Sandra Sánchez Páez
PRO I Modulprojekt
WiSe 2021/2022
24.03.2022

Das Projekt, ein Framework für Textadventure-Spiele in Python zu entwickeln, um die gelernten OOP-Konzepte umzusetzen, erschien mir eine gute Idee zu sein. Es stimmt zwar, dass ich die Schwierigkeit der Arbeit zunächst unterschätzt habe, aber ich bin der Meinung, dass diese Art von Aufgabe sehr motivierend ist.

Die größte Herausforderung, der ich begegnet bin, war struktureller Natur. Ich denke, es ist leicht, plötzlich alles abdecken zu wollen und in kurzer Zeit auffällige Ergebnisse zu sehen, und das kann es sehr schwierig machen, kohärent und fehlerfrei zu arbeiten und zu entwickeln. Oftmals kann die Recherche und Dokumentation anderer ähnlicher Projekte dazu führen, dass wir allzu anspruchsvolle Funktionen implementieren wollen. Als ich einen angemessenen Prozentsatz der Arbeit erledigt hatte, wurde mir klar, dass ich die Beziehungen zwischen den Objekten unnötig verkompliziert hatte, und ich beschloss, noch einmal von vorne anzufangen, von weniger zu mehr. Wir müssen die Grundlagen schaffen und von dort aus weitere Funktionen implementieren.

Ich habe auch angefangen, ein Spiel zu bauen, aber mit dem Rahmen im Kopf, um es später zu extrahieren. Ich habe alle Informationen zu Räumen und Objekten aufbewahrt, die ich später wieder verwendet habe. Am Anfang wollte ich kreativer sein, aber ich habe schnell gemerkt, dass es besser ist, Platzhalter zu verwenden und später kreativ zu werden. Die Philosophie besteht darin, etwas erst funktional und dann elegant zu gestalten.

Die Struktur, der ich gefolgt bin, implementiert eine Hauptklasse "Game", die mit einem Namen, einer Beschreibung und einem anfänglichen Ort (der ein Objekt der Klasse "Room" sein wird) erstellt wird, der alles "weiß", was im Spiel passiert. Zuerst habe ich die Methoden Game.play() und Game.exit_game() implementiert, die das Starten und Beenden des Spiels ermöglichen. In please_the_cat_main.py kannst du sehen, wie die Räume erstellt werden müssen, bevor das Spiel erstellt werden kann.

Als ich das Spiel neu entwickelte, beschloss ich auch, die Klassen zusammen zu halten, um es einfacher zu machen, und erst beim Extrahieren des Frameworks habe ich sie wieder getrennt. Damals habe ich zwei Ordner angelegt, um die Dateien für das Spiel "Please the Cat" bzw. für das Framework zu organisieren. Da ich am Ende jedoch Probleme mit den Importen hatte, habe ich mich entschlossen, die parallele Struktur zu entfernen und den Hauptordner 'modulprojekt' und einen Nebenordner namens 'framework' zu belassen. Die Dateien, die sich auf das Spiel beziehen (und alle anderen), haben beschreibende und eindeutige Namen, zu denen sie gehören (Framework oder Spiel). Die letztgenannte Struktur hat mir keine Probleme bereitet.

Das Spiel benötigt nur die oben genannten Parameter, um zu starten. Die Klasse "Player" wird im Spiel generiert. Die Idee ist, ihm eine persönliche Note zu geben. Indem man den Spieler nach seinem Namen und seiner Beschreibung fragt und ihn eine Nummer wählen lässt, wird eine interaktive Komponente hinzugefügt. Diese Informationen werden gespeichert, um bei einem Sieg oder einer Niederlage personalisierte Nachrichten zu erstellen.

Die Kernlogik des Spiels ist eine while-Schleife, die dafür sorgt, dass wir so lange weiterspielen, wie der Spieler Leben hat oder das Spiel noch nicht gewonnen hat. Die Beschreibung des aktuellen Raums wird wiederholt, und die verfügbaren Aktionen ("inspect, grab, use, leave room, check inventory, exit game") werden nach jeder ausgeführten Aktion angeboten, und der Spieler wird gefragt, was er/sie tun möchte. Der Spieler kann das Spiel verlassen, sein Inventar überprüfen, Dinge inspizieren und mitnehmen und den Raum jederzeit verlassen. Es besteht die Möglichkeit, Gegenstände zu benutzen, aber nur solche, die sich im Inventar des Spielers befinden, können benutzt werden. Gegenstände der Klasse Grabbable können mitgenommen werden. Andere können inspiziert werden. Die Möglichkeit, dass der Benutzer Fehler bei der

Eingabe macht, wird immer berücksichtigt, und der Benutzer erhält die Möglichkeit, den Befehl erneut auszuführen.

Ein Hindernis, auf das ich während der Entwicklung mehrfach gestoßen bin, war die Notwendigkeit, einen str (die Benutzereingabe) mit dem Objekt zu verknüpfen, auf das er sich bezieht. Dazu wurden Hilfsmethoden wie ask_for_thing_to_grab und ask_for_thing_to_use erstellt, bei denen die Benutzereingabe als Variable gespeichert und einzeln mit den Namen der Objekte (z.B. Thing.name) im Rauminventar verglichen wird.

Ich hielt es für notwendig, die statische Methode Game.print_warning() zu dieser Klasse hinzuzufügen, vor allem um die Lesbarkeit zu verbessern und Code-Duplikationen zu vermeiden.

Hinsichtlich der Lesbarkeit muss ich hinzufügen, dass ich es im Allgemeinen nicht für nötig hielt, Kommentare hinzuzufügen, da ich denke, dass der Code klar genug ist. Alle Module, Klassen und Methoden sind mit Docstrings dokumentiert, damit man auf einen Blick versteht, was die einzelnen Dinge tun.

An dieser Stelle muss ich sagen, dass ich eine Reihe von “hot keys" in Pycharm gelernt habe, die mir die Arbeit sehr erleichtert und Flüchtigkeitsfehler vermieden haben. Die Möglichkeit, den Namen einer Variablen an allen Stellen, an denen sie verwendet wird, gleichzeitig zu ändern, ist eine davon.

Ursprünglich wollte ich eine "Animal"-Klasse einrichten, was sich aber als unnötig herausstellte. Die einzige "animierte" Klasse, die im Framework vorhanden ist, ist "Player". Der Spieler hat, wie das Spiel, die Räume und die Objekte, einen Namen und eine Beschreibung. Außerdem hat er Leben, die er durch die Benutzung bestimmter Gegenstände verlieren kann, und ein Inventar, das zu Beginn des Spiels leer ist. ‘grab’ und ‘use’ sind Aktionen, die der Spieler ausführen kann. Der Spieler wird immer gefragt, mit welchem Objekt er die jeweilige Aktion durchführen möchte, und bekommt mögliche Objekte angeboten (für "greifen" muss das Objekt zum Inventar des Raumes gehören, für "benutzen" zu seinem eigenen).

Wenn ein Gegenstand aufgenommen wird, verschwindet er aus dem Inventar des Raumes und wird dem Inventar des Spielers hinzugefügt. Die Methode "Einsatz" ist wichtiger und komplexer, da sie die Möglichkeit des Gewinns oder Verlusts des Spiels auslöst. Bei der Verwendung eines Objekts werden zwei Dinge geprüft: Zum einen, ob das Objekt die Fähigkeit hat, den Spieler zu "töten" oder ihm Leben abzuziehen. Wenn dies der Fall ist, ist das Leben ausgeglichen. Wenn diese gleich Null sind, verliert der Spieler und das Programm endet. Andererseits wird geprüft, ob es sich um einen Einweggegenstand handelt oder nicht. Ist dies der Fall, wird eine Nutzung abgezogen und der Status aktualisiert, da dieser von der Anzahl der Nutzungen abhängt.

Eine weitere Herausforderung bei der Entwicklung des Spiels bestand darin, die Verbindung zwischen den Räumen herzustellen. Zunächst habe ich versucht, dies mit Hilfe von Wörterbüchern zu tun, aber die gegenseitigen Abhängigkeiten machten es schwierig, die "Karte" einzurichten, so dass ich mich dafür entschied, die Koordinaten als Attribute zu jedem Raum hinzuzufügen (mit dem Standardwert "None") und die Verbindung zu anderen Räumen erst dann zu aktualisieren, wenn der Raum erstellt wurde.

Die Methode “leave_room” folgt dem gleichen Muster wie die anderen Funktionen. Es prüft die verfügbaren Zimmer und bietet dem Spieler deren Namen an. Es speichert die Eingaben und verknüpft sie mit dem entsprechenden Raumobjekt. Dann wird der aktuelle Standort (und sein Inventar) aktualisiert und die neue Beschreibung wird angezeigt.

Ich habe bis fast zum Schluss nicht herausgefunden, wie ich das Spiel gewinnen kann. Es war zwar klar, was man zum Gewinnen brauchte (ein bestimmtes Objekt im Inventar des Spielers), aber das musste erst einmal umgesetzt werden. Es schien nicht viel Sinn zu machen, den besonderen Gegenstand beim Betreten eines Raumes anzuzeigen, das schien zu einfach. Die Lösung bestand darin, eine Unterklasse von Room, 'SpecialRoom', zu erstellen, die das spezielle Objekt enthalten würde, das aber nicht von Anfang an sichtbar sein würde. Dieser SpecialRoom erbt die Attribute und Funktionalitäten von Room, hat aber zwei zusätzliche Attribute: eine zweite

Beschreibung und ein spezielles Objekt. Der Mechanismus ist wie folgt: Der SpecialRoom ist auch der Ort, an dem das Spiel beginnt, aber er enthält anfangs nicht das spezielle Objekt. Erst wenn der Spieler den Raum zum ersten Mal verlässt, wird die Beschreibung des Raums mit der zweiten Beschreibung aktualisiert. In diesem Moment wird der besondere Gegenstand dem Inventar des Raumes hinzugefügt. Wenn der Spieler das nächste Mal den Raum betritt, ist er in der Lage, das Spiel zu gewinnen. Auf diese Weise können wir den Spieler verwirren, der durch alle Räume gehen und mit allen Objekten interagieren kann, bevor er den Schlüssel findet, um das Spiel zu gewinnen.

Eine andere Möglichkeit, die ich in Erwägung gezogen habe, war, das besondere Objekt bei jedem Spielstart in einen zufälligen Raum zu legen, aber ich habe es weggelassen, um das Ganze nicht zu verkomplizieren. Vielleicht wäre dies eine Möglichkeit, die in Zukunft bei der Arbeit mit dem Rahmen erforscht werden könnte.

Eine Bemerkung, die ich machen muss, ist, dass ich, als ich nur noch wenig Zeit hatte, um das Projekt zu beenden, feststellte, dass ich vielleicht nicht genug von den Konzepten, die ich gelernt hatte, umgesetzt hatte. Ich denke, der Rahmen ist sinnvoll, so wie er ist, und es war nicht notwendig, mehr Funktionen zu verwenden. Um jedoch ein anderes Konzept zu nutzen, habe ich die Enum-Klasse "Deadliness" (mit zwei Elementen: deadly und not_deadly) erstellt, um den Parameter "kills" von GrabbableThing zu ersetzen, der ein Boolean war.

Die Klasse Thing ist die letzte der zu besprechenden Klassen, die ihrerseits GrabbableThing als Unterklasse hat. Thing ist die Klasse der Objekte, die nur betrachtet werden können. Diejenigen, die genommen werden können, sind Grabbable, das wiederum InfiniteUseThing (Objekte, die unbegrenzt verwendet werden können und nie aus dem Inventar verschwinden) und FiniteUseThing, das wiederum SIngleUseThing als Unterklasse hat. Gegenstände vom Typ FiniteUseThing haben die Fähigkeit, den Spieler zu "töten" oder ihm Leben zu nehmen, nachdem er sie benutzt hat.

Die Besonderheit von FiniteUseThing im Vergleich zu SingleUseThing ist, dass diese Instanzen "Zustände" haben, die mit der Anzahl ihrer verfügbaren Verwendungen verbunden sind. Wenn die maximale Anzahl von Einsätzen erreicht ist, verschwinden sie aus dem Inventar des Spielers. SingleUseThing kann nur einmal verwendet werden und trotzdem verschwinden. Sie hat auch die Fähigkeit, das Spiel für den Spieler zu gewinnen. Das heißt, der spezielle Eintrag hat den Parameter wins=True. Sobald der Spieler sie benutzt, gewinnt er und beendet das Spiel.

Abschließend möchte ich anmerken, dass ich von Anfang an die Funktion für restart_game implementieren wollte (und es in der ersten Version auch getan habe), aber je weiter ich mit der Implementierung des Frameworks fortgeschritten bin, desto komplizierter ist es geworden. Ich konnte das Spiel wieder starten, aber das Zurücksetzen der Räume und ihrer Objekte auf die Eingabewerte ist etwas komplizierter. Ich glaube nicht, dass dies eine grundlegende Funktion ist, aber es wäre schön, sie zu haben. Vielleicht ist dies auch eine weitere mögliche Verbesserung für die Zukunft.

Eine letzte Bemerkung zu den Tests, die im Framework-Ordner enthalten sind: Obwohl es spielerischer wäre, die Tests für das gesamte Spiel, das wir erstellt haben, zu entwerfen, dachte ich, es wäre interessant, ein völlig neues Beispiel zu erstellen, damit wir sehen können, dass alles so funktioniert, wie es sollte.

In einer weiteren Datei ist das UML-Diagramm enthalten, das die Beziehungen zwischen den Objekten im Framework erläutert.