht mehr – darum kümmert sich jetzt die Stage-Klasse. Da wir die beiden Buttons als Spielfiguren auf der Bühne erscheinen lassen

⁴⁵ https://pygame-zero.readthedocs.io/en/stable/hooks.html#event-handling-hooks

13.5 Ein Quit-Button

(appear_on_stage), kennt die Stage-Klasse diese nun und versorgt diese mit Aufrufen der dort implementierten Ereignis-Handler-Methoden. Wir fügen gleichzeitig den zweiten gewünschten Quit-Button hinzu. Die benötigten Bilder start_button_normal.png, start_button_hover.png, quit_button_normal.png und quit_button_hover.png legen wir noch im images-Unterverzeichnis ab.

```
class Start(Stage):
    def __init__(self):
        self.background_image = "start"
        self.reset()
    def reset(self):
        self.leave all()
        self.start_button = Button("start_button_normal", "start_button_hover",
                                   (WIDTH * 0.5, HEIGHT * 0.5), self.start_game)
        self.quit_button = Button("quit_button_normal", "quit_button_hover",
                                   (WIDTH * 0.1, HEIGHT * 0.9), sys.exit)
        self.start_button.appear_on_stage(self)
        self.quit_button.appear_on_stage(self)
    def draw(self):
        screen.draw.text("The Crab Game", center = (WIDTH//2, HEIGHT * 0.25),
                         color="brown", fontsize=60, fontname="zachary")
    def on_key_up(self, key):
        if key == keys.SPACE:
            self.start_game()
    def start_game(self):
        beach_stage.reset_game()
        beach_stage.show()
```

Die Initialisierung der Startbühne haben wir ähnlich wie in der Strandbühne umgesetzt. Die neue reset-Methode müssen wir nun noch von der Strandbühne aus aufrufen, um beim Neuanzeigen der Startbühne sicher zu stellen, dass diese wieder ihren Initialzustand hat. Ansonsten könnte vielleicht noch der Start-Button im "hover"-Zustand angezeigt werden, obwohl die Maus gar nicht mehr über ihm steht. In der restart-Methode der Beach-Klasse ergänzen wir den folgenden Aufruf:

```
def restart(self):
    if self._victorious:
        self._level += 1
        self.prepare_beach()
    elif self._defeated:
        <u>start_stage.reset();</u>
        start_stage.show()
    else:
        raise Exception("restart on unfinished game detected!")
```

Zwei Szenenbilder zeigt Abbildung 47.

13.5 Ein Quit-Button



Abbildung 47: Neuer Startbildschirm, der nun auf Spielfigur-Buttons basiert. Der Mauszeiger wird durch einen Pfeil symbolisiert.

12.2 14.1 Game Assets

14 Musik

14.1 Game Assets

Zunächst kopieren wir einige benötigte Dateien:

• Vier Bilddateien⁴⁶ sound_on_normal.png, sound_on_hover.png, sound_off_normal.png und sound_off_hover.png in den Ordner images,

• Eine MP3-Datei⁴⁷ sthlm_sunset.mp3 in einen neuen Ordner music.

Wir sorgen nun für Hintergrundmusik, und zwar ganz am Ende, kurz bevor wir die Bühne freigeben:

```
music.play("sthlm_sunset")
start_stage = Start()
beach_stage = Beach()
start_stage.show()

pgzrun.go()
```

Wie das music-Objekt funktioniert, steht z. B. in der Online-Dokumentation⁴8 von Pygame Zero. Dort ist auch eine große Warnung dokumentiert, dass dieses Feature noch experimentell ist. Sollte es auf Ihrem Computer durch Einsatz der Hintergrundmusik zu Schwierigkeiten kommen, müssen Sie darauf wohl zunächst verzichten und auf die nächste Pygame Zero Version warten ⊕. Unabhängig davon werden wir in den folgenden Abschnitten die Möglichkeit schaffen, Sounds abzuschalten. Dies soll auch Geräusche betreffen, die beim Fressen von Würmern und bei der Kollision mit einem Hummer entstehen. D. h., auch wenn Musik auf Ihrem Rechner nicht funktioniert, lesen Sie bitte weiter.

14.2 Sound-Button

Um die Musik abzuschalten, erstellen wir einen Button. Wir erstellen diesen als globale Variable, da wir den Sound-Button in beiden Bühnen benötigen:

```
sound_button = Button("sound_on_normal.png", "sound_on_hover.png", (WIDTH-30, HEIGHT-30), toggle_sound)
sound_button.currently_playing = True
```

Wir ergänzen in diesem Button ein Objekt-Attribut currently_playing, in dem wir speichern, ob die Musik gerade spielt oder nicht. Die als letzter Parameter angegebene Funktion toggle_sound, die bei Betätigung des Buttons ausgeführt werden soll, müssen wir noch programmieren:

Die Funktion toggle_sound prüft, in welchem Zustand der Wert currently_playing gerade ist. Wenn die Musik gerade spielt, tauschen wir das aktuelle Bildpaar gegen das andere Bildpaar "sound_off..." aus und versetzen die Musik mit music.pause() in den Pause-Zustand. Falls die Musik gerade aus, machen wir es anders herum. Schließlich invertieren wir den Wahrheitswert sound_button.currently_playing, indem wir ihn mit seinem Gegenteil belegen (not). Am Ende sorgen wir noch durch

⁴⁶ Bildquelle: Eigene Modifikation basierend auf Icons von Victor Erixon via https://www.iconfinder.com/icons/106215/off_sound_icon#size=64.

⁴⁷ Dateiquelle: Ehrling via https://soundcloud.com/ehrling

⁴⁸ https://pygame-zero.readthedocs.io/en/stable/builtins.html#music

sound_button.update_image() dafür, dass der Button das richtige Bild aus dem neuen Bildpaar ("hover" oder "normal") tatsächlich anzeigt. Die neue Methode update_image in Button schreiben wir wie folgt:

```
class Button(GameObj):
    def __init__(self, image_normal, image_hover, pos, action):
        self.pos = pos
        self.image = image_normal
        self.image_normal = image_normal
        self.image_hover = image_hover
        self.action = action
        self.hover = False
    def update_image(self):
        if self.hover:
            self.image = self.image_hover
        else:
            self.image = self.image_normal
    def on_mouse_move(self, pos):
        self.hover = self.collidepoint(mouse_state.pos)
        self.update_image()
    def on_mouse_down(self, pos, button):
        if button == mouse.LEFT and self.hover:
            self.action()
```

D. h., wir haben einfach einen Teil des Codes aus on_mouse_move in die neue update_image-Methode verschoben.

14.3 Sound-Button in beiden Bühnen verwenden

Die globale Variable sound_button referenzieren wir nun aus den Initialisierungsroutinen beider Bühnen:

```
class Start(Stage):
    ...
    def reset(self):
     ... Anfang unverändert ...
     sound_button.appear_on_stage(self)
```

Und:

```
class Beach(Stage):
    ...
    def prepare_beach(self):
        ... Anfang unverändert ...
        sound_button.appear_on_stage(self)
```

Nun ist der Sound-Button auf beiden Bühnen rechts unten zu sehen. Wenn man mit der Maus darüber fährt, wird der Button etwas heller dargestellt (s. Abbildung 48). Wenn man ihn betätigt, wechselt er das Bild (s. Abbildung 49 links) und die Musik stoppt. Dieser Zustand des Buttons ändert sich auch nicht, wenn man nun das Spiel startet (s. Abbildung 49 rechts). Man kann den Sound in beiden Bühnen beliebig ein- und wieder ausschalten.



Abbildung 48: Sound-Button mit und ohne Maus-"Hovering"



Abbildung 49: Ausgeschalteter Sound

Bleibt uns noch, dafür zu sorgen, dass die übrigen Sounds (Fressen der Würmer, Verlust der Krabbe) ebenfalls unterdrückt werden, wenn der Spieler um Ruhe gebeten hat. In der Beach-Klasse müssen wir je eine if-Anweisung einfügen:

```
def take_out_neighbouring_worms(self, crab):
    for w in self.get_game_objects(Worm):
        if w.overlaps(crab):
            w.leave_stage()
            if sound_button.currently_playing:
                sounds.blop.play()
            self._score += 1
   if not self._victorious and self.count_game_objects(Worm) == 0:
        self._victorious = True
        self.leave_all(Lobster)
        self.schedule_gameover()
def take_out_neighbouring_crab(self, lobster):
    for c in self.get_game_objects(Crab):
        if not c.shielded and c.overlaps(lobster):
            c.leave_stage()
            if sound_button.currently_playing:
                sounds.au.play()
            # There is only one crab on the beach, so we set
            # _defeated=True if we found at least one overlapping crab:
            self._defeated = True
            self.schedule_gameover()
```

14.4 Das vollständige Programm

Erneut wollen wir das vollständige Programm abdrucken. Eine kleine Änderung haben wir noch vorgenommen. Die globalen Variablen beach_stage und start_stage sind eigentlich Konstanten. Deshalb schreiben wir Sie nun auch wie Konstanten: mit Großbuchstaben. Dasselbe gilt für die globale Variable sound_button.

```
import random
import pgzrun
from pgzo import *
WIDTH = 560
               # screen width
HEIGHT = 460
               # screen height
TITLE = "Crab" # window title
# "Schaltzentrale" für die Schwierigkeit des Spiels:
START_WITH_LOBSTERS = 1
START_LOBSTER_STRENGTH = 0.2
START_WITH_WORMS = 20
class Button(GameObj):
   def __init__(self, image_normal, image_hover, pos, action):
        self.pos = pos
        self.image = image_normal
        self.image_normal = image_normal
        self.image_hover = image_hover
        self.action = action
        self.hover = False
   def update_image(self):
        if self.hover:
            self.image = self.image_hover
       else:
            self.image = self.image_normal
   def on_mouse_move(self, pos):
        self.hover = self.collidepoint(mouse_state.pos)
        self.update_image()
   def on_mouse_down(self, pos, button):
        if button == mouse.LEFT and self.hover:
            self.action()
class Start(Stage):
```

```
def __init__(self):
       self.background_image = "start"
       self.reset()
   def reset(self):
       self.leave_all()
       self.start_button = Button("start_button_normal", "start_button_hover",
       self.start_button.appear_on_stage(self)
       self.quit_button.appear_on_stage(self)
       SOUND_BUTTON.appear_on_stage(self)
   def draw(self):
       screen.draw.text("The Crab Game", center = (WIDTH//2, HEIGHT * 0.25),
                        color="brown", fontsize=60, fontname="zachary")
   def on_key_up(self, key):
       if key == keys.SPACE:
           self.start_game()
   def start_game(self):
       BEACH_STAGE.reset_game()
       BEACH_STAGE.show()
class Beach(Stage):
   def __init__(self):
       self.background_image = "sand"
   def reset_game(self):
       self.\_level = 1
       self._score = 0
       self.prepare_beach()
   def prepare_beach(self):
       self._defeated = False
       self._victorious = False
       self.leave_all()
       crab = Crab((WIDTH / 2, HEIGHT * 0.7))
       crab.appear_on_stage(self)
       for dummy in range(START_WITH_WORMS):
           w = Beach._create_random_worm()
           w.appear_on_stage(self)
       num_lobsters = START_WITH_LOBSTERS + self._level - 1
       for dummy in range(num_lobsters):
           lobster = Beach._create_random_lobster(START_LOBSTER_STRENGTH)
           lobster.appear_on_stage(self)
       SOUND_BUTTON.appear_on_stage(self)
   def take_out_neighbouring_worms(self, crab):
       for w in self.get_game_objects(Worm):
           if w.overlaps(crab):
               w.leave_stage()
               if SOUND_BUTTON.currently_playing:
                   sounds.blop.play()
               self._score += 1
       if not self._victorious and self.count_game_objects(Worm) == 0:
           self._victorious = True
           self.leave_all(Lobster)
           self.schedule_gameover()
   def take_out_neighbouring_crab(self, lobster):
       for c in self.get_game_objects(Crab):
           if not c.shielded and c.overlaps(lobster):
               c.leave_stage()
```

```
if SOUND_BUTTON.currently_playing:
                    sounds.au.play()
                # There is only one crab on the beach, so we set
                # _defeated=True if we found at least one overlapping crab:
                self._defeated = True
                self.schedule_gameover()
   def restart(self):
        if self._victorious:
           self._level += 1
            self.prepare_beach()
        elif self._defeated:
            START_STAGE.reset();
            START_STAGE.show()
       else:
            raise Exception("restart on unfinished game detected!")
   def schedule_gameover(self):
        clock.schedule_unique(self.restart, 4.0)
   @staticmethod
   def _create_random_lobster(strength):
        result = Lobster((random.randrange(WIDTH), random.randrange(HEIGHT * 0.4)),
                         1 + strength * 4,
                         1 + strength * 9,
                        1 + strength * 24)
        result.turn(random.randrange(360))
        return result
   @staticmethod
   def _create_random_worm():
        return Worm((random.randrange(WIDTH), random.randrange(HEIGHT)))
   def draw(self):
        screen.draw.text("Score: " + str(self._score), topleft= (10,10),
                         color="black", fontsize=20, fontname="zachary")
        screen.draw.text("Level: " + str(self._level), topleft= (10,35),
                         color="black", fontsize=20, fontname="zachary")
       if self._defeated:
            screen.draw.text("You lose!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
       if self._victorious:
            screen.draw.text("You win!", center = (WIDTH//2, HEIGHT * 0.25),
                             color="brown", fontsize=60, fontname="zachary")
class Crab(GameObj):
   IMAGE_PREFIX = "crab"
   IMAGE\_COUNT = 6
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 5
   def __init__(self, pos):
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0
       self.traveled = 0
        self.shielded = False
        self.shield_energy = 10
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Crab.IMAGE_PREFIX + str(image_index)
   def draw(self):
        if self.shielded:
            shield = Actor("shield", self.pos)
            shield.draw()
        energy_block = Actor("energy_block")
```

```
x = WIDTH - 20
        y = 20
        for dummy in range(self.shield_energy):
            energy_block.topright = (x, y)
            energy_block.draw()
            x -= 10
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
        else:
            self.speed = 0
        self.stage.take_out_neighbouring_worms(self)
        if keyboard.left:
            self.turn(10)
        if keyboard.right:
            self.turn(-10)
        if keyboard.up:
            self.speed_up()
        if keyboard.down:
            self.slow_down()
        if keyboard.space and not self.shielded and self.shield_energy > 0:
            self.shielded = True
            self.shield_energy -= 1
            clock.schedule_unique(self.unshield, 3)
   def unshield(self):
        self.shielded = False
   def switch_image(self):
        if self.traveled > Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Crab.IMAGE_COUNT)
            self.traveled -= Crab.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
class Lobster(GameObj):
   IMAGE_PREFIX = "lobster"
   IMAGE\_COUNT = 2
   TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 7
   def __init__(self, pos, speed, drunkenness, jumpiness):
        self._set_image_index(0)
       self.pos = pos
        self.speed = speed
        self.traveled = 0
        self.drunkenness = drunkenness
        self.jumpiness = jumpiness
   def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Lobster.IMAGE_PREFIX + str(image_index)
   def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
            self.turn(25)
        if random.randrange(10) < self.drunkenness:</pre>
            self.turn(random.uniform(-self.jumpiness, self.jumpiness))
        self.stage.take_out_neighbouring_crab(self)
   def switch_image(self):
        if self.traveled > Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Lobster.IMAGE_COUNT)
            self.traveled -= Lobster.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
```

```
class Worm(GameObj):
    IMAGE_PREFIX = "worm"
    IMAGE\_COUNT = 2
    TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS = 3
    def __init__(self, pos):
        self._set_image_index(0)
        self.pos = pos
        self.speed = 0.5
        if random.randrange(2) == 0:
            self.speed = -self.speed
        self.traveled = 0
    def _set_image_index(self, image_index):
        self.image_index = image_index
        self.image = Worm.IMAGE_PREFIX + str(image_index)
    def act(self):
        self.switch_image()
        if self.can_move(self.speed):
            self.move(self.speed)
            self.traveled += abs(self.speed)
        else:
            self.speed = -self.speed
        if random.randrange(100) < 10:</pre>
            self.speed += random.uniform(-1, 1) / 50
    def switch_image(self):
        if self.traveled > Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS:
            self._set_image_index((self.image_index + 1) % Worm.IMAGE_COUNT)
            self.traveled -= Worm.TRAVEL_DISTANCE_BETWEEN_IMAGE_FLIPS
def toggle_sound():
    if SOUND_BUTTON.currently_playing:
        SOUND_BUTTON.image_normal = "sound_off_normal.png"
SOUND_BUTTON.image_hover = "sound_off_hover.png"
        music.pause()
    else:
        SOUND_BUTTON.image_normal = "sound_on_normal.png"
        SOUND_BUTTON.image_hover = "sound_on_hover.png"
        music.unpause()
    SOUND_BUTTON.update_image()
    SOUND_BUTTON.currently_playing = not SOUND_BUTTON.currently_playing
SOUND_BUTTON = Button("sound_on_normal.png", "sound_on_hover.png", (WIDTH-30, HEIGHT-30), toggle_sound)
SOUND_BUTTON.currently_playing = True
music.play("sthlm_sunset")
START_STAGE = Start()
BEACH_STAGE = Beach()
START_STAGE.show()
pgzrun.go()
```

15 Und jetzt?

Sie haben mit dem Crab-Spiel Ihr vermutlich erstes größeres Python-Projekt erfolgreich absolviert. Wir haben eine Menge über Python gelernt. Allerdings haben wir auch einiges weggelassen, was vielleicht in einem Ihrer nächsten Spiele relevant sein könnte:

- while-Schleifen,
- Details zu Listen und Tupeln,
- Verarbeitung von Strings, Zugriff auf einzelne Zeichen,
- Formatierung von Strings,
- Datei-Eingabe und Datei-Ausgabe (z. B. zum Speichern von Spielständen),
- rekursive Funktionen,
- mehr Details zur Vererbung von Klassen, insb. zu super()
- mehr Details zur Handhabung unveränderlicher Datentypen wie int, float, bool, str,
- mehr Details zum Interpreter (Byte-Code, Python-VM).

Sie sehen, es kann sich lohnen, ein einführendes Python-Buch⁴⁹ zur Hand zu nehmen, um die vielen Konzepte, die Sie in diesem Buch gelernt haben, noch einmal von anderer Seite und systematisch dargestellt zu bekommen, und um neue Konzepte zu lernen, die wie hier ausgelassen haben. Ihnen wird zwar bei der Lektüre vieles bekannt vorkommen. Aber dadurch kommen Sie sicher auch schnell voran.

Um sich nun zu vertiefen bietet sich folgendes an:

- Überlegen Sie sich selbst eine Spielidee.
- Während Sie das Python-Buch lesen, programmieren Sie Ihr Spiel. Sobald Sie ein neues Konzept gelernt haben, überlegen Sie, wo Sie dieses in Ihrem Spiel einsetzen können, und tun es dann.
- Wenn Sie sich von Pygame Zero lösen wollen, sollten Sie Pygame (ohne Zero) besser kennen lernen. Setzen Sie sich also intensiver mit Pygame auseinander. Es gibt viele hilfreiche Tutorials auf der Projekt-Webseite⁵⁰. Ein für den Pygame-Einstieg geeignetes deutsches Tutorial habe ich in einem Wiki⁵¹ einer Spieleprogrammierer-Community gefunden.
- Wenn Sie bei Pygame Zero bleiben wollen, können Sie die Bibliothek pgzo.py weiterhin einsetzen. Schauen Sie sich den Programmcode der Bibliothek an. Sie können die Klassen dieser Bibliothek selbst verändern, wenn dies für Ihre Spielidee sinnvoll erscheint.

Viel Erfolg!

⁴⁹ Ein gutes Einführungsbuch wird in Fußnote 7 auf Seite 5 genannt.

⁵⁰ https://www.pygame.org/wiki/tutorials

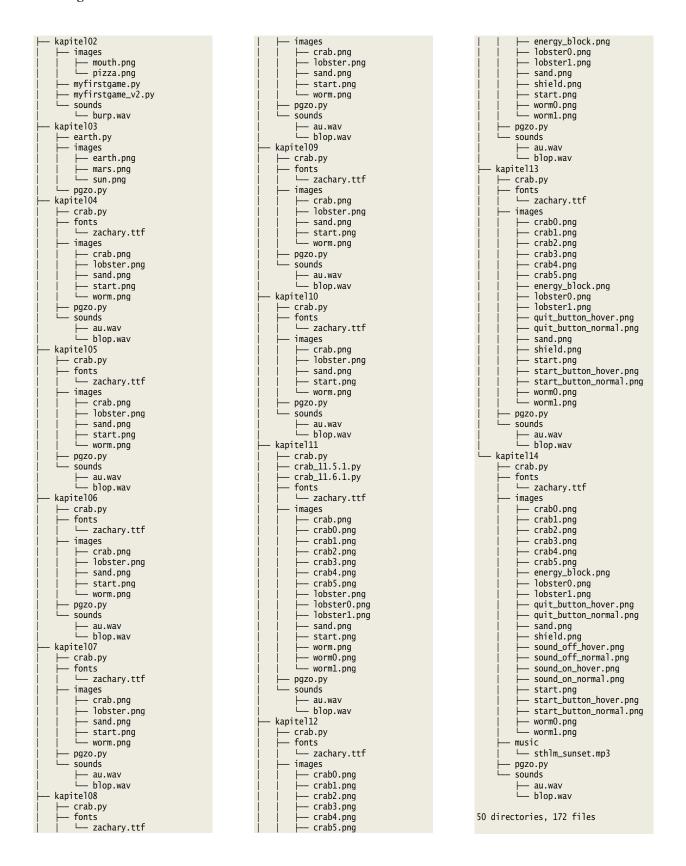
⁵¹ https://www.spieleprogrammierer.de/wiki/Pygame-Tutorial

16.1 Dateien 131

16 Anhang

16.1 Dateien

Die Quellcode-Dateien zu jedem Kapitel sind in einem ZIP-Archiv verfügbar. Der Inhalt des Archivs ist hier aufgelistet:



16.2 Dokumentation zu pgzo.py

DGZO (version 0.9)

```
Class library from the introductory book.

This module exports two classes ``Stage`` and ``GameObj`` and a global object ``mouse_state``.

Also this module implements all hook methods of Pygame Zero, i. e. ``draw``, ``update``, ``on_mouse_down``, ``on_mouse_up``, ``on_mouse_move``, ``on_key_down``, ``on_key_up``. All hook methods delegate to the current stage's respective methods (see ``Stage.show()`` and ``Stage.current``).
```

Classes

class GameObj(pgzero.actor.Actor)

An actor on stage.

```
On the current stage there can be several *game objects*. A game object has a position and a rotation angle, a current speed and a current image.

A game object has got an ``act`` method that takes care of all event processing and other game state manipulations. Also there is a ``draw`` method that will draw the image onto the screen. The ``act`` method usually is overridden by subclasses in order to add game specific behaviour. The ``draw`` method should only be overridden in order to tweak visual appearance of the game object.
```

Method resolution order:

GameObj

pgzero.actor.Actor

builtins.object

Methods defined here:

__init___(self, image=None, pos=None, speed=None, orbit_center=None, stage=None, center_drawing_color=None, pos_drawing_color=None, rect_drawing_color=None,

**kwargs)

Create a game object with ``image`` and ``center`` position .

In contrast with ``Actor`` all parameters are optional in order to support client-side initialization by setting attributes.

When an ``image`` is missing, we use a 1x1 pixel transparen t image instead.

If ``center_drawing_color``, ``pos_drawing_color``, or ``rect_drawing_color`` are given, then this class will draw a center point, a coordinate tuple, or a bounding rectangle respectively.

act(self)

Act - empty method, game objects have no default action.

appear_on_stage(self, stage)

Put this game object on a stage.

If this game object currently is on a stage, it is taken ou $\ensuremath{\mathsf{t}}$

first by automatically calling leave_stage.

can_move(self, distance=None, edge=(0, 0))

Check, if we can move forward without leaving the stage.

If we get beyond one of the edges of the stage, return ``False``.

draw(self)

Draw image and additional artifacts.

This additionally draws a center point, a coordinate tuple, or a bounding rectangle, if the respective attributes have been set to a color.

full_orbits(self)

Return the accumulated full orbits.

A full orbit is 360 degrees.

is_beyond_stage_edge(self, edge=(30, 30))

Test if we are beyond the edges of the stage.

Return ``True`` is we are.

leave_stage(self)

This game object leaves it's current stage.

If this object is not on a stage, a ``RuntimeError``

is raised.

move(self, distance=None)

Move forward in the current direction.

next_hop(self, distance=None)

Get (x,y) tuple where a ``move(distance)`` call would finish.

on_key_down(self, key, mod, unicode)

Empty method, game objects have no default event handling.

on_key_up(self, key, mod)

Empty method, game objects have no default event handling.

on_mouse_down(self, pos, button)

Empty method, game objects have no default event handling.

on_mouse_move(self, pos, rel, buttons)

Empty method, game objects have no default event handling.

on_mouse_up(self, pos, button)

Empty method, game objects have no default event handling.

orbit(self, angle)

Orbit around ``orbit_center`` attribute by ``angle`` degrees.

Orbit goes towards the left (counter clockwise).

overlaps(self, other)

Check for pixel-exact overlap of two game objects.

slow down(self)

Increment current speed by -0.1.

The resulting speed is floored by a minimum (negative) speed.

speed_up(self)

Increment current speed by 0.1.

The resulting speed is ceiled by a maximum speed.

turn(self, angle=1)

Turn ``angle`` degrees towards the left (counter clockwise)

Static methods defined here:

___**new**___(typ, *args, **kwargs)

Create and return a new object. See help(type) for accurat e signature.

Data descriptors defined here:

image

Image name or ``None``.

mask

An image mask for collision detection.

rect

The rectangle around this object.

Methods inherited from pgzero.actor.Actor:

angle_to(self, target)

Return the angle from this actors position to target, in degrees.

distance_to(self, target)

Return the distance from this actor's pos to target, in pixels.

Data descriptors inherited from pgzero.actor.Actor:

anchor

angle

pos

X

y

class Stage(builtins.object)

```
The game can consist of several stages.
```

There is only one stage visible at the same time. All drawing and all updates of game state will be done in the active stage. A stage usually contains many game objects, that each perform individual ``draw`` and ``update`` operations.

Methods defined here:

__init__(self, background_image=None)

Create an empty stage with a background.

If ``background_image`` is ``None``, we draw a
white background.

count_game_objects(self, cls=<class 'object'>)

Return number of game objects of given class.

draw(self)

Draw Background and dispatch ``draw`` call to all game objects.

get_game_objects(self, cls=<class 'object'>)

Iterate this stage's game objects of given class.

is_beyond_edge(self, pos, edge=(0, 0))

Check, if ``pos`` is beyond the stage's edges.

With the ``edge`` parameter you can define a tuple of x, y coordinates which makes the stage smaller on each of the four sides.

is_on_stage(self, game_obj)

Check if ``game_obj`` is on this stage.

leave_all(self, cls=<class 'object'>)

Let all game objects of given class leave this stage.

on_key_down(self, key, mod, unicode)

Dispatch ``on_key_down`` call to all game objects.

on_key_up(self, key, mod)

Dispatch ``on key up`` call to all game objects.

on_mouse_down(self, pos, button)

Dispatch ``on_mouse_down`` call to all game objects.

on_mouse_move(self, pos, rel, buttons)

Dispatch ``on_mouse_move`` call to all game objects.

on_mouse_up(self, pos, button)

Dispatch ``on_mouse_up`` call to all game objects.

show(self)

Make this stage the current stage.

update(self)

Dispatch ``act`` call to all game objects.

Static methods defined here:

__new__(typ, *args, **kwargs)

Create and return a new object. See help(type) for accurate signature.

class MouseState(builtins.object)

The current state of the mouse.

```
Each attribute represents a mouse button. For example, ::
    mouse state.left
is ``True`` if the left button is depressed, and ``False``
otherwise.
By calling ::
    mouse_state.pos
the current mouse position will be returned.
In order to check, whether the mouse position has changed since
the last call, use ::
    mouse state.moved
Note, that an immediately following call to ``mouse.moved`` will
always return ``False``. The value will be ``True`` after the next
mouse movement.
 Data descriptors defined here:
 pos
       Current mouse position.
 moved
       ``True`` if mouse has been moved since last check.
```

Functions

update()

Pygame Zero global hook method.

Data

mouse_state = <pgzo.MouseState object>