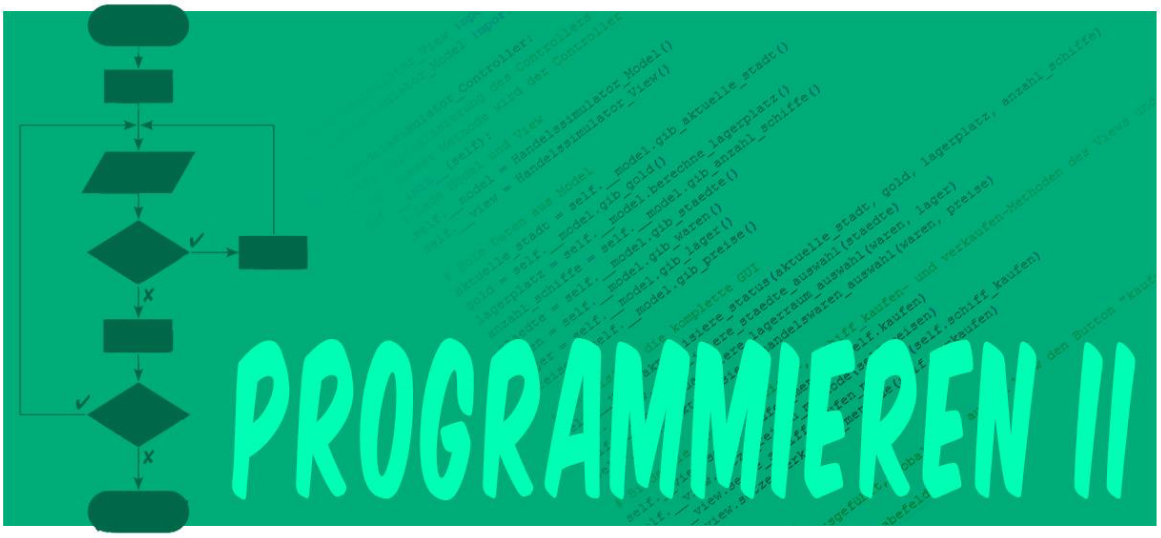
creative commons

Wissensfabrik
Mehr Wissen. Mehr Können. Mehr Zukunft.

# IT2School

**Gemeinsam IT entdecken**




# Modul A3 – Programmieren II

## Objektorientierte Programmierung mit Python

Eine Entwicklung von

OFFIS
INSTITUT FÜR INFORMATIK

In Kooperation mit

Carl von Ossietzky
Universität
Oldenburg

Im Auftrag der

Wissensfabrik
Mehr Wissen. Mehr Können. Mehr Zukunft.

# Inhalt


1 Objektorientierte Programmierung mit Python ..... 3
2 Warum gibt es das Modul? ..... 4
3 Ziele des Moduls ..... 4
4 Die Rolle der Unternehmensvertreterin/des Unternehmensvertreters ..... 4
5 Inhalte des Moduls ..... 5
5.1 Python ..... 5
5.2 Entwicklungsumgebung Geany ..... 5
6 Unterrichtliche Umsetzung ..... 6
6.1 Grober Unterrichtsplan ..... 7
6.2 Stundenverlaufsskizzen ..... 8
6.2.1 Einstieg in Python ..... 8
6.2.2 Einführung in die objektorientierte Programmierung ..... 8
6.2.3 Entwicklung eines Memory-Spiels ..... 9
6.2.4 Vertiefung in die objektorientierte Programmierung ..... 10
7 Einbettung in verschiedene Fächer und Themen ..... 11
8 Anschlussthemen ..... 11
9 Literatur und Links ..... 12
10 Arbeitsmaterialien ..... 12
11 Glossar ..... 13



Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# 1 Objektorientierte Programmierung mit Python



Möchte man größere und aufwendigere Programme schreiben, dann ist es zwingend erforderlich, sich auch tiefergehend mit der Programmierung an sich zu beschäftigen.

Die Schülerinnen und Schüler befassen sich im Rahmen dieses Moduls mit der textuellen Programmiersprache Python und steigen dabei mit einfach Beispielen, wie das Nachprogrammieren eines Programms von Scratch nach Python ein. Im Anschluss daran lernen die Schülerinnen und Schüler Grundkonzepte der Programmierung kennen, wie beispielsweise die objektorientierte Programmierung, Vererbung, Datenkapselung und Entwurfsmuster. Alle Inhalte werden praktisch angewendet, so haben Schülerinnen und Schüler die Möglichkeit eigene Spiele mit einer grafischen Oberfläche zu programmieren

| | | |
|---|---|---|
| **Lernfeld/Cluster:** | Daten erforschen | |
| **Zielgruppe/Klassenstufe:** | | 4. bis 5. Klasse |
| | | 6. bis 7. Klasse |
| | **X** | 8. bis 10. Klasse |
| | **X** | 11. bis 12. Klasse |
| **Geschätzter Zeitaufwand:** | 8 – 10 Einzelstunden | |
| **Lernziele:** | • Kennenlernen von Python als Programmiersprache.<br>• Überführung einfacher Scratchprogramme in Python.<br>• Prinzip der objektorientierten Programmierung am Beispiel von Python kennenlernen.<br>• Datenkapselung, Vererbung und Entwurfsmuster (speziell MVC) im Zusammenhang mit der objektorientierten Programmierung kennlernen.<br>• Entwicklung und Ausbau von Programme in Python unter Verwendung der objektorientierten Programmierung.<br>• Gestaltung von grafischen Oberflächen in Python mit der Bibliothek *Tkinter*. | |
| **Vorkenntnisse der Schülerinnen und Schüler:** | Erforderlich:<br>• Grunderfahrung in der Programmierung mit Scratch | |
| **Vorkenntnisse der/des Lehrenden:** | Erforderlich:<br>• Grunderfahrung in der Programmierung in Scratch<br>Empfohlen:<br>• Weitere Programmiererfahrung mit anderen Programmiersprachen | |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | |
|---|---|
| **Vorkenntnisse der Unternehmensvertreterin/des Unternehmensvertreters** | Keine |
| **Sonstige Voraussetzungen:** | Erforderlich:<br>• Ausreichende Anzahl an Computern<br>• Python und Geany sollten vorinstalliert sein |

## 2 Warum gibt es das Modul?

Dieses Modul baut auf das Modul *B5 – Programmieren* auf. Die Erfahrungen mit der Programmieroberfläche Scratch werden in diesem Modul aufgegriffen und in eine textuelle Programmiersprache übersetzt. Die Schülerinnen und Schüler erhalten damit einen Einblick in die erweitere Programmierung und deren komplexe Entwicklung.

Der Schwerpunkt dieses Moduls liegt auf der objektorientierten Programmierung, die ein wichtiger Bestandteil der zeitgemäßen und praxisnahen Anwendungsentwicklung ist. Darüber hinaus wird der Gebrauch von Entwurfsmustern aufgezeigt. Dies ermöglicht es für wiederkehrende Probleme und Anforderungen Lösungsschemata zu verwenden.

Zielsetzung dieses Moduls ist es, die Schülerinnen und Schüler für die Programmierung von komplexeren Anwendungen zu motivieren und ihnen einen praxisnahen Einblick zu gewähren. Durch die Umsetzung von eigenen Projekten können die Schülerinnen und Schüler kreative Ideen umsetzen und die Möglichkeit zur Mitgestaltung in der digitalen Gesellschaft erfahren.

## 3 Ziele des Moduls

- Kennenlernen von Python als Programmiersprache.
- Überführung einfacher Scratchprogramme in Python.
- Prinzip der objektorientierten Programmierung am Beispiel von Python kennenlernen.
- Datenkapselung, Vererbung und Entwurfsmuster (speziell MVC) im Zusammenhang mit der objektorientierten Programmierung kennlernen.
- Entwicklung und Ausbau von Programme in Python unter Verwendung der objektorientierten Programmierung.
- Gestaltung von grafischen Oberflächen in Python mit der Bibliothek *Tkinter*..

## 4 Die Rolle der Unternehmensvertreterin/des Unternehmensvertreters

Im *Modul A3 – Programmieren II* hat die Unternehmensvertreterin/der Unternehmensvertreter mehrere Möglichkeiten aktiv mitzuwirken. Hier einige Anregungen:

- Unterstützung der Lehrkraft, z. B. bei der Einführung in die Programmierung.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

- Unterstützung der Schülerinnen und Schüler bei der Umsetzung ihrer eigenen Projekte.
- Sie oder er kann zur Abschlusspräsentation der Projektergebnisse als Special-Guest eingeladen werden.

# 5 Inhalte des Moduls

## 5.1 Python

Bei Python handelt es sich, ähnlich wie bei Scratch aus B5 Programmieren, um eine Programmiersprache. Anders als Scratch handelt es sich bei Python jedoch um eine textuelle Programmiersprache, die universell einsetzbar ist und auf verschiedenen Systemen Anwendung findet. Dies bedeutet, dass die Anweisungen, die vom Programm ausgeführt werden, in Textform (dem sogenannten Quellcode bzw. Sourcecode) verfasst wird. Anschließend kann dieser Quellcode auf jedem System, das über die Programmiersprache (bzw. den Interpreten) Python verfügt, ausgeführt werden.

Für die Entwicklung von Programmen existierten viele verschiedene Umgebungen. In diesem Modul beschränken wir uns auf die Entwicklungsumgebung Geany. Jedoch ist es auch möglich, das Programm mit dem ganz gewöhnlichen Editor (nicht zu verwechseln mit Word oder Wordpad) zu schreiben und dem Standardterminal unter Windows, Linux etc. auszuführen.

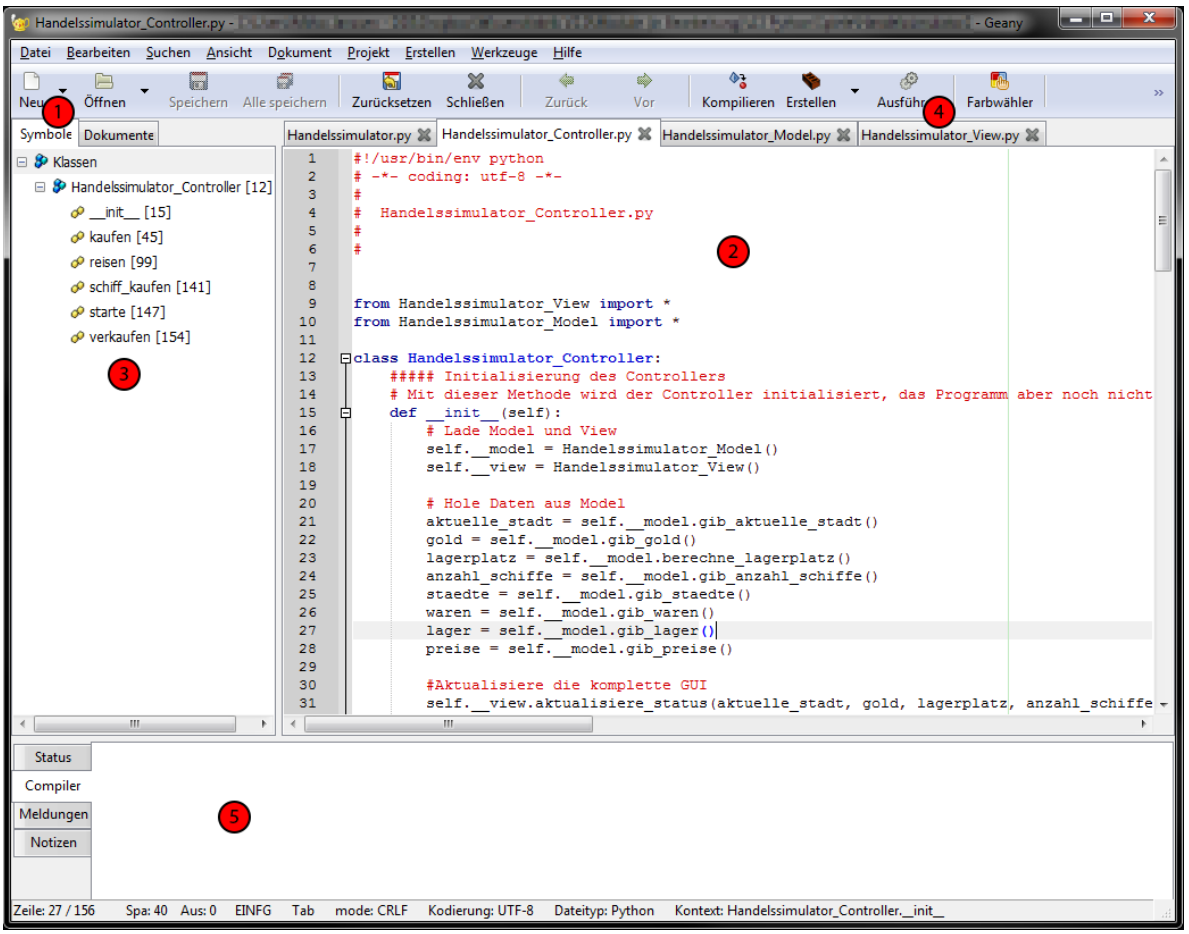
Python liegt in zwei Versionen vor und kann von der offiziellen Webseite http://python.org/download/ ganz einfach als Setup runtergeladen und installiert werden. Wir verwenden die Version 2.7 in diesem Modul.

Im Gegensatz zu anderen Programmiersprachen werden in Python Befehle nicht durch ein Semikolon beendet und Schleifen oder Bedingungen durch geschweifte Klammern eingegrenzt. Im letzten Fall werden stattdessen die entsprechenden Anweisungen eingerückt. Dies soll dafür sorgen, dass der Quellcode im Vergleich zu anderen Sprachen möglichst lesbar und damit anfängerfreundlich ist.

## 5.2 Entwicklungsumgebung Geany

Geany ist eine Entwicklungsumgebung für verschiedene Programmiersprachen (darunter auch Python) und kann unter Linux, OS X und Windows verwendet werden. Die Entwicklungsumgebung besitzt hilfreiche Funktionen, wie zum Beispiel Übersichten von Methoden oder Autovervollständigung. Geany lässt sich dabei ganz einfach wie übliche Entwicklungsumgebungen benutzen. Es folgen eine Grafik der Entwicklungsumgebung, inklusiver der Kennzeichnung einiger wichtiger Elemente sowie deren Erläuterung:

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.




1. Hier kann eine neue Datei angelegt werden. Wichtig zu erwähnen ist hier, dass verschiedene Formate (.py für Python) als Vorlagen ausgewählt werden können.
2. In diesem Bereich befindet sich der Quellcode des Programms.
3. Hier werden die Klassen und deren Methoden aus der aktuellen Datei aufgelistet.
4. Mit diesem Button kann das Programm ausgeführt werden.
5. In diesem Bereich werden alle Benachrichtungen in Bezug auf das Kompilieren bzw. Ausführen des Programmes angezeigt. Fehlermeldungen werden grundsätzlich im Terminal des ausgeführten Programmes angezeigt.

# 6 Unterrichtliche Umsetzung

In diesem Modul steht die selbstständige Arbeit der Schülerinnen und Schüler im Vordergrund. Zu Beginn werden die Schülerinnen und Schüler angeleitet, neue Konzepte zu lernen. Danach können die Schülerinnen und Schülern sich eigenverantwortlich in die Thematik vertiefen und auch Arbeitsaufträge umsetzen.

Als Lehrkraft sind sie daher zum großen Teil als Lernbegleitung im Einsatz. Dies bedeutet, dass sie Materialien und Aufgabenstellungen vorgeben, jedoch bei der Problemlösung nur wenig eingreifen bzw. den Weg zur Lösung nicht direkt vorgeben.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

Nach Abschluss des Moduls verfügen die Schülerinnen und Schüler über ein erweitertes Verständnis bzgl. der Programmierung (im Vergleich zu dem Modul B5). Die Schülerinnen und Schüler sind nun in der Lage, eigene Projekte und Ideen zu entwickeln und umzusetzen.

.

## 6.1 Grober Unterrichtsplan

| Unterrichtsszenarien | Kurze Zusammenfassung |
|---|---|
| **Einstieg** | Python sowie Geany werden vorgestellt und die Schülerinnen und Schüler überführen ein Programm von Scratch nach Python. |
| **Erarbeitung** | Die Schülerinnen und Schüler machen erste Erfahrungen mit dem Konzept der objektorientierten Programmierung und der Entwicklung von grafischen Oberflächen. |
| **Vertiefung** | Die Schülerinnen und Schüler programmieren ohne direkte Anleitung ein Memory mit grafischer Oberfläche. |
| **Erarbeitung** | Das Konzept der objektorientierten Programmierung wird vertieft und Themen wie Datenkapselung, Vererbung und Entwurfsmuster werden behandelt. |
| **Vertiefung** | Die Schülerinnen und Schüler arbeiten sich in ein vorgegebenes unfertiges Spiel (Handelssimulator) ein und stellen dieses fertig. |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

## 6.2 Stundenverlaufsskizzen

**Abkürzungen/Legende**
AB = Arbeitsblatt/Arbeitsblätter; L = Lehrkraft; MuM = Mitschülerinnen und Mitschüler; SuS = Schülerinnen und Schüler;
UV = Unternehmensvertreterin/Unternehmensvertreter

### 6.2.1 Einstieg in Python

| Zeit | Phase | Sozialform/ Lehrerimpuls | Inhalt/Unterrichtsgeschehen | Material |
|---|---|---|---|---|
| 5 Min. | Einstieg | Plenum | Begrüßung; Vorstellung von Python und Geany | |
| 25 Min. | Erarbeitung | Einzel-/Partnerarbeit | Die SuS befassen sich mit A3.1 beschäftigen.<br>L steht bei Problemen mit der Entwicklungsumgebung zur Verfügung.<br>Wichtig ist, dass die L nicht direkt Lösungen vorgibt, da die Anleitung zum einen das Machen eines bestimmten Fehlers vorsieht und zum anderen sehr detaiiert ist.<br>Zum Abschluss kann der Lehrer auch die Befehlsübersicht A3.8 austeilen. | A3.1, Computer, ggf. A3.8 |
| 15 Min. | Abschluss | Plenum | Einige Programme können von den SuS präsentiert und Aufgabe 4 kann besprochen werden. Zum Abschluss kann die L noch einen Überblick über die kommenden Stunden geben. | Computer, Beamer |

### 6.2.2 Einführung in die objektorientierte Programmierung

| Zeit | Phase | Sozialform/ Lehrerimpuls | Inhalt/Unterrichtsgeschehen | Material |
|---|---|---|---|---|
| 5 Min. | Einstieg | Plenum | Begrüßung; Ablauf der Unterrichtsstunde klären | |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | | | | |
|---|---|---|---|---|
| 15 Min. | Erarbeitung | Lehrervortrag oder Einzel-/Partnerarbeit | Einstieg in die objektorientierter Programmierung; Möglich durch Lehrervortrag, oder durch Austeilen und selbstständiges Arbeiten mittels A3.2.<br>Hier sollte deutlich werden, dass die objektorientierte Programmierung wichtig für die kommenden Stunden ist: Bei der Programmierung einer grafischen Oberfläche wird ausschließlich mit Klassen und Objekten gearbeitet! | A3.2 |
| 20 Min. | Vertiefung | Einzel-/Partnerarbeit | SuS arbeiten sich in die Grundlage der grafischen Oberflächen mittels des Tutorials A3.3 ein. | A3.3, Computer |
| 5 Min. | Abschluss | Plenum | Im Plenum sollte der Inhalt der Unterrichtsstunde kurz reflektiert und besprochen werden. Ein Ausblick auf die kommende Stunde kann ebenfalls gegeben werden. | |

### 6.2.3 Entwicklung eines Memory-Spiels

| Zeit | Phase | Sozialform/ Lehrerimpuls | Inhalt/Unterrichtsgeschehen | Material |
|---|---|---|---|---|
| 5 Min. | Einstieg | Plenum | Begrüßung; Besprechung des Ablaufs dieser und nächsten Unterrichtsstunde | |
| 70 Min.[1] | Erarbeitung | Einzel-/Partnerarbeit | Die SuS entwickeln selbstständig ein eigenes Memory-Spiel. Als Grundlage erhalten sie das AB A3.4 und das ZIP-Archiv A3.5. Das AB A3.4 bietet ein paar Hilfestellungen für mögliche Probleme, ebenso wie die L als Hilfestellung bereitstehen sollte. | A3.4, A3.5, Computer |
| 15 Min. | Abschluss | Plenum | Zum Abschluss können alle oder einige Programme im Plenum präsentiert werden. Es ist auch möglich, die verschiedenen Programm zum Austausch bereitzustellen. | |

[1] Wir gehen hier handlicherweise von einer Doppelstunde aus, jedoch kann der Inhalt auch auf zwei Einzelstunden verteilt werden.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

### 6.2.4 Vertiefung in die objektorientierte Programmierung

| Zeit | Phase | Sozialform/ Lehrerimpuls | Inhalt/Unterrichtsgeschehen | Material |
|---|---|---|---|---|
| 5 Min. | Einstieg | Plenum | Begrüßung; Ablauf der Unterrichtsstunde klären | |
| 20 Min. | Erarbeitung | Lehrervortrag oder Einzel-/Partnerarbeit | Einstieg in den zweiten Teil der objektorientierter Programmierung; Möglich durch Lehrervortrag, oder durch Austeilen und selbstständiges Arbeiten mittels A3.6.<br>Hier sollte deutlich werden, dass die objektorientierte Programmierung wichtig für die kommenden Stunden ist: Bei der Programmierung einer grafischen Oberfläche wird ausschließlich mit Klassen und Objekten gearbeitet! | A3.6 |
| 95 Min.[2] | Erarbeitung | Einzel-/Partnerarbeit | Die SuS arbeiten sich in den Handelsimulator ein und erweitern diesen. | A3.7, Computer |
| 15 Min. | Abschluss | Plenum | Zum Abschluss können alle oder einige Programme im Plenum präsentiert werden. Es ist auch möglich, die verschiedenen Programme zum Austausch bereitzustellen. | |

[2] Bei einer starken Klasse könnten auch nur zwei Unterrichtsstunden reichen, jedoch sind drei zu empfehlen.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# 7 Einbettung in verschiedene Fächer und Themen

Als Einbettung in ein speziellen Unterrichtsfach bietet sich in erster Linie die Informatik an. Dies liegt vor allem in der Tatsache, dass es sich bei diesem Modul um eine Vertiefung der Programmierung handelt. Da gerade die Informatik nicht in allen Bundesländern fester Bestandteil der Schuldbildung ist und einige Schulen kein Unterrichtsfach in Richtung der Informatik anbieten, würde es sich als Alternative anbieten, dieses Modul fächerübergreifend im Rahmen einer Projekt- bzw. Themenwoche einzubinden.

Die folgenden Kompetenzen finden sich entweder in den Bildungsstandards der Kultusministerkonferenz oder in den einzelnen Rahmenlehrplänen der Länder wieder:

**Informatik**

Die Schülerinnen und Schüler …

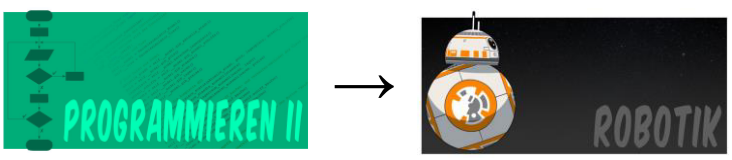
- erstellen Produkte unter Anwendung fortgeschrittener Techniken von Standardsoftware; falls es das Produkt erfordert, arbeiten sie sich in geringem Umfang in Spezialsoftware ein.
- erwerben beim Bearbeiten von Softwareprojekten in angemessenem Umfang Kenntnisse über Analyse- und Modellierungsverfahren sowie Projektmanagement.
- verstehen Programmabläufe und die Arbeitsweise von Schnittstellen.
- können erdachte Systeme in technische Systeme übertragen.
- kennen sich in Entwicklungsumgebungen/Programmierumgebungen aus.
- eine Spielumgebung entsprechend der Spielidee gestalten.
- eine Spielvariante mit informatischen Werkzeugen entwerfen und implementieren.

# 8 Anschlussthemen

Als Anschlussthemen im Zusammenhang mit IT2School bietet sich folgendes Modul an:

**Beispiel: Ein Einstieg in eine weitere textuelle Programmiersprache**

Im Erweiterungsmodul E3 Robotik haben die Schülerinnen und Schüler die Möglichkeit einen eigenen BB8 (ein Roboter aus dem Star Wars Universum) zu bauen. Bei diesem Bauvorhaben kann auf den Arduino zurückgegriffen werden. Dies bietet die Möglichkeit sich nochmals einer anderen textuellen Programmiersprache zu widmen. Ein wichtiger Unterschied im Vergleich zu der bisheriegen Programmierung ist dann, dass mit der dort verwendeten Programmiersprache unter anderem Motoren angesteuert werden können.




**Beispiel: Verschiedene Vertiefungsmöglichkeiten**

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

Verfügen die Schülerinnen und Schüler ersteinmal über ein gewisses Potenzial in der Programmierung, so kann hierauf hinsichtlich verschiedener Vertiefungsmöglichkeiten aufgebaut werden. So wäre es beispielsweise möglich sich intensiver mit dem Entwicklungsprozess bzw. der Planung und Konzeption von Anwendungen zu beschäftigen. Schülerinnen und Schüler könnten sich in dem Fall in dem Bereich von UML-Diagrammen oder Ähnlichem vertiefen. Eine weitere Vertiefungsmöglichkeit stellt auch das Thema Datenbanken dar. Hier könnten die Schülerinnen und Schüler eigene Datenbanken für gewisse Aufgabenstellungen entwerfen und anschließend in ihre Programme integrieren.

# 9 Literatur und Links


- Glöde, Martina & Reit, Birgit (Hg.) (2015): **Programmieren supereasy. Einfacher Einstieg in Scratch und Python**. München: Dorling Kindersley.
- Onlie-Kurs bzw. -Tutorial zu den Grundlagen von Python – online: http://www.python-kurs.eu/kurs.php
- **Online-Kurs bzw. -Tutorial zu den Grundlagen der Gestaltung von grafischen Oberflächen mit Tkinter in Python** – online: http://www.tkdocs.com/tutorial/index.html
- **Lernplattform für Python** sowie die Auswertung von Daten mittels Python – online: https://www.datacamp.com


# 10 Arbeitsmaterialien

| Nr. | Titel | Beschreibung |
|---|---|---|
| A3.1 | Einstieg | Bietet einen Einstieg in Python, wobei ein Programm/Spiel von Scratch in Python überführt wird. |
| A3.2 | Objektorientierte Programmierung – Teil 1 | Einstieg in die objektorientierte Programmierung. Erklärt, was es ist und gibt Beispiele zur Verwendung von Klassen und Objekten. |
| A3.3 | Einstieg grafische Oberflächen | Vorstellung der Bibliothek Tkinter zur Gestaltung von grafischen Oberflächen. Enthält auch ein Beispiel Programm als Tutorial. |
| A3.4 | Memory | Aufgabenstellung zur Gestaltung eines eigenen Memorys. Beinhaltet auch Problemstellungen inklusive deren Lösungen. |
| A3.5 | Memory – Vorlage | Vorlage für Schülerinnen und Schüler zur Aufgabenstellung in A3.4. |
| A3.6 | Objektorientierte Programmierung – Teil 2 | Der zweite Teil der objektorientierten Programmierung wird vorgestellt. Inhaltlich umfasst dieses Material die Vererbung, Datenkapselung und das Entwurfsmuster MVC. |


Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | | |
|---|---|---|
| A3.7 | Handelssimulator | Aufgabenstellung zur Einarbeitung in eine Rohvariante des Handelssimulator sowie der zustzlichen Aufgabenstellung zur Erweiterung dieses Programms/Spiels. |
| A3.8 | Befehlsübersicht | Eine Gegenüberstellung einiger Bausteine aus Scratch zu den Befehlen aus Python. |
| A3.9 | Installationsanleitung | Kurze Einführung in Python und Geany sowie eine Installationsanleitung. |

**Legende**

Material für Schülerinnen und Schüler

Material für Lehrkräfte sowie Unternehmensvertreterinnen und Unternehmensvertreter

Zusatzmaterial

# 11 Glossar

| Begriff | Erläuterung |
|---|---|
| **Datenkapselung** | Bezeichnet die Zugriffsbegrenzung von Attribute (und damit auch werden Daten/werten). Bei der Definition einer Klasse können deren Attribute als öffentlich, privat oder geschützt gekennzeichnet werden. |
| **Entwurfsmuster** | Als Entwurfsmuster werden Lösungsschablonen zu wiederkehrenden Entwicklungsproblemen bezeichnet. Dies kann zum Beispiel das Problem bzw. der Wunsch sein, dann bei der Entwicklung einer Anwendung die Logik, die Datenquellen sowie die grafischen Oberflächen im Quellcode von einander getrennt werden. |
| **Geany** | Eine Entwicklungsumgebung für mehrere Programmiersprachen, so auch für Python. |
| **MVC** | Ein Entwurfsmuster, dann bei der Entwicklung einer Anwendung zum Einsatz kommt, wenn die Logik, die Datenquellen sowie die grafischen Oberflächen im Quellcode von einander getrennt werden sollen. |
| **Objektorientierte Programmierung** | Ein Paradigma der Programmierung, bei dem Objekte und Klassen als Grundstrukturen der Wirklichkeit in der Konzeption von Anwendungen entworfen und verwendet werden. |
| **Scratch** | Eine blockbasierte Programmiersprache, welche sehr gut für junge Schülerinnen und Schüler als Einstieg in die Programmierung geeignet ist. |
| **Vererbung** | Ein grundlegensten Konzepte der objektorientierten Programmierung, welche Beziehungen zwischen |

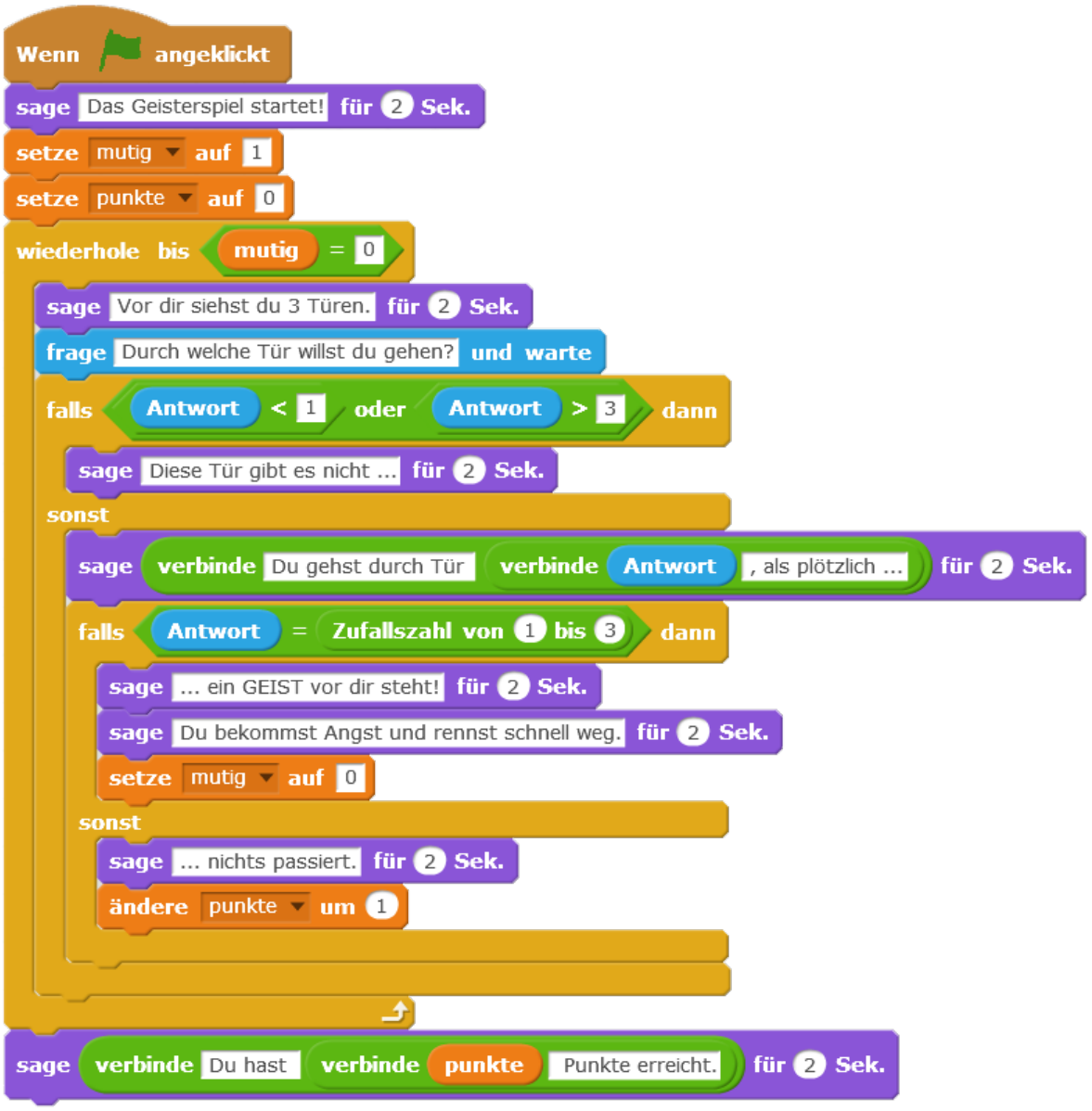
Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | |
|---|---|
| | Klassen/Objekten zulässt. Methoden und Attribute können dabei von der Mutterklasse auch die abgeleiteten Klasse vererbt werden. |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Das Geisterspiel

Zum Einstieg in Python, wollen wir zunächst ein kleines Spiel programmieren. Dieses Spiel handelt von mehreren Türen und einem Geist. Damit du weißt, was genau auf dich zu kommt, kannst du dir das folgende Scratch-Skript anschauen. Hier wurde das Spiel bereits einmal programmiert.




## Aufgabe 1

Schau dir das obige Skript genau an und versuche die einzelnen Abschnitte zu verstehen. Beschreibe kurz für dich den Ablauf des Spiels.

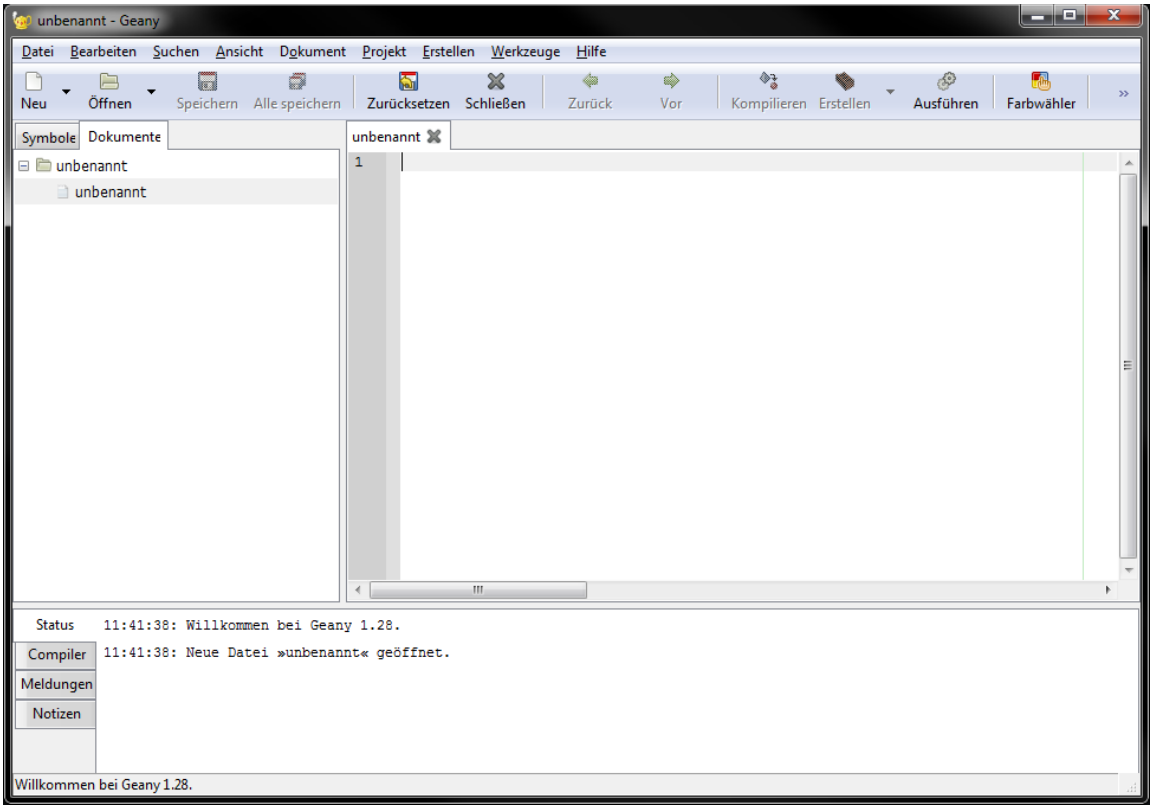
## Aufgabe 2

Arbeite die folgenden Schritte sorgfälltig durch.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

## Schritt 1: Die Entwicklungsumgebung starten

Zunächst einmal musst du die Entwicklungsumgebung für Python starten. Sofern dir dein Lehrer nichts anderes gesagt hat, startest du das Programm *Geany*. Du findest es entweder auf dem *Desktop* oder im *Startmenü* und *Alle Programme*. Das Fenster sieht dann ungefähr so aus:




## Schritt 2: Die Programmierung

Wir werden nun die Bausteine aus dem Scratch-Skript Baustein für Baustein von oben nach unten durchgehen. Dabei kannst du immer wieder vergleichen, welcher Befehl in Python zu dem Baustein in Scratch passt.

Den ersten Baustein *Wenn grüne Fahne angeklickt* können wir ignorieren, daher starten wir erst einmal mit den weiteren drei Bausteinen. Schreibe folgendes (**ohne** die Zeilennummern) in *Geany* ab:

```
01  print("Das Geisterspiel startet!")
02  mutig = 1
03  punkte = 0
```

**Anmerkung:** Der Befehl *print* sorgt dafür, dass Ausgaben auf dem Bildschirm angezeigt werden. Das setzen von Varianten ist in Python auch ganz einfach. Du kannst einfach die Variable gleich ihrem Wert setzen.

Bei dem nächsten Baustein handelt es sich um eine Schleife, die solange durchlaufen wird, wie der Spieler mutig ist. Anschließend soll dem Spieler gesagt werden, dass er vor drei Türen steht. In Python sieht dies dann so aus:

```
04  while(mutig == 1):
05      print("Vor dir siehst du 3 Türen.")
```

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

**Anmerkung:** Bei vergleichenden Aussagen (wie zum Beispiel ist mutig gleich 1), wird immer ein doppeltes Gleichheitszeichen verwendet. Damit unterscheidet man Zuweisungen von Werten (einfaches Gleichheitszeichen) von Vergleichungen (doppeltes Gleichheitszeichen).
Sehr wichtig ist, dass in Python die Zeilen innerhalb einer Schleife oder Bedingung immer eingerückt werden. Sehen kannst du dies auch in Scratch, wo die weiteren Bausteine auch etwas nach rechts rücken. Drücke in *Geany* dazu einfach die *Tabulator* Taste.

Nun soll der Spieler aufgefordert werden, sich für eine Tür zu entscheiden. In Python verwenden wir hierfür:

```
06    Antwort = input("Durch welche Tür willst du gehen?")
```

**Anmerkung:** Eingabeaufforderungen werden in Python mit dem Befehl *input* eingeleitet. Vor dem Befehl muss die erwartete Eingabe einer Variable zugewiesen werden. Dabei muss diese Variable nicht zwingend Antwort heißen.

Nun erfolgt zum ersten Mal etwas typisches für Python. Bei Scratch musstest du dir keine Gedanken um Datentypen von Variablen machen. Du konntest ganz einfach die Zahl 1 mit der Eingabe 1 bzw. dem Text bestehend aus der Ziffer 1 vergleichen. In Python ist dies nicht so einfach möglich. Eingaben werden immer als Strings bzw. Texte behandelt. Das bedeutet, dass bei einem Vergleich die Eingabe nicht direkt mit der Zahl 1 verglichen werden kann. Um dies doch zu tun, müssen wir die Eingabe in einen Integer bzw. ganze Zahl umwandeln.

Der folgende Quellcode zeigt dir, wie ein *falls-dann*-Baustein in Python aussehen muss:

```
07    if((int(Antwort) < 1) or (int(Antwort) > 3)):
08        print("Diese Tür gibt es nicht ...")
09    else:
10        print("Du gehst durch Tür " + Antwort + ", als plötzlich ...")
```

**Anmerkung:** Mit dem *int* Befehl kann man zum Beispiel aus einem String einen Integer machen.
Um im *print* Befehl mehrere Strings (Texte, Variablen) zu verketten, kann man einfach das + verwenden.
Bei Python ist immer auf den Datentyp von Variablen zu achten. Wäre zum Beispiel die Variable *Antwort* kein String, so wäre das Verketten nicht so direkt möglich. (Dazu später mehr.)

Weiter geht es mit der Frage, ob hinter der ausgewählten Tür sich nun der Geist befindet. Es gibt also wieder einen *falls-dann*-Baustein mit entsprechenden Ausgaben und Variablenzuweisungen, der in Python dann so aussieht:

```
11        if((int(Antwort) == randint(1, 3))):
12            print("... ein GEIST vor dir steht!")
13            print("Du bekommst Angst und rennst schnell weg.")
14            mutig = 0
15        else:
16            print("... nichts passiert.")
17            punkte = punkte + 1
```

**Anmerkung:** In Python werden *falls-dann*-Bausteine zu *if-else* Anweisungen.
Der Befehl *randint* ermittelt eine ganzstellige Zufallszahl. Bei dem obigen Beispiel wird eine Zufallszahl zwischen 1 und 3 ermittelt.
Mit dem *int* Befehl kann man zum Beispiel aus einem String einen Integer machen.
Um im *print* Befehl mehrere Strings (Texte, Variablen) zu verketten, kann man einfach

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

das + verwenden.
Bei Python ist immer auf den Datentyp von Variablen zu achten. Wäre zum Beispiel die Variable *Antwort* kein String, so wäre das Verketten nicht so direkt möglich. (Dazu später mehr.)

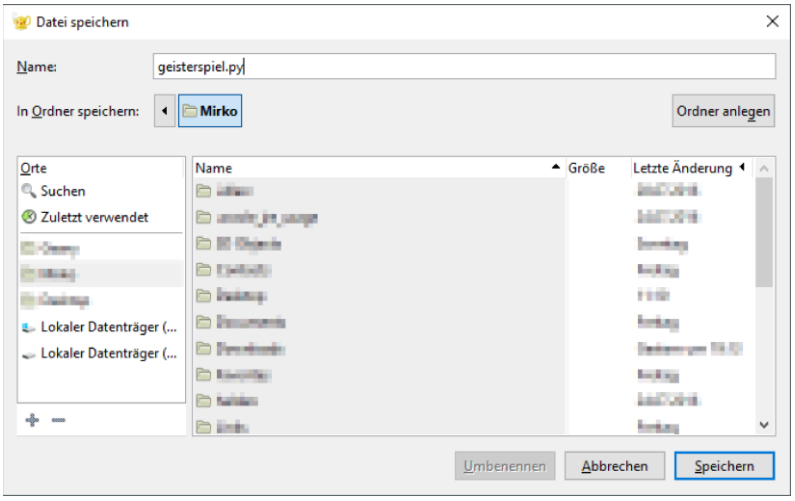
Am Ende des Programms, sobald der Spieler nicht mehr mutig ist, sollen noch die Punkte angezeigt werden:

```
18  print("Du hast " + str(punkte) + " Punkte erreicht.")
```

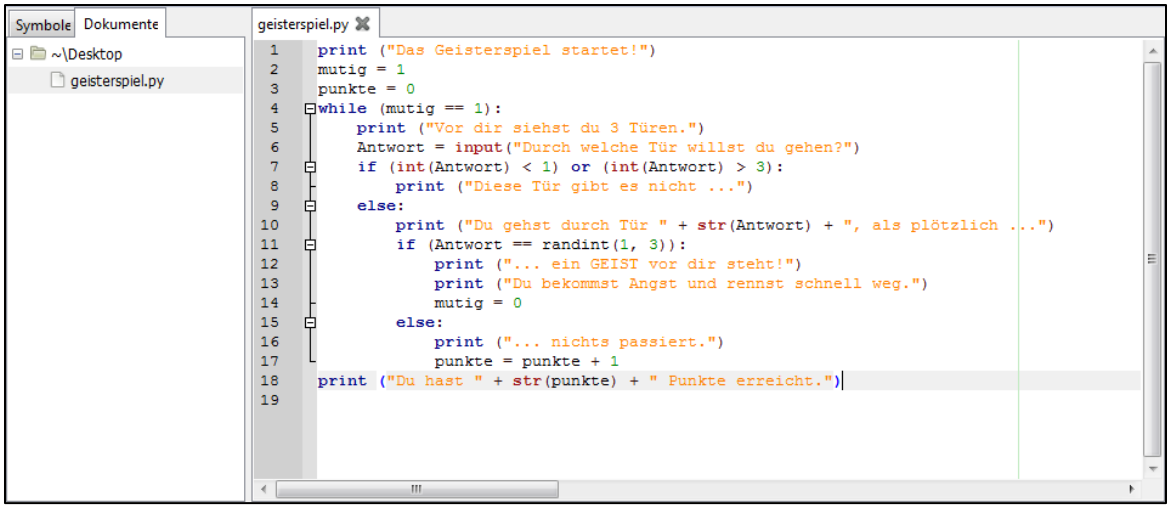
**Anmerkung:** Auch hier ist auf den Datentyp der Variable punkte zu achten. Da die Variable ein Integer ist (durch die Zuweisung von 0 am Anfang), müssen wir diese für die Ausgabe kurzzeitig zu einem String machen. Dies geht mittels *str* Befehl.

## Schritt 3: Erste Test mit vorprogrammiertem Crash

Speichere zunächst dein Programm ab. Klicke dazu auf Speichern, wähle einen passenden Speicherort und achte beim Dateinamen auch auf das richtige Dateiformat für Python. Als Dateinamen empfehlen wir geisterspiel.py.




Sobald du das Programm abgespeichert hast, sollte dir auffallen, dass deine Befehle im Programm farbig sind. *Geany* markiert dir so bestimmte Befehle aus Python, damit du dich besser im Quellcode orientieren kannst. Wenn du fertig bist, sollte das ganze so aussehen:




Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

Nun wollen wir das ganze aber auch einmal testen. Du musst auf Ausführen klicken, um dein Programm zu starten.




Beim Testen solltest du merken, dass dein Programm direkt nach der Eingabe nicht mehr funktionieren sollte. Die Fehlermeldung sollte wie folgt lauten:

```
Traceback (most recent call last):
  File "geisterspiel.py", line 11, in <module>
    if (Antwort == randint(1, 3)):
NameError: name 'randint' is not defined
```

Ganz unten steht dabei die eigentliche Fehlermeldung. Der Befehl randint ist nicht definiert. Das heißt, dass Python damit gar nichts anfangen kann. (Die Lösung dafür siehst du gleich.) Die Zeilen darüber helfen dir, den Ursprung des Fehlers herauszufinden. In der zweiten Zeile steht, wo der Fehler aufgetreten ist: In der Datei geisterspiel.py, in der Zeile 11. Darunter wird dir in der dritten Zeile der Quellcode der fehlerhaften Zeile angezeigt.

Fehler treten beim Programmieren immer wieder auf und sind generell auch nichts schlimmes. Wichtig ist nur, sich die Fehlermeldung genau anzuschauen und den Fehler so ausfindig zu machen. Sollte dir das mal nicht direkt gelingen, kann es auch helfen, eine Suchmaschiene zu fragen.

## Schritt 4: Fehler beheben

Wie bereits erwähnt, kennt Python den Befehl *randint* noch nicht. Das lässt sich schnell ändern und ist auch nicht ungewöhnlich. Python beinhaltet nicht immer alle Befehle, da sonst die Programme zu groß wären, wenn für jede Eventualität jeder Befehl eingebunden wäre. Daher, müssen manche Befehle erst noch importiert bzw. bekannt gemacht werden.

In unserem Fall heißt das, dass wir folgenden Quellcode an den Anfang unseres Programms hinzufügen müssen:

```
from random import randint
```

Das ganze Programm sollte nun wie folgt aussehen:

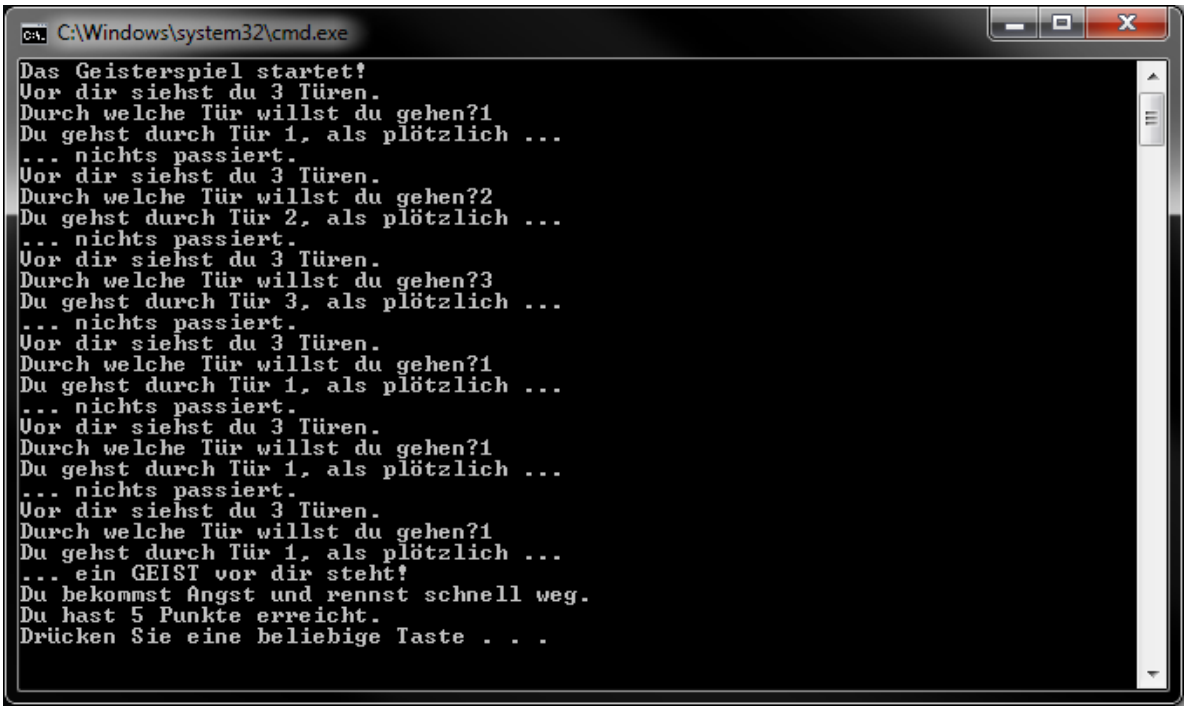
```
01  from random import randint
02
03  print("Das Geisterspiel startet!")
04  mutig = 1
05  punkte = 0
06  while(mutig == 1):
07      print("Vor dir siehst du 3 Türen.")
08      Antwort = input("Durch welche Tür willst du gehen?")
09      if((int(Antwort) < 1) or (int(Antwort) > 3)):
10          print("Diese Tür gibt es nicht ...")
11      else:
12          print("Du gehst durch Tür " + Antwort + ", als plötzlich ...")
```

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

```
13          if((int(Antwort) == randint(1, 3))):
14              print("... ein GEIST vor dir steht!")
15              print("Du bekommst Angst und rennst schnell weg.")
16              mutig = 0
17          else:
18              print("... nichts passiert.")
19      punkte = punkte + 1
20  print("Du hast " + str(punkte) + " Punkte erreicht.")
```

## Schritt 5: Spiel testen

Du kannst nun dein Spiel testen. Das Ergegniss könnte ungefähr so aussehen:




## Aufgabe 3

Mache nun selbst kleinere Änderungen am Spiel. Füge zum Beispiel weitere Türen hinzu oder sorge dafür, dass der Text etwas leserlicher ist.

**Anmerkung:** Möchtest du im *print* Befehl einen Zeilenumbruch machen, dann musst du an der entsprechenden Stelle ein \n einfügen, wie im folgenden Beispiel gezeigt:

```
print("Zeile 1\nZeile2")
```

## Aufgabe 4

In der folgenden Tabelle siehst du links, die Bausteine aus Scratch. Schreibe in die rechten Spalten den entsprechenden Befehl in Python.

| Scratch | Python |
|---|---|
|  | |



Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | |
|---|---|
|  | |
|  | |
|  | |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Objektorientierte Programmierung

Die objektorientierte Programmierung ist ein wichtiges Konzept in der modernen Programmierung. Doch in wie weit unterscheidet sie sich von der Programmierung in Scratch? Die Antwort ist, dass sie sich nicht wirklich davon unterscheidet. Trotzdem ist es wichtig zu wissen, was Objekte und Klassen sind und was diese mit der objektorientierten Programmierung zu tun haben.

## Ein Beispiel aus dem Alltag

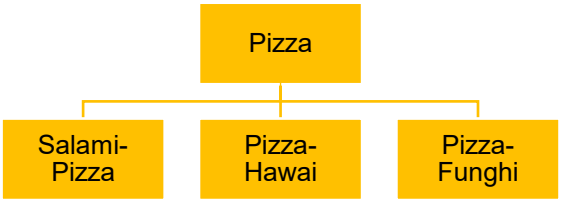
Schauen wir uns doch mal die folgenden drei Pizzen an:





Sie haben alle einen unterschiedlichen Belag, aber bei allen handelt es sich um eine Pizza. Für eine Pizza lässt sich festlegen, dass jede Pizza einen Grundteig (z. B. normaler Hefeteig oder Vollkornteig) und verschiedene Zutaten für den Belag besitzt. Zusätzlich gibt es noch bestimmte Tätigkeiten, die mit einer Pizza möglich sind, wie zum Beispiel bestellen, backen, schneiden oder essen. Doch was hat das mit Objekten oder Klassen zu tun?

Diese Frage ist ganz einfach zu beantworten. Die verschiedenen Varianten von Pizzen sind unsere Objekte und diese lassen sich alle als Pizza klassifizieren. Oder anders gesagt: Pizza bildet eine Klasse und die tatsächliche Umsetzung einer Pizza, zum Beispiel die Salami-Pizza, ist ein Objekt dieser Klasse.




## Bitte einmal Pizza mit Python …

Doch wie sieht das ganze in Python aus und was kann man damit machen? Eine Umsetzung des Pizza-Beispieles kann wie folgt aussehen:

```
01  class Pizza:
02      def __init__(self):
03          self.teig = 'Hefeteig'
04          self.zutaten = ['Tomatensosse', 'Kaese']
05          self.rand = 'ohne'
06
07      def bestellen(self):
08          pass
```

Klassen werden in Python mit dem Befehl *class* eingeleitet. Innerhalb von Klassen wird der Quellcode so eingerückt, wie es bei Schleifen oder Bedingungen bekannt sein sollte. Die Methode *__init__* wird ausgeführt, sobald ein Objekt dieser Klasse initialisiert (erzeugt) wird. Im obigen Beispiel wird der Teig standardmäßig auf Hefeteig, der Rand auf ohne und die Zutaten

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

auf Tomatensoße und Käse gesetzt. Dies soll die Vorlage für jede Pizza sein (dies kann sich aber bei einzelnen Objekten ändern). Wichtig ist die Verwendung von *self*. Bei der Definition der Methode *__init__* (z.B. Zeile 2) wird es innerhalb der Klammern eingebunden. Dies ist wichtig, da durch *self* gekennzeichnet wird, dass die daran anschließende Variable (siehe z. B. Zeile 3) zu den einzelnen Objekten gehört. Tut man dies nicht, so könnte es sich um eine beliebige globale Variable handeln. Der Befehl *pass* wird benutzt, wenn eine Methode (innerhalb von Klassen nennt man Funktionen Methoden) nicht detaiiert beschreiben möchte.

Wie bereits erwähnt ist die objektorientierte Programmierung nicht viel anders als das bekannte Programmieren in Scratch. In Scratch stellten die einzelnen Sprites/Figuren die Objekte und Klassen dar, wobei Objekt und Klasse hier sehr eng verbunden waren. Das eigentliche Scratch-Programm hatte mehrere Sprites bzw. Klassen, welche miteinander interagiert haben.

## Wo liegt der nutzen?

An dieser Stelle ist die Frage nach dem Nutzen sehr berechtigt. Ein sehr großer Vorteil, den du auch in den kommenden Unterrichtsstunden noch sehen wirst, ist, dass Variablen und Methoden sinnvoll zusammengepackt werden. Schau dir dieses Beispiel einmal an:

```
01  class Pizza:
02      def __init__(self):
03          self.teig = 'Hefeteig'
04          self.zutaten = ['Tomatensosse', 'Kaese']
05          self.rand = 'ohne'
06
07      def bestellen(self):
08          print('Pizza mit ' + self.teig + ' und ' + self.rand + \
09                '-Rand wird bestellt.')
10          print('Zutaten-Liste:')
11          print(self.zutaten)
12
13      def vergleiche_mit(self, andere_pizza):
14          pass
15
16  pizza1 = Pizza()
17  pizza1.zutaten.append('Salami')
18  pizza1.zutaten.append('Schinken')
19  pizza1.rand = 'Käse'
20
21  pizza2 = Pizza()
22  pizza2.teig = 'Vollkornteig'
23  pizza2.zutaten.remove('Tomatensoße')
24  pizza2.zutaten.append('Currysoße')
25  pizza2.zutaten.append('Ananas')
26  pizza2.zutaten.append('Hähnchenfleisch')
27  pizza2.rand = 'mit'
28
29  pizza1.bestellen()
30  pizza2.bestellen()
31  pizza1.vergleiche_mit(pizza2)
```

Wichtig ist, auf das self zu achten.

Die Methode definieren wir nich vollständig und nutzen daher das *pass*.

Objekte werden initialisiert.

Die Methode ist zwar nicht denau definiert, aber man erkennt, dass auch Objekte übergeben werden können.
Dies hat den Vorteil, dass nicht alle Variablen des Objekt an dessen stelle übergeben werden müssen.

Im obigen Beispiel sieht man, wie zwei Objekte der Klasse Pizza initialisiert (erzeugt) und angepasst werden. Dabei sind die Eigenschaften der beiden Pizzen voneinander getrennt und beeinflussen sich gegenseitig nicht. Gerade in größeren Programmen ist es wesentlich einfacher mit Objekten zu arbeiten, als ohne. Neben diesem Vorteil gibt es aber noch weitere Vorteile, auf die wir jedoch erst später im Unterricht eingehen werden. Übrigens kann man mit \ einen Zeilenumbruch im Quellcode anzeigen (siehe Zeile 8).

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Grafische Oberflächen

Mit Python lassen sich auch recht simpel grafische Oberflächen zu den Programmen erstellen. Dabei kann man auf die Funktionen der Bibliothek (so nennt man eine größere Menge von zusammengehörenden Klassen) *Tkinter* zurückgreifen. Das folgende Beispiel zeigt dir, wie du ein einfaches Fenster mit einem *Hallo Welt* - Text unter Verwendung von *Tkinter* erstellst und anzeigen lässt:

```
from Tkinter import *
fenster = Tk()
text = Label(fenster, text="Hella Welt")
text.pack()
fenster.mainloop()
```

In den folgenden Schritten wirst du dein erstes kleines Programm mit grafischer Oberfläche (GUI) programmieren und sollst die Grundlagen der GUI-Programmierung kennenlernen.

## Schritt 1: Grundgerüst

Zunächst einmal musst du die Entwicklungsumgebung Geany starten und ein neues Python Programm öffnen. Anschließend kannst du den folgenden Quellcode abschreiben:

```
# Laden der Bibliothek
from Tkinter import *

# Fenster initialisieren und Titel setzen
fenster = Tk()
fenster.title("Additionsrechner")

# Fenster ausfuehren
fenster.mainloop()
```

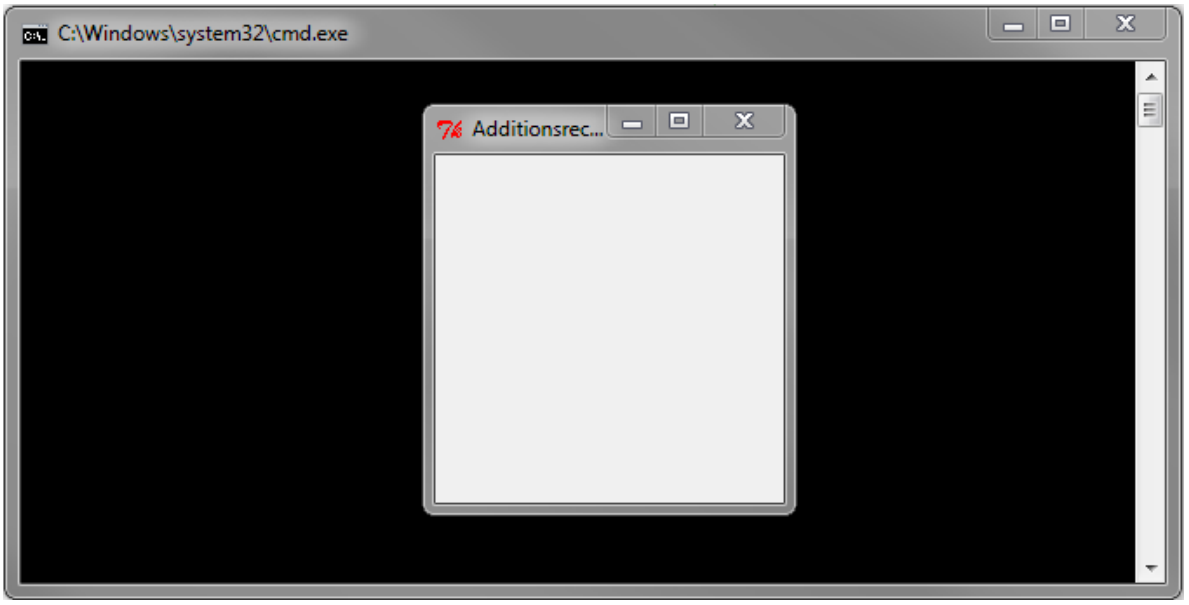
Dies soll zunächst einmal das Grundgerüst für den Additionsrechner sein. Damit grafische Oberflächen erzeugt werden können, muss die Bibliothek *Tkinter* importiert werden (siehe Zeile 2).

**Hinweis:** Um uns etwas Aufwand zu sparen, importieren wir direkt alles aus dieser Bibliothek durch *from … import **.* Ansonsten müssten wir vor allen Klassen aus dieser Bibliothek immer *Tkinter.* schreiben.

Mit *Tk()* wird das Fenste initialisiert und mit der Methode *title*() lässt ein ein Titel für das Fenster festlegen. Die Methode *mainloop()* sorgt dafür, dass das Fenster angezeigt wird und alle damit verbundenen Programmlogiken ausführbar sind.

Wenn du das Programm ausführst, sollest du folgendes angezeigt bekommen. (Die Console, das Fenster mit dem schwarzen Hintergrund, wird immer mitgeöffnet.)

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.




## Schritt 2: Die ersten Komponenten

Komponenten wie Labels (Textbausteine), Eingabefelder oder Buttons müssen vor dem Aufruf von *mainloop()* zum Fenster hinzugefügt werden. Zunächst wollen wir das erste Eingabefeld hinzufügen. Füge dafür folgenden Quellcode nach der Initialisierung des Fenster ein:

```
# Erstes Eingabefeld hinzufuegen
zahl1_entry = Entry(fenster, width=5, justify="center")
zahl1_entry.grid(row=0, column=0)
```

Die Klasse Entry steht für typische Eingabefelder. Bei der Initialisierung Entry(…) muss der erste Parameter immer die Komponente sein, in die das Eingabefeld eingebunden wird (in diesem Fall das Fenster). Anschließend können weitere Parameter frei übergeben werden (in diesem Fall die Breite (width) und Textausrichtung (justify)).

Zum Schluss muss noch angegeben werden, wo genau das Eingabefeld positioniert werden soll. Hierfür gibt es 3 verschiedene Methoden: *pack(), grid()* und *place()*. Mit der *pack*-Methode können Elemente grob durch Angabe einer Richtung (left, right, top, down) angeordnet werden. Mit der *grid*-Methode unterteilt man die grafische Oberfläche in eine tabellenartige Struktur. Hier lässt sich ein Element unter Angabe von Zeile (row) und Spalte (column) anordnen. Mit der *pack*-Methode können Elemente pixelgenau auf der Oberfläche angeordnet werden. Der einfachhaltshalber verwenden wir hier nur die *grid*-Methode.

Würdest du das Programm nun ausführen, solltest du folgende Oberfläche angezeigt bekommen:



## Schritt 3: Weitere Komponenten

Nun sollen weitere Komponenten hinzugefügt werden. Füge dazu den folgenden Quellcode nach der Platzierung des ersten Eingabefeldes hinzu:

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

```
# Plus-Label hinzufuegen
plus_label = Label(fenster, width=5, text="+")
plus_label.grid(row=0, column=1)

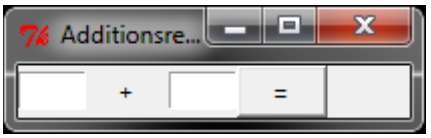
# Erstes Eingabefeld hinzufuegen
zahl2_entry = Entry(fenster, width=5, justify="center")
zahl2_entry.grid(row=0, column=2)

# Gleich-Button hinzufuegen
gleich_button = Button(fenster, width=5, text="=", command=addiere)
gleich_button.grid(row=0, column=3)

# Ergebnis-Label hinzufuegen
ergebnis_label = Label(fenster, width=5, text="")
ergebnis_label.grid(row=0, column=4)
```

Im obigen Quellcode werden nun vier weitere Komponenten hinzugefügt. Ein Label mit einem Pluszeichen als Text, ein weiteres Eingabefeld, ein Button mit einem Gleichheitszeichen als Text und ein Label ohne Text. Der Quellcode der neuen Komponenten ist ähnlich zu dem aus dem letzten Schritt. Einzige Besonderheit ist der Parameter *command* des Buttons. Dieser gibt an, welche Methode/Funktion ausgeführt wird, sobald auf diesen Button geklickt wird.

Die grafische Oberfläche würde nun wie folgt aussehen:




Ausführen lässt sich das Programm aber noch nicht, da die Methode/Funktion *addiere* noch nicht definiert wurde.

## Schritt 4: Das Addieren

Wichtig ist, dass die Funktion addiere vor dem Button definiert werden muss, da dieser auf die Funktion verweist. Es empfiehlt sich, solche Funktionen immer am Anfang, direkt nach dem Importieren der Bibliotheken, zu definieren. Füge folgenden Quellcode vor der Initialisierung des Fenster hinzu:

```
# Methode zum Addieren definieren
def addiere():
	# Hole Eingaben aus Eingabefeldern
	zahl1 = zahl1_entry.get()
	zahl2 = zahl2_entry.get()

	# Pruefe ob Eingaben auch Zahlen sind
	if (zahl1.isdigit() and zahl2.isdigit()):
		# Wandle Eingaben in Integer um
		zahl1 = int(zahl1)
		zahl2 = int(zahl2)

		# Aendere Text des Ergebnis-Labels in berechnetes Ergebnis
		ergebnis_label.config(text = str(zahl1 + zahl2))
```

An sich sollte der obrige Quellcode anhand der Kommentare selbsterklärend sein. Wichtige Stellen sind hier nur die Verwendung der *get()* Methode, um den Inhalt der Eingabefelder zu erhalten, das Prüfen auf Integer-Zahlen mittels *isdigit()* und das Ändern des *text*-Attributes des Ergebnislabels mittels *config()*.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

## Schritt 5: Ungültige Eingaben

Was nun jedoch noch fehlt ist, eine Nachricht oder ein Hinweis, sofern keine ordentlichen Eingaben getätigt wurden. Hierfür muss die *if*-Anweisung der *addiere*-Funktion mit einer *else*-Anweisung erweitert werden. Als kleine Besonderheit soll jedoch die Fehlermeldung in Form eines keinen Hinweisfenster angezeigt werden. Solche Nachrichtenfenster können mit einer zusätzlichen Klasse ganz einfach angezeigt werden.

Importiere dazu die Klasse *tkMessageBox*. Füge dazu folgenden Quellcode nach dem Import von Tkinter ein:

```
import tkMessageBox
```

Anschließend kannst du folgenden Quellcode an das Ende der addiere Funktion hinzufügen:

```
# Wenn Eingaben keine Zahl, dann Hinweis-Fenster anzeigen
else:
	tkMessageBox.showinfo("Fehler", "Fehlerhafte Eingabe!")
```

Die Methode *showinfo* der Klasse *tkMessageBox* sorgt dafür, dass ein Nachrichtenfenster mit dem Titel *Fehler* und der Nachricht *Fehlerhafte Eingabe!* angezeigt wird.

Hiermit ist dein erstes Programm mit grafischer Oberfläche fertig.

## Überblick

Der Quellcode des gesamten Programms sollte, wenn du alles richtig gemacht hast, wie folgt aussehen:

```
# Laden der Bibliothek
from Tkinter import *
import tkMessageBox

# Methode zum Addieren definieren
def addiere():
	# Hole Eingaben aus Eingabefeldern
	zahl1 = zahl1_entry.get()
	zahl2 = zahl2_entry.get()

	# Pruefe ob Eingaben auch Zahlen sind
	if (zahl1.isdigit() and zahl2.isdigit()):
		# Wandle Eingaben in Integer um
		zahl1 = int(zahl1)
		zahl2 = int(zahl2)

		# Aendere Text des Ergebnis-Labels in berechnetes Ergebnis
		ergebnis_label.config(text = str(zahl1 + zahl2))

	# Wenn Eingaben keine Zahl, dann Hinweis-Fenster anzeigen
	else:
		tkMessageBox.showinfo("Fehler", "Fehlerhafte Eingabe!")

# Fenster initialisieren und Titel setzen
fenster = Tk()
fenster.title("Additionsrechner")

# Erstes Eingabefeld hinzufuegen
zahl1_entry = Entry(fenster, width=5, justify="center")
zahl1_entry.grid(row=0, column=0)

# Plus-Label hinzufuegen
plus_label = Label(fenster, width=5, text="+")
plus_label.grid(row=0, column=1)
```

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

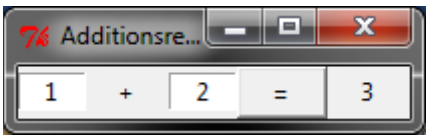
```
# Erstes Eingabefeld hinzufuegen
zahl2_entry = Entry(fenster, width=5, justify="center")
zahl2_entry.grid(row=0, column=2)

# Gleich-Button hinzufuegen
gleich_button = Button(fenster, width=5, text="=", command=addiere)
gleich_button.grid(row=0, column=3)

# Ergebnis-Label hinzufuegen
ergebnis_label = Label(fenster, width=5, text="")
ergebnis_label.grid(row=0, column=4)

# Fenster ausfuehren
fenster.mainloop()
```

Ein Berechnung auf der grafischen Oberfläche kann so aussehen:




Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Memory

Als nächstes Projekt mit Python kannst du dein eigenes Memory möglichst selbstständig programmieren. Hilfestellung bekommst du dabei in erster Linie durch die folgenden Beispiele und der Recherche in Online-Tutorials. Eine kleine Übersicht guter Tutorials findest du am Ende.

## Aufgabenstellung

Programmiere dein eigenes Memory mit grafischer Oberfläche. Das Spielfeld sollte aus 4 Paaren von Karten bestehen. Der Spieler soll auf eine Karte klicken können, woraufhin diese aufgedeckt wird. Sollte bereits eine Karte aufgedeckt worden sein, dann muss geprüft werden ob es ein Paar ist. Wenn ja, dann bleiben beide Karten aufgedeckt und der Spieler kann fortfahren, wenn nicht, dann werden die beiden Karten wieder umgedreht. Zu Beginn wird es nur einen Spieler ohne Punktestand geben.

Ein Grundgerüst des Programms inklusive Bilder für die Karten steht dir in Form eines ZIP-Archives bereit.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Hilfestellung zum Memory

## Bilder als Buttons

Um Bilder in einem Button einzubinden, muss zunächst einmal ein Objekt der Klasse PhotoImage (in der *Tkinter*-Blibliothek enthalten) erzeugt werden:

```
bild  = PhotoImage(file='Pfad zur GIF-Datei')
```

Wichtig ist, dass es sich bei Python 2.7 um eine GIF-Datei handelt. Andere Dateiformate werden erst ab Python 3.6 unterstützt.
Anschließend kann das Bild dem Button mittels image Attribut zugewiesen werden:

```
Button(..., image=bild, ...)
```

Hier ein ganzes Beispiel:

```
fenster = Tk()
bild  = PhotoImage(file='button.gif')
button = Button(fenster, image=bild)
button.pack()
fenster.mainloop()
```

## Verzögertes Ausführen einer Funktion

Angenommen du möchtest nach 5 Sekunden etwas ausführen lassen, dafür bietet das *Tk*-Objekt (also das Fenster) die *after*-Methode an:

```
fenster.after(ZEIT, FUNKTION, PARAMETER …)
```

Als erstes muss immer die Wartezeit in Millisekunden übergeben werden, anschließend der Name der Funktion die ausgeführ werden soll (also ohne Klammern und Parameter) und letztlich alle Parameter, die der Funktion übergeben werden sollen.

Möchtest du zum Beispiel deine eigene Funktion *karte_umdrehen(karte)* nach 5 Sekunden ausführen lassen, so müsste der Quellcode wie folgt lauten:

```
fenster.after(500, karte_umdrehen, karte)
```

## Button soll auf Drücken eine Funktion mit Parametern aufrufen

Mit dem command-Attribut kann bei einem Button eine Methode angegeben werden, die ausgeführt wird, sobald der Button gedrückt wird. Hierbei ist es aber nicht direkt möglich, der Methode auch Parameter zu übergeben. Dafür benötigt man den Befehl *partial* aus der Bibliothek *functools*:

```
from functools import partial
```

Möchte man nun zum Beispiel beim Drücken des Buttons die Funktion karte_aufdecken mit den Parametern 2 und 3 ausführen lassen, dann lautet der Quellcode dafür:

```
mein_button = Button(..., command=partial(karte_aufdecken, 2, 3), ...)
```

Statt festen Werten wie 2 und 3 können natürlich auch Variablen verwendet werden.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

## Array (verschachtelte Listen) initialisieren

Manchmal möchte man zunächst einmal einen Array mit einer festen Größe initialisieren, bevor man ihn befüllt. Dies ist in Python etwas umständlicher als in anderen Programmiersprachen. Der folgende Quellcode erzeugt einen leeren Array mit vorgegebener Anzahl an Spalten und Zeilen:

```
mein_array = [['' for i in range(anzahl_spalten)] \\
                  for j in range(anzahl_zeilen)]
```

Das Ergebnis bei 3 Spalten und 2 Zeilen würde dann so aussehen:

```
[['', '', ''],
 ['', '', '']]
```

Anschließend kann zum Beispiel auf das Feld 1, 0 (zweite Zeile und erste Spalte; beim Programmieren zählt man von 0 an) ein neuer Wert hinterlegt werden:

```
mein_array[1][0] = 'a1'
```

## Zufälliges Element aus einer Liste nehmen

Um aus einer Liste ein zufälliges Element zu nehmen, benötigt man die Funktion *choice* aus der Klasse *random*:

```
from random import choice
```

Anschließend wird mit *choice(meine_liste)* ein zufälliges Element aus der Liste *meine_liste* zurückgegeben. Das Element wird dabei nicht aus der Liste entfernt. Hier ein Beispiel:

```
moegliche_karten = ['a1', 'a2', 'b1', 'b2', 'c1', 'c2', 'd1', 'd2']
zufaellige_karte = choice(moegliche_karten)
```

## Weitere Tutorials

Hier eine kleine Übersicht nützlicher Tutorials zu GUI Programmierung in Python:

- http://www.python-kurs.eu/python_tkinter.php
  Zeigt und erklärt die wichtigsten Komponenten und gibt Beispiele zur Einbindung.
- http://www.tutorialspoint.com/python/python_gui_programming.htm
  Listet nicht nur wichtige Komponenten auf, sondern nennt auch deren wichtigsten Attribute.
- http://www.tkdocs.com/tutorial/index.html
  Sehr umfangreiches Tutorial. Rechts in der Navigation, sollte Python als Programmiersprache im Dropdown-Menü ausgewählt werden.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Objektorientierte Programmierung – Teil 2

Im ersten Teil zur objektorientierten Programmierung gab es noch einige Sachen, die unerwähnt blieben und trotzdem wichtig sind. So erlaubt uns diese Programmierung noch ein paar weitere Feinheiten bei der Verwendung von Klassen sowie auch Beziehungen zwischen den Klassen zu definieren.

## Kapselung von Daten

Häufig ist es sinnvoll, dass die Daten innerhalb eines Objekts (vorgegeben durch die Klasse) nicht von jeder x-beliebiger Stelle direkt verändert werden können. Angenommen wir haben ein Objekt Kunde mit dem Attribut Geburtsdatum. So bietet es sich an, dass die Änderung dieses Attributes nur durch eine intere Methode der Klasse durchgeführt wird. Dies hätte dann den Vorteil, dass zum Beispiel immer geprüft werden kann, ob es sich um ein gültiges Datum handelt. Dies wäre nicht möglich, wenn der Wert des Attributes durch alle möglichen Eingabequellen direkt verändert werden kann.

Mehr Sicherheit bietet daher die Möglichkeit, dass Attribute als *public* (öffentlich), *protected* (gesichert) oder *private* (privat) deklariert werden können:

- **public**
  Diese Variablen besitzen keine Unterstriche vor dem Namen und können sowohl innerhalb des Objekt als auch von außen gesehen und verändert werden.
- **protected**
  Variablen, die einen einzigen Unterstrich vor ihrem Namen haben, sind protected. Das bedeutet, dass man diese zwar auch von innen und außen heraus sehen und bearbeiten kann, aber es laut Entwickler nicht sollte. Der Entwickler möchte damit zeigen, dass es sich um nicht veränderbare Werte (Konstanten) handelt.
- **private**
  Diese Variablen besitzen genau zwei Unterstriche vor ihrem Namen. Sie können nur innerhalb des Objekts verwendet werden und sind nicht einmal von außen zu sehen.

Hier ein kleines Beispiel:

```
class Konto:
    def __init__(self, inhaber, kontostand, kontonummer):
        self.__inhaber = inhaber
        self.__kontostand = kontostand
        self.__kontonummer = kontonummer

    def gib_kontostand(self):
        return self.__kontostand

    def auszahlen(self, betrag):
        if (betrag <= self.__kontostand):
            return True
            self.__kontostand -= betrag
        else:
            return False

    def einzahlen(self, betrag):
        self.__kontostand += betrag

konto = Konto('Max Mustermann', 500, 1234567890)
print('Ich habe ' + str(konto.gib_kontostand()) + ' Euro')
```

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

```
konto.__kontostand = 1000000
if konto.auszahlen(1000000):
    print('Ich bin Millionär!')
else:
    print('Ich bin immer noch kein Millionär...')
```

Ausgeführt wäre das Ergebnis dann:




## Statische Attribute

Bisher besaßen zwei Objekte der selben Klasse meist zwar das gleiche Attribut nur mit unterschiedlichen Werten. Für die Programmeirung ist es manchmal auch sinnvoll, wenn es ein Attribut gibt, dass bei allen Objekten immer den gleichen Wert besitzt. Solche Attribute nennt man *static* (statisch).

In dem folgenden Beispiel gibt es das Attribut objekt_zaehler. Dieses soll immer angeben, wie viele Objekte es von der Klasse gibt.

```
class Zaehler:
    zaehler = 0

    def __init__(self, name):
        self.name = name
        Zaehler.zaehler += 1
        print('Es gibt einen neuen Zaehler mit dem Namen ' + \
              self.name + '.')

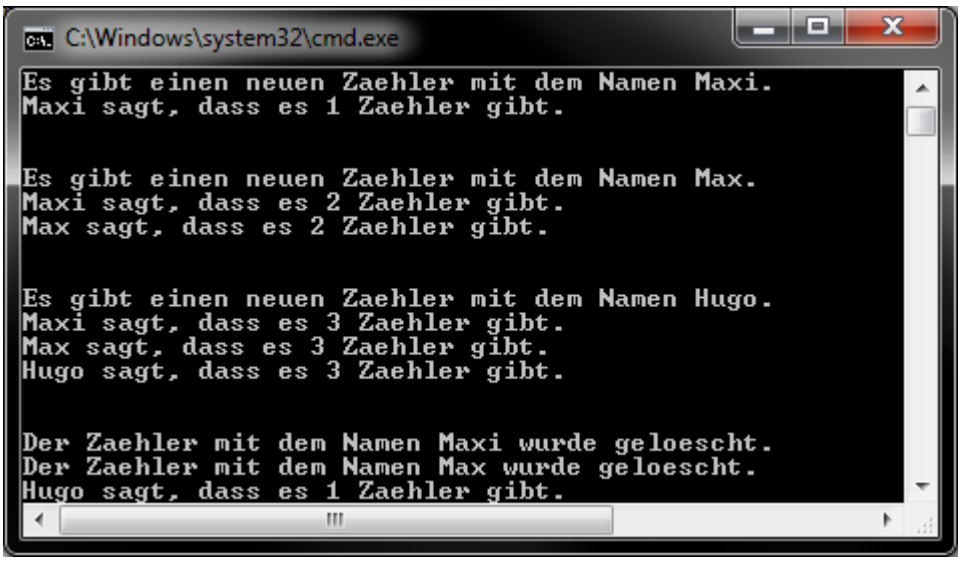
    def nenne_anzahl(self):
        print(self.name + ' sagt, dass es ' + str(Zaehler.zaehler) \
              + ' Zaehler gibt.')

    def __del__(self):
        print('Der Zaehler mit dem Namen ' + self.name + \
              ' wurde geloescht.')
        Zaehler.zaehler -= 1

mein_zahler1 = Zaehler('Maxi')
mein_zahler1.nenne_anzahl()
print('\n')
mein_zahler2 = Zaehler('Max')
mein_zahler1.nenne_anzahl()
mein_zahler2.nenne_anzahl()
print('\n')
mein_zahler3 = Zaehler('Hugo')
mein_zahler1.nenne_anzahl()
mein_zahler2.nenne_anzahl()
mein_zahler3.nenne_anzahl()
print('\n')
del(mein_zahler1)
del(mein_zahler2)
mein_zahler3.nenne_anzahl()
```

Ausgeführt wäre das Ergebnis dann:

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.




## Vererbung

Betrachten wir nun erst einmal wieder das Beispiel mit den Pizzen. Es gab eine Klasse Pizza mit Standardteig und -zutaten. Nun gibt es bei vielen Lieferdiensten unterschiedliche Größen von Pizzen. Da wären zum Beispiel die kleinen Pizzen, die Normalgroßen oder Familienpizzen. Bei allen handelt es sich um Pizza und auf alle können die gleichen Zutaten enthalten, aber trotzdem gibt es Unterschiede wie zum Beispiel den Preis. Angenommen jede Zutat kostet bei einer kleinen Pizza 50 Cent, bei einer Normalen 1 Euro und bei einer Familienpizza 1,50 Euro. Die Berechnung des Preises lässt sich verschieden lösen:

1. Wir erweitern die Klasse *Pizza* um ein Attribut *groesse*. Anschließend kann in der Methode *preis_berechnen()* geprüft werden, welche Größe die Pizza hat und der Preis entsprechend ermittelt werden.
2. Wir nutzen Vererbung. Hierbei werden von der Klasse *Pizza* die Unterklassen *KleinePizza*, *NormalePizza* und *FamilienPizza* erzeugt. Dies hätte den Vorteil, dass kein weiteres Attribut notwendig ist und man kann einzelne Methoden unterschiedlich definieren und programmieren.

Der folgende Quellcode zeigt die Lösung 1:

```
class Pizza:
    def __init__(self):
        self.teig = 'Hefeteig'
        self.zutaten = ['Tomatensosse', 'Kaese']
        self.rand = 'ohne'
        self.groesse = 'normal'

    def bestellen(self):
        print('Pizza mit ' + self.teig + ', ' + self.rand + \
              '-Rand wird bestellt.')
        print('Zutaten-Liste:')
        print(self.zutaten)
        print('Preis in Euro:')
        print(str(self.berechne_preis()))

    def berechne_preis(self):
        if (self.groesse == 'klein'):
            return len(self.zutaten) * 0.5
        elif (self.groesse == 'normal'):
            return len(self.zutaten) * 1
```

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

```
        elif (self.groesse == 'familie'):
            return len(self.zutaten) * 1.5

meine_pizza = Pizza()
meine_pizza.zutaten.append('Salami')
meine_pizza.zutaten.append('Schinken')
meine_pizza.bestellen()
```

Der folgende Quellcode zeigt die Lösung 2:

```
class Pizza:
    def __init__(self):
        self.teig = 'Hefeteig'
        self.zutaten = ['Tomatensosse', 'Kaese']
        self.rand = 'ohne'

    def bestellen(self):
        print('Pizza mit ' + self.teig + ', ' + self.rand + \
              '-Rand wird bestellt.')
        print('Zutaten-Liste:')
        print(self.zutaten)
        print('Preis in Euro:')
        print(str(self.berechne_preis()))

    def berechne_preis(self):
        pass

class KleinePizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 0.5

class NormalePizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 1

class FamilienPizza(Pizza):
    def berechne_preis(self):
        return len(self.zutaten) * 1.5

meine_pizza = NormalePizza()
meine_pizza.zutaten.append('Salami')
meine_pizza.zutaten.append('Schinken')
meine_pizza.bestellen()
```

KleinePizza ist eine Unterklasse von Pizza.

Ähnlich zu Methoden wird die Mutterklasse bei der Definierung der Unterklassen übergeben. Eine Klasse kann auch von mehreren Klassen erben.

Methoden können in Unterklassen anders definiert bzw. programmiert werden. Alles andere bleibt wie in der Mutterklasse.

Generell lässt sich hier nun nicht sagen, welche Lösung besser ist. Wichtig für das Verständnis von Vererbung ist noch, dass die Unterklassen alle Attribute und Methoden der Mutterklasse besitzen. Der Vorteil ist aber, dass diese Unterklassen unterschiedlich voneinander erweitert oder verändert werden können.

## Entwurfsmuster

Ein weiterer fundamentaler Vorteil, den die objektorientierte Programmierung mit sich bringt, ist die Verwendung von Entwurfsmustern. Hierbei handelt es sich um Lösungsansätze für wiederkehrende Entwurfsprobleme bei der Programmierung (besonders bei großen und komplexen Projekten). Es gibt sehr viele verschiedene solcher Entwurfsmuster für die unterschiedlichsten Probleme, die zum Teil auch untereinander kombiniert werden können. Eines, dass hier erwähnt werden soll, ist das *Model-View-Controller* (MVC) Entwurfsmuster.

Beim MVC werden bestimmte Bereiche eines größeren Programms voneinander getrennt, damit die Entwicklungs in größeren Teams erleichtert und mehr Übersicht über den zu entstehenden Quellcode ermöglicht wird. Programme werden beim MVC in das Model, das

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

View und dem Controller unterteilt ist. Dabei ist nicht ausgeschlossen, dass ein Programm nicht über mehrere Models, Views und/oder Controller verfügen kann.

- **Controller**
  Der Controller beinhaltet die Hauptlogik des Programms. Er steuert das ganze Programm und gibt Anweisungen an das Model und den View.
- **Model**
  Das Model dient als Datenquelle für das Programm. Das bedeutet, dass alle notwendigen Informationen in dieser Klasse enthalten sind und der Controller sich diese vom Model holt.
- **View**
  Der View ist für die visuellen Rückgaben an den Benutzer zuständig. Dies kann zum Beispiel das Zusammenbauen und Anzeigen einer grafischen Oberfläche sein. Änderungen an dieser grafischen Oberfläche werden dann meist vom Controller angestoßen und durch den View ausgeführt.

Miteinander verbunden bieten diese Klassen also eine Trennung von Daten, Programmlogik und Ausgabe. Wie bereits gesagt, kann dies den Vorteil haben, dass in einem größeren Projekt sich einzelne Entwickler zum Beispiel nur auf die Bereitstellung, Validierung und Verarbeitung von Daten befassen (also nur am Model arbeiten). Anders könnte es aber auch sein, dass sich einzelne Entwickler nur mit der Gestaltung der grafischen Oberfläche befassen (also nur am View arbeiten). Im letzten Fall könnte es dann auch einzelne Entwickler geben, die sich nur mit der Programmlogik befassen (also nur am Controller arbeiten). Wichtig bei größeren Projekten ist hier, dass alle Entwickler genau miteinander kommunizieren und das Programm auch in Gänze planen. Die letztliche Umsetzung kann dann in kleinere Gruppen unterteilt werden.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Handelssimulator

Zum Abschluss folgt nun ein etwas größeres Projekt. Bei dem (historischem) Handelssimulator handel es sich um ein recht simples Simulationsspiel. Der Spieler schlüpft in die Rolle eines Händlers und kann zwischen verschiedenen Städten hin und her reisen. Innerhalb der Städte kann der Spieler dann verschiedene Waren ein- und verkaufen.




## Aufgabe 1

Als Grundlage für dieses Programm erhälst du eine Vorlage mit MVC Entwurfsmuster. Schau dir daher den Quellcode genau an, bis du den Ablauf des Programms nachvollzogen hast. Die folgenden Übersichten zeigen dir dabei, welche Attribute und Methoden die Klassen besitzen.

**Hinweis:** Hinter jedem Doppelpunkt steht immer der Datentyp des Rückgabewerte oder des Attributes bzw. Parameters. Void steht dabei für eine Methode ohne Rückgabewert.

| **Handelssimulator_Controller** | |
|---|---|
| `__model : Handelssimulator_Model` | Beinhaltet ein Objekt der Klasse *Handelssimulator_Model*. |
| `__view : Handelssimulator_View` | Beinhaltet ein Objekt der Klasse *Handelssimulator_View*. |
| `__init__() : Void` | Mit dieser Methode wird der Controller initialisiert, das Programm aber noch nicht gestartet. |
| `kaufen() : Void` | Diese Methode wird ausgeführt, sobald man im View auf den Button *kaufen* klickt. |
| `reisen() : Void` | Diese Methode wird ausgeführt, sobald man im View auf den Button *weiterreisen* klickt. |
| `schiff_kaufen() : Void` | Diese Methode wird ausgeführt, sobald man im View auf den Button *Schiff kaufen* klickt. |
| `starte() : Void` | Mit dieser Methode kann das Programm gestartet werden. |
| `verkaufen() : Void` | Diese Methode wird ausgeführt, sobald man im View auf den Button *verkaufen* klickt. |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| Handelssimulator_Model | |
|---|---|
| __aktuelle_stadt : String | Beinhaltet den Namen einer Stadt und legt damit den aktuellen Standort des Spielers fest. |
| __anzahl_schiffe : Integer | Beinhaltet die Anzahl der aktuellen Schiffe im Besitz des Spielers. |
| __gold : Integer | Beinhaltet die Anzahl an Gold im Besitz des Spielers. |
| __lager : Liste | Beinhaltet eine Liste mit den einzelnen Mengen der Handelswaren im Besitz des Spielers |
| __lager_pro_schiff : Integer | Beinhaltet die Lagerkapazität eines Schiffes bzw. legt diese fest. |
| __max_preis : Integer | Beinhaltet den Maximalpreis einer Handelsware bzw. legt diese fest. |
| __min_preis : Integer | Beinhaltet den Minimalpreis einer Handelsware bzw. legt diese fest. |
| __preis_pro_schiff : Integer | Beinhaltet den Kaufpreis eines Schiffes bzw. legt diese fest. |
| __preise : Liste | Beinhaltet eine Liste mit den einzelnen Preisen der Handelswaren für alle Städte. |
| __staedte : Liste | Beinhaltet eine Liste mit den Namen der einzelnen Städte, welche vom Spieler bereist werden können. |
| __waren : Liste | Beinhaltet eine Liste mit den Namen der Handelswaren, welche vom Spieler ein- und verkauft werden können. |
| berechne_lagerplatz() : Integer | Mit dieser Methode kann der freie Lagerplatz berechnet werden. Rückgabewert ist der freie Lagerplatz als Integer. |
| berechne_neue_preise() : Void | Mit dieser Methode kann die Preisschwankung berechnet werden. |
| gib_aktuelle_stadt() : String | Gibt den Wert der Variable __ aktuelle_stadt zurück. |
| gib_anzahl_schiffe() : Integer | Gibt den Wert der Variable __ anzahl_schiffe zurück. |
| gib_gold() : Integer | Gibt den Wert der Variable __gold zurück. |
| gib_lager() : Liste | Gibt den Wert der Variable __lager zurück. |
| gib_preise() : Liste | Gibt den Wert der Variable __preise zurück. |
| gib_staedte() : Liste | Gibt den Wert der Variable __staedte zurück. |
| gib_waren() : Liste | Gibt den Wert der Variable __waren zurück. |
| setze_aktuelle_stadt(stadt : String) : Void | Setzt den Wert der Variable __aktuelle_stadt. |
| setze_gold(gold : Integer) : Void | Setzt den Wert der Variable __gold. |
| setze_lager(lager : Liste) : Void | Setzt den Wert der Variable __lager. |

| Handelssimulator_View | |
|---|---|
| __fenster : Tk | Beinhaltet ein Objekt der Klasse Tk. Ist damit ein Fenster der grafischen Oberfläche. |
| __gold_label : Label | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Verfügbares Gold:“. |
| __handelswaren_auswahl : Listbox | Beinhaltet ein Auswahlfeld und ist ein Element auf dem Fenster. Besitzt als Auswahlelemente die einzelnen Handelswaren inklusiver ihres Preises. |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | |
|---|---|
| `__handelswaren_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Handelswaren“. |
| `__kaufen_button : Button` | Beinhaltet einen Button und ist ein Element auf dem Fenster. Beim Betätigen des Buttons soll eine Handelsware gekauft werden. |
| `__kaufen_methode : Function` | Beinhaltet einen Verweis auf eine Methode der Klasse *Handelssimulator_Controller*. |
| `__lager_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Verfügbarer Lagerplatz:“. |
| `__lagerraum_auswahl : Listbox` | Beinhaltet ein Auswahlfeld und ist ein Element auf dem Fenster. Besitzt als Auswahlelemente die einzelnen Handelswaren inklusiver ihrer Anzahl im Lager. |
| `__lagerraum_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Lagerraum“. |
| `__menge : Entry` | Beinhaltet ein Eingabefeld und ist ein Element auf dem Fenster. Auf den Wert des Eingabefeldes wird beim Ein- und Verkauf zugegriffen. |
| `__ort_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Aktueller Ort:“. |
| `__platzhalter : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Das Label besitzt einen Inhalt und dient nur als Platzhalter. |
| `__platzhalter2 : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Das Label besitzt einen Inhalt und dient nur als Platzhalter. |
| `__reisen_button : Button` | Beinhaltet einen Button und ist ein Element auf dem Fenster. Beim Betätigen des Buttons soll zu einer neuen Handelsstadt gereist werden. |
| `__reisen_methode : Function` | Beinhaltet einen Verweis auf eine Methode der Klasse *Handelssimulator_Controller*. |
| `__schiff_button : Button` | Beinhaltet einen Button und ist ein Element auf dem Fenster. Beim Betätigen des Buttons soll ein weiteres Schiff gekauft werden. |
| `__schiff_kaufen_methode : Function` | Beinhaltet einen Verweis auf eine Methode der Klasse *Handelssimulator_Controller*. |
| `__schiffe_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Schiffe im Besitz:“. |
| `__staedte_auswahl : Listbox` | Beinhaltet ein Auswahlfeld und ist ein Element auf dem Fenster. Besitzt als Auswahlelemente die einzelnen Handelsstädte. |
| `__staedte_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Handelsstädte“. |
| `__status_label : Label` | Beinhaltet ein Label und ist ein Element auf dem Fenster. Besitzt den Text „Händlerstatus“. |
| `__verkaufen_button : Button` | Beinhaltet einen Button und ist ein Element auf dem Fenster. Beim Betätigen des Buttons soll eine Handelsware verkauft werden. |
| `__verkaufen_methode : Function` | Beinhaltet einen Verweis auf eine Methode der Klasse *Handelssimulator_Controller*. |
| `__init__() : Void` | In dieser Methode werden Variablen initialisiert und die GUI des Programms zusammengebaut. |
| `aktualisiere_handelswaren_auswahl(waren : Liste, preise : Liste) : Void` | Mittels dieser Methode soll die Auswahl der Handelswaren in der GUI aktualisiert werden. |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | |
|---|---|
| `aktualisiere_lagerraum_auswahl(waren : Liste, lager : Liste) : void` | Mittels dieser Methode soll die Auswahl des Lagerraumes in der GUI aktualisiert werden. |
| `aktualisiere_staedte_auswahl(staedte : Liste) : Void` | Mittels dieser Methode soll die Auswahl der Städte in der GUI aktualisiert werden. |
| `aktualisiere_status(aktuelle_stadt : String, gold : Integer, lagerplatz : Integer, anzahl_schiffe : Integer) : void` | Mittels dieser Methode soll der Status-Bereich der GUI aktualisiert werden. |
| `gib_ausgewaehlte_handelsware() : Liste` | Gibt das ausgewählte Element in __handelswaren_auswahl zurück. |
| `gib_ausgewaehlte_lagerware() : Liste` | Gibt das ausgewählte Element in __lagerraum_auswahl zurück. |
| `gib_ausgewaehlte_stadt() : Liste` | Gibt das ausgewählte Element in __staedte_auswahl zurück. |
| `gib_eingabe_menge : String` | Gibt die Eingabe in __menge zurück. |
| `kaufen() : Void` | Diese Methode wird ausgeführt, sobald man auf den Button *kaufen* klickt. |
| `reisen() : Void` | Diese Methode wird ausgeführt, sobald man auf den Button *weiterreisen* klickt. |
| `schiff_kaufen() : Void` | Diese Methode wird ausgeführt, sobald man auf den Button *Schiff kaufen* klickt. |
| `setze_kaufen_methode(methode : Function) : Void` | Setzt den Wert der Variable __kaufen_methode. |
| `setze_schiff_kaufen_methode(methode : Function) : Void` | Setzt den Wert der Variable __schiff_kaufen_methode. |
| `setze_reisen_methode(methode : Function) : Void` | Setzt den Wert der Variable __reisen_methode. |
| `setze_verkaufen_methode(methode : Function) : Void` | Setzt den Wert der Variable __verkaufen_methode. |
| `starte() : Void` | Mit dieser Methode kann die GUI gestartet werden. |
| `verkaufen() : Void` | Diese Methode wird ausgeführt, sobald man auf den Button *verkaufen* klickt. |
| `zeige_nachricht(title : String, nachricht : String) : Void` | Mit diese Methode kann eine Nachrichtenbox (für Fehler, Informationen etc.) angezeigt werden. |

## Aufgabe 2

Beschreibe den Ablauf beim …

a) Starten des Spiel bis zu Anzeige der grafischen Oberfläche.

b) Kauf einer Handelsware. (Beginn bei Drücken des Buttons.)

Erläutere dabei, auf welche Klassen und jeweilige Methoden zugegriffen wird. Nutze im Idealfall ein Sequenzdiagramm zur Beschreibung des Ablaufs.

## Aufgabe 3

In der Vorlage sind noch nicht alle Funktionen (Waren aus dem Lagerraum verkaufen; ein weiteres Schiff kaufen) implementiert worden. Füge diese selber in die Vorlage ein und teste sie anschließend. (Du musst dafür lediglich in der Controller-Klasse arbeiten.)

Starten kannst du das Programm von der Datei *Handelssimulator.py* aus.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

## Aufgabe 4

Nimm dir aus der folgenden Liste weitere Features (Schwierigkeit nimmt absteigend zu) für das Spiel heraus und implementiere diese in das Programm:

- Ein Spielende nach einer bestimtmen Anzahl an Spielrunden mit der Berechnung einer Punktzahl.
- Möglichkeit während der Reise von Piraten, Unwettern etc. Überrascht zu werden und einen Teil der Schiffsladung zu verlieren.
- Preise schwanken nicht global gleichmäßig, sondern für jede Stadt werden die Preise je Ware einzelnt bestimmt. (Jede Stadt besitzt also eine eigene Preisliste für die Waren, welche sich bei jedem Weitereisen verändert.)
- Möglichkeit des Eintritts eines besonderen Ereignisses (z.B. Naturkatastrophen, Kriege etc.), die den Preis einer Ware in allen Städten für mehrere Spielrunden stark an- oder absteigen lassen.
- Einen Computergegner, der ebenfalls verschiedene Städte bereist und Waren ein- und verkauft. Am Ende sollten dessen Punktzahlen mit der des Spielers verglichen werden.
- Gestalte eine neue grafische Oberfläche mit mehr Bildern und Fenstern.

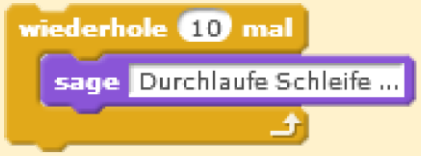
Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Befehlsübersicht

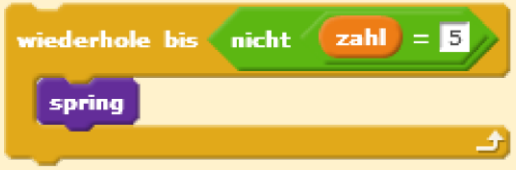
In der folgenden Tabelle sind einzelne Befehle aus Python dem Bausteinen aus Scratch gegenübergestellt.

| Beschreibung | in Scratch | in Python |
|---|---|---|
| Text auf Bildschirm ausgeben |   | `print ("Hallo Welt!")` |
| Einer Variable eine Zahl zuordnen |   | `zahl = 42` |
| Einer Variable einen String zuordnen |   | `wort = "Fisch"` |
| Erhöhe den Wert der Variable *zahl* um 1 |   | `zahl = zahl + 1`<br>oder alternativ<br>`zahl += 1` |
| Addieren |   | `zahl + 1` |
| Subtrahieren |   | `zahl - 2` |
| Multiplizieren |   | `zahl * 3` |
| Dividieren |   | `zahl / 4` |
| Setzt die Variable *zahl* auf einen zufälligen Wert zwischen 1 und 10 |   | `zahl = randint(1, 10)`<br>**Hinweis:** Der Befehl randint() muss importiert werden:<br>`from random import randint` |

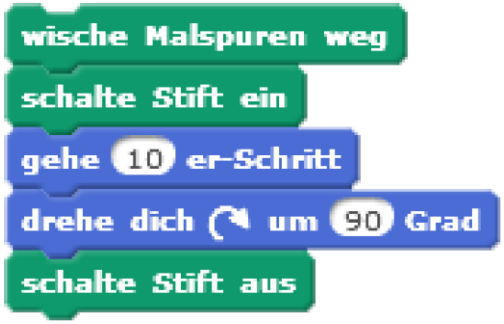
Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | | |
|---|---|---|
| Endlosschleife |  | `while True:`<br>`  print ("Durchlaufe Schleife ...")` |
| Schleife, die 10 mal durchlaufen wird |  | `while i in range(10):`<br>`  print ("Durchlaufe Schleife ...")` |
| Bedingung:<br>Wenn *wort* gleich *Tannenbaum* ist, dann *Hallo Welt!* |  | `if (wort == "Tannenbaum"):`<br>`  print ("Hallo Welt!")` |
| *wort* gleich *Tannenbaum*? |  | `wort == "Tannenbaum"` |
| Bedingung:<br>Wenn *zahl* gleich 5 ist, dann *Hallo Welt!*, sondern *Auf Wiedersehen!* |  | `if (zahl == 5):`<br>`  print ("Hallo Welt!")`<br>`else:`<br>`  print ("Auf Wiedersehen!")` |
| *zahl* gleich 5? |  | `zahl == 5` |
| *zahl* kleiner 10? |  | `zahl < 10` |
| *zahl* größer 20? |  | `zahl > 20` |
| Länge eines Strings |  | `len(wort)` |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | | |
|---|---|---|
| Eingabeaufforderung und Ausgabe der Antwort |  | `antwort = input("Wie heißt du?")`<br>`print ("Hallo " + antwort)` |
| Verbinden von Strings |  | `print ("Bernd hat " + zahl + " Äpfel.")` |
| Definieren von Funktionen |  | `def spring():` |
| Befehl ausführen |  | `spring()` |
| Bedingte Schleife mit Ungleichung |  | `while (zahl != 5):`<br>`  spring()` |
| Warten |  | `sleep(1)`<br><br>**Hinweis:** Der Befehl sleep() muss importiert werden:<br>`from time import sleep` |
| Definition einer Funktion mit Parameter |  | `def gruesse(name):`<br>`  print ("Hallo" + name)` |
| Befehl mit Parameter ausführen |  | `gruesse("Hugo")` |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

| | | |
|---|---|---|
| Liste anlegen |  | `tiere = list()` |
| Füge *Tiger* zur Liste *tiere* hinzu |  | `tiere.append("Tiger")` |
| Lösche Eintrag aus Liste |  | `del tiere[1]` |
| Ausgabe der Anzahl der Listeneinträge |  | `print ("In der Liste sind " + len(tiere) + " Tiere.")` |
| Prüfe ob Element in der Liste ist |  | `if ("Katzenbaby" in tiere):`<br>`  print ("Wie süß!")` |
| Turtle-Grafik |  | `clear()`<br>`pendown()`<br>`forward(10)`<br>`right(90)`<br>`penup()`<br><br>**Hinweis:** Die obigen Befehle müssen importiert werden:<br>`from turtle import *` |

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

# Python

Python ist eine **Programmiersprache**. Es liest einen Text ein, den du geschrieben hast (auch **Code** genannt), wandelt ihn in Befehle um, die für den Computer verständlich sind und führt diese dann aus. Du wirst lernen selbst Code zu schreiben, der coole und nützliche Dinge tut. Du wirst in der Lage sein, Programme selber zu schreiben, um mit Deinem Computer Dinge zu tun und nicht mehr nur Software von anderen Programmierern verwenden müssen.

Zuerst wirst du lernen Python zu benutzen und mit ihm zu kommunizieren. Es gibt mehrere Wege das zu tun; der erste ist, mit dem Python Interpreter über die Textkonsole deines **Betriebssystems** in Kontakt zu treten.

Eine **Konsole** (auch 'Terminal' oder 'Prompt') ist ein *schriftlicher* Weg deinem Betriebssystem Befehle zu erteilen, so wie der 'Desktop' zusammen mit deiner Maus ein *grafischer* Weg zu deinem System ist.

## Wie öffne ich eine Konsole?

Die Windows-Konsole wird *Eingabeaufforderung* genannt, oder *cmd*. Die Tastenkombination Windows+R (Windows meint die Taste mit dem Windows-Logo) öffnet dieses als neues Fenster mit schwarzem Hintergrund. Gib dann *cmd* ein und klick Enter. Die erste Zeile sollte in etwa so aussehen:

```
C:\Users\myusername>
```

Du kannst den Python Interpreter (siehe unten) auch direkt ausführen, oder das IDLE-Programm das Python mitliefert. Diese findest Du im Startmenü.

## Python verwenden

*Python* verhält sich standardmäßig als etwas das sich *Interpreter* nennt. Ein *Interpreter* nimmt Befehle entgegen und führt sie dann aus – super praktisch um Dinge auszuprobieren!

Tippe in deine Konsole *python* ein und drücke Enter um den Python-Interpreter zu starten. Sobald Python startet, zeigt es Dir einige Kontextinformationen ähnlich zu diesen:

```
Python 3.3.2 (default, January 21 2014, 15:40:45)
[GCC 4.8.0 20130502 (prerelease)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Tipp: Der Prompt >>> in der letzten Zeile signalisiert, dass du dich nun in der interaktiven *Python Konsole* befindest, auch *Python Shell* genannt. **Diese Konsole ist etwas anderes als das normale Kommandozeilenterminal!**

Theoretisch ist es also möglich in dieser *Python Shell* zu programmieren, jedoch dient es in den meisten Fällen nur für sehr kurze Anweisungen oder Tests. Für die Programmierung von richtigen Programmen verwendet man auch hier eine Entwicklungsumgebung. Als Entwicklungsumgebung reicht für Python theoretisch schon der ganz normale *Editor* unter Windows (nicht zu verwechseln mit Wordpad oder Word). Damit die Handhabung für dich aber

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

leichter ist, verwenden wir in dieser Einheit die Entwicklungsumgebung Geany, mit welcher sich unter anderem auch Python Programme schreiben, testen und ausführen lassen.

## Python Installation

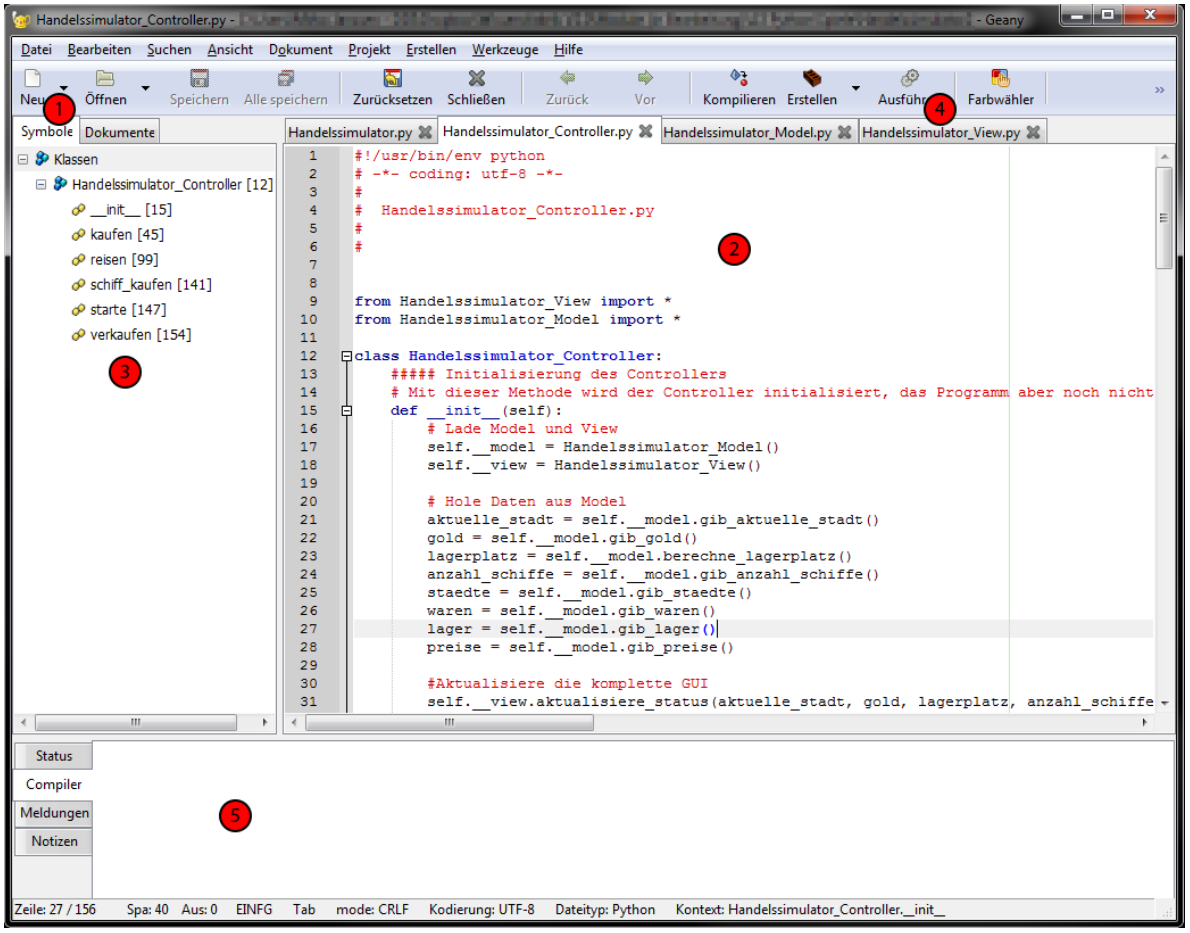
Wenn Python noch nicht auf deinem Computer installiert ist, findest du die aktuellen offiziellen Installationspakete hier:

http://python.org/download/

Wir verwenden Python 2.7 in dieser Unterrichtseinheit, prinzipiell sind beide Versionen nutzbar. Python 3 ist die eigentliche neueste Version, Python 2 wird jedoch parallel noch weiter mit entwickelt. Unter https://wiki.python.org/moin/Python2orPython3 findest du die Hauptunterschiede und Gemeinsamkeiten beider Versionen.

**Hinweis:** Unter Windows ist es nötig, dass du Python zu deinem Systempfad hinzufügst, sodass anderen Programmen in der Lage sind Python zu finden. Eigentlich sollte das in dem Setup-Programm bereits ausgewählt und damit automatisch gemacht werden. Für den Fall, dass du beim späteren Programmieren merkst, dass dies nicht der Fall ist, kannst du in das Installationsverzeichnis (C:\Python33\) navigieren, den Tools-Ordner und anschließend den Scripts-Ordner öffnen, durch ausführen der Datei win_add2path.py wird die Umgebungsvariable dann gesetzt.

## Geany




Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.

1. Hier kann eine neue Datei angelegt werden. Wichtig zu erwähnen ist hier, dass verschiedene Formate (.py für Python) als Vorlagen ausgewählt werden können.
2. In diesem Bereich befindet sich der Quellcode des Programms.
3. Hier werden die Klassen und deren Methoden aus der aktuellen Datei aufgelistet.
4. Mit diesem Button kann das Programm ausgeführt werden.
5. In diesem Bereich werden alle Benachrichtungen in Bezug auf das Kompilieren bzw. Ausführen des Programmes angezeigt. Fehlermeldungen werden grundsätzlich im Terminal des ausgeführten Programmes angezeigt.

Eine Entwicklung von OFFIS e.V. in Kooperation mit der Universität Oldenburg
im Auftrag der Wissensfabrik – Unternehmen für Deutschland e.V.