# Introduction

Tucker Quest [[1]](#footnote-1) is a text-based game, created to teach design and programming concepts, as part of the AWS Cloud Institute Development Fundamentals 2 (DF2) course. The application is used on the *Fun Fridays* session, which is why a gaming related theme was selected. However, the focus is not on game development, but on various Python concepts, and design best practices.

This document is not a documentation for the game play, which in itself is an indication of the focus given to the learning aspect. The game operations are simple, and are generally intuitive when running the application.

## Disclaimer

Tucker Quest is offered for educational purposes. It is not intended to be used in production.

* This code is provided "as is" without warranty of any kind, either expressed or implied.
* The author is not responsible for any damages or other consequences that may occur as a result of using this code
* The author may not be held liable for any damages or other consequences that may occur as a result of using this code.

This disclaimer was generated by CodeWhisperer, and is included at the top of most classes in the game source code.

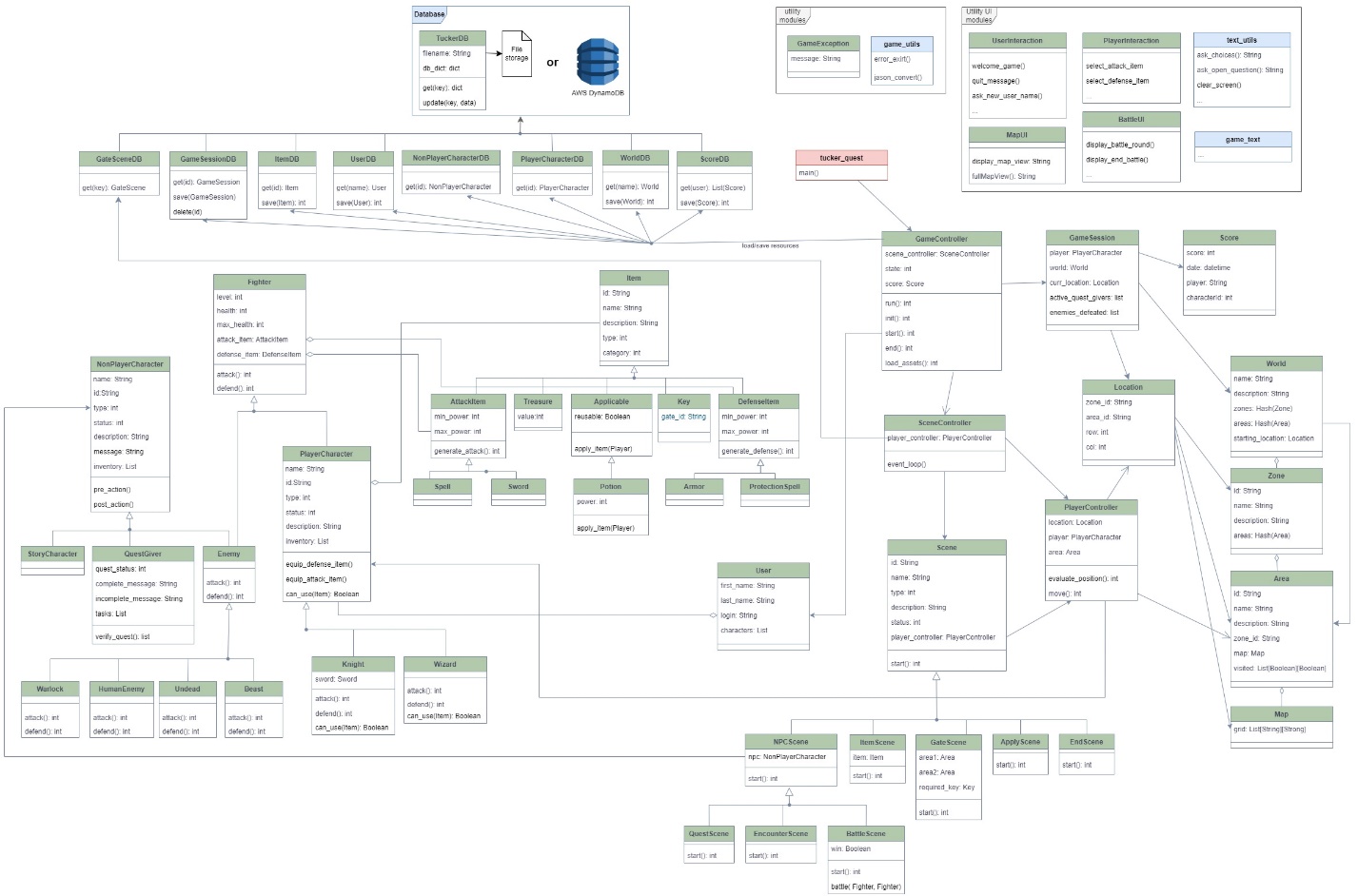
## Tenants

The Tucker Quest application is guided by the following tenants:

* The application should be at the level of an intermediate Python developer, who has gone through AWS Cloud Institute Development Fundamentals 1, or equivalent knowledge.
* The application should use only standard Python libraries, or libraries that have been discussed in the Developer Fundamentals AWS Cloud Institute curriculum.
* The application should be strictly text based, such that it can be run in a terminal window in the learner’s AWS Cloud9 practice environment.
  + This means the application will not include any graphical elements, or web pages.
  + All application user interactions will be done through text prompts, and universally supported special characters.
* The application should be easily expandable, such that learners can add their own modifications to it.

## Class Diagram

The class diagram below illustrates the overall design of the application. Clearly it is not easy to visualize at this level, but individual portions of this design will be discussed separately. Bear in mind that not every attribute and operation in the classes are reflected in the diagram, or it would become very difficult to represent it. The intent is to document those attributes and operations that fundamentally define the class.



# Development Process

The initial version of the application was designed and implemented in a short time frame. Additional content will be progressively added in the subsequent weeks of the DF2 Spring Quarter. This document will be updated as significant changes are made, and the major changes are documented below.

## Change Log

Changes in version1.4:

* Multiple small changes to support class discussion on data analytics
  + Updated Score class to store the game status. This makes no difference in game play, but it will allow for differentiation of ongoing games vs final scores when displaying scores outside of game.
  + Added stats collection for battles. This is not needed for game play, but it will be used for analytics later.
  + Improved the handling of default attack/defense items when fighters have none, so it will display better on stats
* Correct bug where adding an additional player of the same type for a user would replace the other user players of that type, instead of adding.
* Corrected a small bug in the sample *Thief* class provided in the Appendix.

Changes in version1.3:

* Implemented saving and loading of games
  + Made some changes to *GameSession* class. This is done primarily so that when we persist the game session (for saving and loading games), the current world maps are saved along with it. All changes should be backward compatible.
  + Created the *GameSessionDB* class to persist a game session
  + Add a standard option to save the game at any point
  + Updated *GameController* to support either creating a new session, or loading an existing one
  + Modified the Area JSON representation to correctly match the Area class. Specifically, the map array was previously an attribute directly under the Area.map attribute, when it should actually be inside the grid attribute in the Map. This is necessary to load and save the current area maps. This is a small change, but would cause previous maps to be incompatible.
  + Updated *PlayerCharacter* to also set inventory and health in \_\_init\_\_
* Implemented proper game exit
  + *SceneController* event loop returns value to *GameController* to indicate whether player won, lost, or quit
  + *GameController* implements appropriate methods for each of the possible outcomes, which is losing, winning, or leaving in progress.
  + User given the option of saving before exiting
  + Score is saved (see below)
* Implemented end of game
  + Created new *EndScene* class to implement steps needed for ending
  + Updated *MapUI* to create a text-based representation of the final temple
  + Updated *PlayerController* and *SceneController* to handle the approaching and initiation of the ending scene
  + In *map\_config.py,* changed the name of the NPC\_CELL\_TYPES list to PROXIMITY\_CELL\_TYPES. The list still has the same purpose (cell types that need to be checked if they are anywhere around the player), but we generalized the name, so they can be used for end scenes as well, as opposed to just NPCs.
* Implemented score keeping
  + Implemented a simple score calculation based on multiple metrics in the game session
  + Implemented the *ScoreDB* class to persist scores to the database.
* Moved code to create specific items, players and NPCs from JSON dictionaries, from the DB classes to the config modules. This was done so they can be reused in other places. The process is still the same, but just shifted some code.
* Added new zones and maps to the base package. No change in format, but just more data to make the basic world a little more complete. The last map has an end scene cell.
* Updated this document to document the new features, and to improve explanations in certain areas.

Changes in version1.2:

* Implemented quest scenes
* Implement key entry for locked gates
* Removed the *ChallengeScene* class from the game. Most of what was intended for this scene type were taken up by the *QuestScene*.
* Added new *pre\_action()* and *post\_action( )*methods to the *NonPlayerCharacter* class. These are discussed in section 6.5.1.16.5.1.
* Changed the NPC encounter scenes (*BattleScene* and *EncounterScene*) to invoke the *pre\_action()* and *post\_action()* methods at the beginning and the end of engagement.
* Added a standard option to display the player inventory
* *GameSession* now maintains a list of enemies defeated, which is updated in *BattleScene*
* *GameSession* also maintains a list of active quest givers
* Added a *pronoun* attribute to all characters, in order to display messages correctly in the UI.
* Implemented quest scene

## Content in Progress

At this time, all the base game features have been implemented. Time allowing, other things can be added as well, but only if they’ll help in the instructional aspect of the game. Which is why there are some "place holder" attributes added to classes, such as status and level. These are not used now, but they are the type of things that will be needed for a larger game.

# Naming Conventions

Tucker Quest is and object-oriented application, so with the exception of a few utility functions, the vast majority of the applications is implemented through Python classes

## Naming Conventions

The naming convention for classes, objects, variables and files is based on the [standard Python PEP 8 style guide](https://peps.python.org/pep-0008/). Please refer to the PEP 8 documentation for full details on recommended styles, but some of the key ones observed in Tucker Quest are:

* Variables are represented using snake case (Ex: curr\_player, new\_scene, etc)
* Methods and functions are represented using snake case (ex: get\_item(), player\_can\_use(), etc)
* Class names are represented in camel case (Ex: Item, BattleScene, etc)

## Class Files

The code follows the common object-oriented practice of using one file per class. This may appear to be tedious, particularly for beginners, but it is generally beneficial for reuse and maintainability of large-scale object-oriented applications. Although the class names are camel case, following standard Python guidelines, the corresponding file names are snake cased.

Example: the class *PlayerCharacter* will be in the *player\_character.py* file

### Main Program

The main program that starts execution is **tucker\_quest.py** . Specifically, the program can be started by simply typing “python tucker\_quest.py” in a terminal screen from the root tucker-quest folder.

# Code Structure

Large applications rarely keep all their files in one folder. This makes it difficult to locate files, maintain modules, and divide the work across team members. There is no rigid guideline for how to structure a Python project, so the Tucker Quest code is split into reasonable logical lines. Following is the source code structure:

**tucker-quest** 🡪 root folder, with the *tucker\_quest.py* main program and game control classes

* **config** 🡪 configuration files used throughout application
* **character** 🡪 classes defining player and non-player characters
* **item** 🡪 classes defining items
* **scene** 🡪 classes defining game scenes
* **world** 🡪 classes defining the world map
* **ui** 🡪 classes and utility functions controlling user interface
* **db** 🡪 classes isolating access to database, and TuckerDB implementation
* **data** 🡪 default location for JSON data files currently storing game data
* **logs** 🡪 default location for log files

## Accessing Classes in Different Modules

Classes in different modules will have to access each other. That is common in any programming language. There are a number of techniques and libraries available in Python to support that. In Tucker Quest, the mechanism adopted is to use the standard Python “sys” library to add the module folders to the main class. Specifically, the first few lines of the *tucker\_quest.py* main program include:

sys.path.append('db')

sys.path.append('config')

sys.path.append('item')

sys.path.append('character')

sys.path.append('ui')

sys.path.append('scene')

sys.path.append('world')

# Application Logging

Tucker Quest makes extensive use of the standard Python logging library. Proper logging is essential for maintaining and troubleshooting large applications. Most of the logging is done at the DEBGU level, so it will not be logged by default. However, it is recommended that during the development phase, the level is set to DEBUG. In order to control the logging without having to change code, Tucker Quest supports the following run time configurations:

* TUCKER\_LOGLEVEL environment variable will set the logging level (defaults to INFO)
* TUCKER\_LOGFILE environment variable will set the file name (defaults to tucker\_quest.log)
* The “-l” argument can be passed at tun time to override the log level. For example, this would set the log level to DEBUG for this execution:

Ex: python tucker\_quest.py -l 10

# Detailed Design

This section will describe the design of individual sections of the application.

## Game Framework

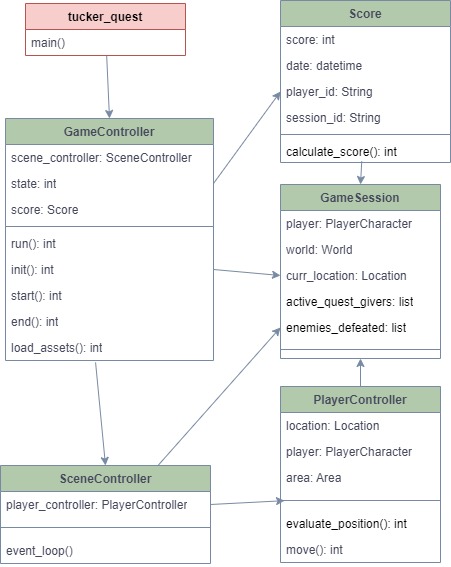
In this detailed design section, we will be introducing a number of different classes, grouped into different hierarchies, that perform key functions in the Tucker Quest application. We'll see classes that define items in the game, classes that define characters in the game, classes that define action scenes in the game, and many others.

These classes will perform their individual tasks perfectly, but we still need components that orchestrate the interactions between these classes to achieve the end goal. That is a common pattern in object-oriented programming. Think of an object-oriented application as an orchestra with different sections of different instruments. Each instrument can play their own part, but to conduct a symphony, you still need the music score for when each instrument should play, and a conductor to drive the execution. This is what the Game Framework, or Game Engine, will do for the application.

### Game Framework Classes

Shown below is a subset of the overall class diagram covering the classes that define basic game framework. As we just saw, the main program for the application is *tucker\_quest.py*. The main program does no application specific logic. It simply takes in run time arguments and environment variables, prepares the system path include the various application modules, and sets up application logging.

The actual application logic will start when the main program instantiates an object of the *GameController* class, and calls the *run()* method in it. This will turn control of the application logic to *GameController*, and the other framework classes we'll discuss next.



### The GameController Class

The game controller prepares the resources needed to execute the game, and initiates their executions. It's responsible for creating the resources that are needed to start the game, which include creating the *GameSession* and *SceneController*, and loading the critical game resources.

As part of initialization, *GameController* will download the layout of our world, since we cannot play a game without a world where things will take place. The organization of the world is discussed in detail later in section 6.2.

*GameController* will then prompt the user for their login so it can load the user data and player characters. If this a new user, GameController will drive the creation of a new user, who will then be saved to the database for future games. Existing users can select player characters previously created to use in the game, or create new characters. New users will create a character as they are creating their account. We will see later in section 6.4, the classes used to represent users and characters.

Once all the initializations and preliminary actions are completed, *GameController* creates a *SceneController* object. It will then invoke the *SceneController* *event\_loop()* method, which will start the transitions between the scenes that make up the game.

When *SceneController* we event loop exists, the *GameController* takes back control and executes the tasks needed to end cleanly end the game.

### The GameSession Class

The game session is a simple data object that maintains data that needs to be used throughout the game. It is created by importing the *game\_globals.py* file, which declares a global object *game\_session*. Classes throughout the application that need access to the game session will import the same *game\_session* object from the *game\_globals.py* file. This implements what we call a "singleton pattern", where a single variable is shared for the whole application.

Please note that this pattern should not be used as a replacement to normal interfacing between classes through method arguments and variable references. It's only used for those truly global aspects of the game. Currently, the following data is maintained in the session:

* The current player character being used
* The player location in the world
* The enemies defeated by the player
* Current active quest givers in the game
* A reference to the world (including all areas) being played
* A session id, which will be unique within a player character

Aside from being used inside the game logic, the *GameSession* object plays an important part in the ability to save games. When we save a game to continue later, what we are saving is the game session. That will have the complete record of what the player has done. It will be saved in the database, using the *GameSessionDB* class we see in section 6.7.4.

### The SceneController Class

Most games or stories, are a succession of different scenes. In fact, we could say life is succession of scenes. Scenes can range from simple encounters with characters in our story, to finding treasures, to facing complex battles with enemies. Therefore, progressing through our game is a matter of moving from scene to scene, until the game is over. In section 6.6 we will look at the different types of scenes in depth, but as in our analogy to an orchestra, The *SceneController* will be the conductor that will orchestrate the transition between scenes.

The *SceneController* primary method is *event\_loop()*. As the name implies, this method is a long loop that keeps evaluating the current the current game state and initiating scenes as required. In order to accomplish this, the *SceneController* works together with the *PlayerController* class, which we'll see next. After initialization steps, which include creating a *PlayerController* object to manage the player, *SceneController* goes into the even loop performing these tasks in order:

1. Invoke the *evaluate\_position()* method in the *PlayerController* object to determine whether the player is currently in a position that will initiate a new scene
2. If the eval method indicates a specific scene, the *SceneController* will take the steps needed to initiate the scene. This includes creating the appropriate scene object, passing the necessary data to it, and starting its execution.
3. If the player is in a regular location in the map, which is not associated with a scene, *SceneController* will provide the player with the standard navigation options to move around the world, along with other standard game options. These options are described in the next sub-section.
4. Unless the scene controller received an indication from one of the scene objects that the game should be over, or the user choses to quit, the loop will go back to the top and reevaluate the current position on step 1.

When the *SceneController* exists the event loop, when it receives an indication that the game is over, it will set a *game\_result* attribute to indicate the outcome of the game, and return it as the method return value. The game result will indicate whether a player won (reached the end cell), lost, or simply quit (which allows the player to save the session and continue later)

#### Standard Player Options

When the *PlayerController* evaluation does not indicate the beginning of a scene, the *SceneController* will always present the player with the following standard options:

* **Move** – The player is given the option to move in the standard (up/down/left/right) directions. *SceneController* will call the *move()* method in the *PlayerController* to effectuate the move.
* **Display inventory** – The current player inventory will be displayed.
* **Use an item** – *SceneController* will initiate an Apply Scene (see section 6.6.3.1).
* **Save the game** – SceneController will save the current game sessions (see section 6.7.5)
* **Help** – *SceneController* displays help information on how to navigate the game.
* **Quit** – *SceneController* will exit the event loop , and control is returned to the *GameController* class to handle the game termination.

### The PlayerController Class

As we've just seen, the *PlayerController* has a critical role in the operation of the game. It is responsible for moving the player around the world, and evaluating their position in the map to decide whether the player is at a point where a new scene needs to start. In order to perform these tasks, *PlayerController* maintains a reference to the current player in the game, the world object, and the current player position in the world. These are initially taken from the *GameSession* object in the \_\_init\_\_ method, but are subsequently updated by the controller itself.

Some of the key operations supported by the player controller include:

* **evaluate\_position()** – This method evaluates the position of the player in the current area map, and checks the cells immediately surrounding it to determine whether a new scene needs to be started. Some scenes, like finding an item, are activated if the player is on top of the item cell in the map, but other scenes, such as NPC encounters, can be triggered if the player is located adjacent to the NPC. The method returns the scene type and cell location data, which the *SceneController* can use to initiate a scene.
* **move(direction)** – This method moves the player to a new location based on a standard up/down/left/right move.
* **update\_area(area\_id, row, col)** – This method moves the player to a completely different area. This is necessary during gate scenes, where a player is not simply moving to an adjacent cell, but to a completely different area.

### The Score Class

The *Score* class is responsible for maintaining the attributes that determine a player score, and calculating the overall score. At this time, the score is a simple weighted calculation of various elements in the game session, such as the number of items found, enemies defeated, etc. When we want to update the score, the *calculate\_score()* method is invoked, and it will calculate the score based on the current session, and update the score attributes. The historical scores for users are persisted using the *ScoreDB* class.

### Unique Ids

One critical aspect of the overall game framework, is the use of unique ids, and their specific format. Most resources in the application will have unique ids to identify them. With few exceptions, the ids will follow a 5-character format, “<type><number>”, where type is a single character identifying the type of resource, and number is a four-digit number. Examples:

* “s0001”, “s0202”, etc are Ids for swords
* “e0101”, “e1002”, etc are Ids for enemies
* “a0021”, “a3002”, are Ids for armors
* And so on …

The character used as prefix is arbitrary, but it helps in the identification of the resource type. Ids are case sensitive, so both “s” and “S” can be used as prefixes for different types of resources. In fact, they don’t even have to be letters. “$” would work just as well. For that matter, the use of four-digit numbers is also arbitrary, because they could be anything. However, the use of numbers makes it easier to visualize the uniqueness.

Using 5 characters for every id is a convention used primarily to aid in the visualization of the world maps. This will become clear when we discuss the representation of a World map.

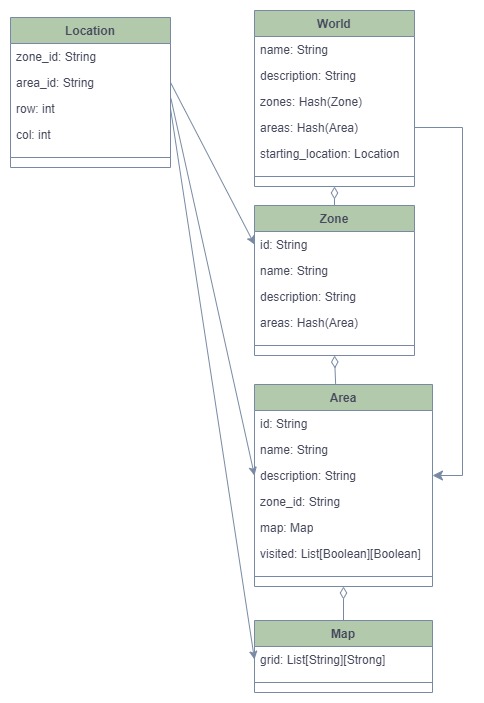
## World Design

The "World" is the canvas where the game takes place. It is divided in multiple areas, with different characteristics. As the game progresses, the player will move through different areas of the world. Programmers will have the ability to customize and add new areas to the world.

### World Classes

This is a subset of the overall class diagram covering the classes that define the world of this game. The basic design is:

* A World contains a collection of Zones
* A Zone contains a collection of Areas
* An Area contains a Map for it
* Areas and Zones are identified by a unique id
* A Location object marks a specific location in the World
* Zones and areas have text description fields. They will be used as part of the user interface in the game.



Most of the game operations happen in the context of an *Area*, and more specifically, the map in that area. A *Zone* is more of a logical division of the world, to help identify significant differences in characteristics. You can think of a Zone as equivalent to a "continent" in our real world. Most of real-world action happens in different countries (which would be analogous to the Areas here). Continents simply group countries in similar geographic locations.

### JSON Data Representation

The class objects are mapped to JSON structures in the database. Below is an example of a world representation in JSON. The *WorldDB* class described in section 0 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* Each area has a map "grid" attribute, and we'll discuss the details of that representation in the next section.
* The "visited" attribute is used in the user interface to register which map locations have been visited by the player.

{

"Tucker World": {

"name" : "Tucker World",

"description" : "A dangerous world full of wonders",

"starting\_location": {

"zone\_id": "z0100",

"area\_id": "a0101",

"row": 10,

"col": 5

},

"zones": {

"z0100": {

"id": "z0100",

"name": "Dungeon",

"description": "This is a dark temple …",

"areas": ["a0101", "a0102"]

}

},

"areas": {

"a0101": {

"id": "a0101",

"name": "Dungeon - first floor",

"description": "This is the first area …",

"zone\_id": "z0100",

"map": {

"grid": [ *<see next section>* ]

},

"visited": null

### Map Representation

The area maps play a critical role in the game, and are not simply a means to navigate the world. The way we add items, enemies, gates and many other elements into the game play, is by placing them on the map. More specifically, we will add their ids to the map. We will discuss this in subsequent sections.

#### Map Data Structure

The area maps are grids represented as a two-dimensional array. In Python, that’s essentially a List of Lists. For instance, a 5 by 5 grid like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BBBBB | BBBBB | BBBBB | BBBBB | BBBBB |
| BBBBB | WWWWW |  |  | BBBBB |
| BBBBB | WWWWW |  | WWWWW | BBBBB |
| BBBBB | WWWWW |  | WWWWW | BBBBB |
| BBBBB | BBBBB | BBBBB | BBBBB | BBBBB |

Would be represented in Python as:

grid = [["BBBBB", "BBBBB", "BBBBB", "BBBBB", "BBBBB"],

["BBBBB", "WWWWW", " ", " ", "BBBBB"],

["BBBBB", "WWWWW", " ", "WWWWW", "BBBBB"],

["BBBBB", "WWWWW", " ", "WWWWW", "BBBBB"],

["BBBBB", "BBBBB", "BBBBB", "BBBBB", "BBBBB"]]

#### Map Cell Types

As we mentioned earlier, the value of a map cell can indicate not only the physical make up of the world (walls and borders), but also the position of game elements (items, enemies, gates, etc). In order for the application to recognize the type of element in a map cell, we need to follow a consistent naming convention. In order to make the map easier to visualize, we use 5 character for everything that goes in a map (we'll see an example next). We use 5 characters, because as we discussed in section 6.1.6, 5 characters is what we use to represent the unique ids that define our game elements. We will need the full 5-character id to process a game scene, but in order to determine the type of cell in the map, the application only looks at the first character. Using just the first character makes it easier to map cell types to game elements and display representations. These mappings are found in the *map\_config.py* configuration file, which we discuss in section 8.1.5.

At this time, these are the valid cell values you might currently find in the map. The first characters highlighted, because that's the only thing used in determining the cell type:

Cell values indicating a physical aspect of the world:

* "BBBBB" – A border around the whole map
* "WWWWW" – A wall or general blockage in the world
* " " – An open space players can walk through

Cells representing non-player characters:

* "N\*\*\*\*" – A story character
* "E\*\*\*\*" – An enemy character
* "Q\*\*\*\*" – A quest giver character

Cells representing game items:

* "h\*\*\*\*" – A potion item
* "p\*\*\*\*" – A protection spell item
* "a\*\*\*\*" – An armor item
* "m\*\*\*\*" – A spell item
* "s\*\*\*\*" – A sword item
* "t\*\*\*\*" – A treasure item
* "k\*\*\*\*" – A key item

Cells representing other special game events:

* "SSSSS" – Used only once in the first map to mark the starting point in the game
* "XXXXX" – Used only once to mark then end goal of the game
* "G\*\*\*\*" – A gate element, which will initiate a transfer to a different area

Please note that I said these are the cell values you **currently** find in the map, because the application is designed to be expandable. We will see in section 8 how we do that, and it may involve adding a new cell type in our configuration files.

#### Tucker Quest Map Example

Using the map structure and cell types defined, a map in Tucker Quest will place different game elements in the map array. Below is an example of a full map representation. This would go in the "grid" attribute of the area structure we saw earlier.

"map": {

"grid": [

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","BBBBB"],

["BBBBB","WWWWW","G0001","WWWWW"," "," "," ","BBBBB"],

["BBBBB","WWWWW"," "," ","E0201","WWWWW"," ","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW"," ","BBBBB"],

["BBBBB"," ","m0003"," ","s0003"," "," ", BBBBB"],

["BBBBB"," ","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","BBBBB"],

["BBBBB"," "," ","p0002"," ","h0002","WWWWW","BBBBB"],

["BBBBB","a0002","WWWWW","WWWWW","WWWWW"," ","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW"," ","N0301","E0101","WWWWW","BBBBB"], ["BBBBB","WWWWW","WWWWW","SSSSS","WWWWW","WWWWW","WWWWW","BBBBB"], ["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"]

]

Some cells in this example map are used to describe the physical make of the area:

* "BBBBB" – A border around the whole map
* "WWWWW" – A wall or general blockage in the world
* " " – An open space players can walk through
* "SSSSS" – The starting point in the game

The other cells represent a variety of game elements. By placing the id of an element in a cell, that specific element will be found in that location. In this particular example, we see the following highlighted elements:

* "N0301"– A story character
* "E0101"– An enemy character
* "h0002"– A potion item
* "p0002"– A protection spell item
* "a0002"– An armor item
* "m0003"– A spell item
* "s0003"– A sword item
* "E0201"– Another enemy character
* "G0001"– A gate element

This is just one example of one are in the map, which is made intentionally busy, so that we could illustrate various types of elements. Building a world in this application, will be a matter of creating many such maps, and laying out different game elements however we see fit. We discuss these different game element types in subsequent sections.

Please note **an important border convention** used in the game. All maps should be completely surrounded by border cells ("BBBBB" cell values) in the outer rows and columns. This serves two key purposes.

* First, when this is interpreted by out user interface class, it provides a nice visual marker of the overall dimensions of the area we are in.
* Second, and perhaps more important, it simplifies some of the game programming, because it eliminates the need to always check for index out of bound conditions. Since the first and last element of a row is always a non-playable cell, it is always safe to check all the cells around a player. Designing a map without a border wall, will almost certainly cause an out of bounds index exception in the game.

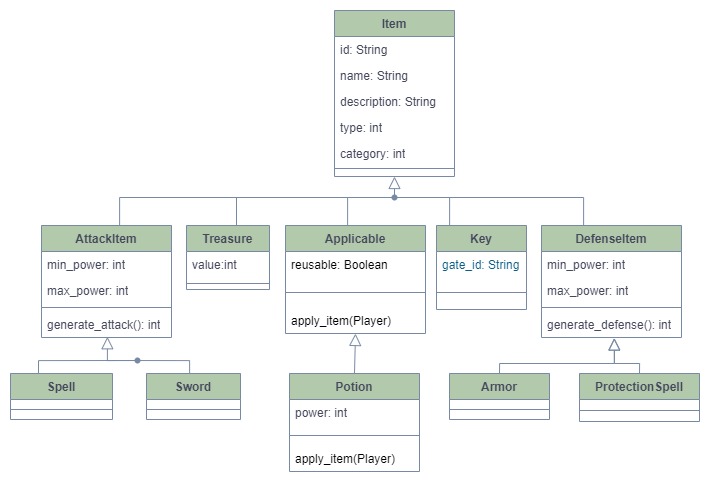
## Item Hierarchy

Items are general "things" the player will find during the game. In most cases they'll be things that the player needs to progress through the game, such as weapons, potions, keys, or armor. However, they can also be items that simply add "character", such as different types of treasures. Programmers will have the ability to customize and add new types of items to the game (see section 8.2).

### Item Classes

This is a subset of the overall class diagram covering the classes that define various items found across the game. The basic design is:

* All items inherit from the base *Item* class, which defines key characteristics for all items
  + All items have a unique *id.*
  + All items have a *type* attribute, defining their specific type (SWORD\_ITEM, POTION\_ITEM, ARMOR\_ITEM, etc).
  + All items have a *category*attribute, which defines a broader category. Current categories in the game are ATTACK\_ITEM, DEFENSE\_ITEM, APPLICABLE\_ITEM or GENERAL\_ITEM.
  + All items have a description, that will be used for the user interface in the game
  + The base class *Item*implements an *\_\_eq\_\_()* method that defines items being equal if they have the same id.
* Items which are used in attacking are placed under an *AttackItem* sub-class, which defines attributes and operations specific to attacking
* Items which are used in defending are placed under a *DefenseItem* sub-class, which defines attributes and operations specific to attacking
* Much of the Item hierarchy is meant to support the ability to easily add new item types to the game.



### Item Types

There are a few different types of items currently implemented in the application. As we've seen in the class diagram, they share some common attributes, but they may also have very different purposes. Items get picked up by players as they encounter them in the game, or they may be gained from NPC encounters. They will be kept in a player's inventory. Non-player characters may also have items, but in their case the item(s) is assigned to the NPC via configuration.

Note that not every item is usable by every player type. A wizard for instance, can use spells, but not swords. A knight can wear armor, but a wizard cannot. Only a character type "knows" what items it can use, so that will be defined in the specific player character classes, through the *can\_use()* operation. See section 6.4.1 for more information on player characters.

#### Attack Items

Attack items derive from the *AttackItem*class, and are used by both player and non-player characters to attack during a battle. They each have a minimum and maximum attack power, and a *generate\_attack()* operation to generate an attack value for the item. There is a default implementation for *generate\_attack()* that can be used any item, which simply generates a random number between the minimum and maximum power. However, programmers are welcome to specialize the operation in their derived classes for customized behavior.

At this time the only types of attack items implemented are spells and swords. However, the programmers are encouraged to create new ones, and incorporate them into the game.

#### Defense Items

Defense items derive from the *DefenseItem*class, and are used by both player and non-player characters to defense during a battle. They each have a minimum and maximum defense power, and a *generate\_defense()* operation to generate a defense value for the item. There is a default implementation for *generate\_defense()* that can be used any item, which simply generates a random number between the minimum and maximum power. However, programmers are welcome to specialize the operation in their derived classes for customized behavior.

At this time the only types of defense items implemented armor and protection spells. However, the programmers are encouraged to create new ones, and incorporate them into the game.

#### Applicable Items

Applicable items, are items that can be applied to generate an action in the game. It is a very general term, because it opens the door for very general application of the concept in the future. An applicable item implements an operation *apply\_item().* That operation can perform just about anything the programmer is able to implement. An applicable item also has a *reusable*attribute, which determines where that particular item can be reused or not. If an item is not reusable, it will be removed from the player's inventory after it's used.

At this time, the only type of application item implemented is *Potion*which a player can use to restore health during the game. So, the *apply\_item()* operation in that case will interact with the player and restore its health.

#### Key Items

A key is a special item used to open gates in Gate Scenes. Some gates will require a key to open, and the player will need the corresponding key to proceed. See a discussion of Gate Scenes in section 6.6.3.

#### Treasure Items

A treasure is an item of value, but without a direct impact in game action. They will have a value that can impact a player score, and they can also be referred to as requirements for a quest. But the item within itself doesn't perform a specific task.

### Item JSON Data Representation

Once again, the class objects are mapped to JSON structures in the database. Below is an example of some item representations in JSON. The *ItemDB* class described in section 0 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* The *type* field is an integer, but it will be mapped to easily recognizable names specified in the *game\_types\_config.py* configuration file described in section 8.1.
* Note that different types of items will have slightly different JSON attributes, since the corresponding class has different attributes.
* The item category is not set in the JSON data, because each specific item class constructor will define what category It belongs to.

{

"s0001": {

"id": "s0001",

"name": "Wooden sword",

"description": "a rough wooden sword",

"type": 100,

"min\_power": 50,

"max\_power": 75

},

"t0003": {

"id": "t0003",

"name": "Silver necklace",

"description": "a silver necklace",

"type": 1,

"value": 1

},

"h0002": {

"id": "h0002",

"name": "Medium healing potion",

"description": "a bottle labeled Healing Potion - Power 100",

"type": 400,

"power": 100,

"reusable": false

},

"a0002": {

"id": "a0002",

"name": "Tin armor",

"description": "a rusty tin armor",

"type": 200,

"min\_power": 40,

"max\_power": 50

},

"m0003": {

"id": "m0003",

"name": "Fire spell",

"description": "teach wizards to launch a stream of fire flames",

"type": 101,

"min\_power": 100,

"max\_power": 125

}

}

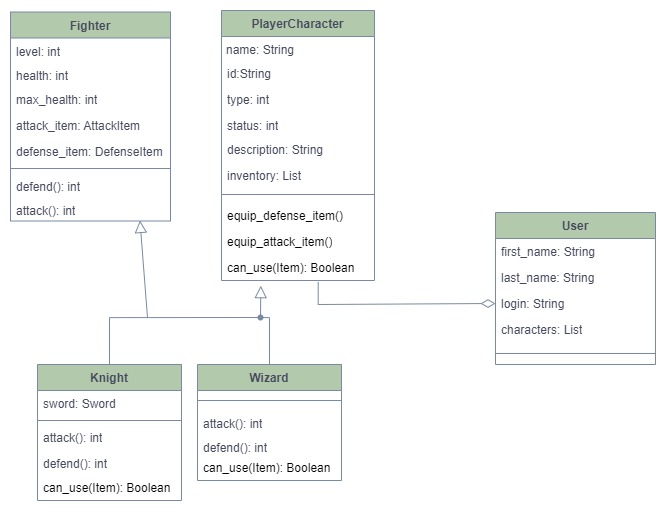
## User and Player Character Hierarchy

A user is a person playing the game. Someone with a regular first name, last name, and a login to enter the game. A Player Character is a character created by the user to play the game. For instance, a user "Fernando", might play the game as "Fernando the Knight" or "Fernando the Wizard". A user can create multiple characters to play the game. Programmers will have the ability to customize and add new types of playing character types to the game.

### User and Player Classes

This is a subset of the overall class diagram covering the classes that define the user and player characters. The basic design is:

* The *User* class represents a user of the game, and will include a list of playing characters for the user
* The *PlayerCharacter* base class defines attributes that are common for all types of playing characters.
  + All characters have a unique *id.*
  + All characters have a *type* attribute, defining their specific type, which currently can be a KNIGHT or a WIZARD.
  + All characters have an inventory of items that can be used throughout the game.
  + All items have a description, that will be used for the user interface in the game
  + All player characters implement a *can\_use()*operation, which lets the game engine know which item types can be used by this player character type.
* The *PlayerCharacter* class is derived from the *Fighter* class, which defines the attributes that are related to fighting actions.
  + The *Fighter* class is also inherited by the *Enemy* hierarchy we'll see later, and that provides a consistent set of attributes across combatants in a battle
* Much of the Item hierarchy is meant to support the ability to easily add new item types to the game. This is discussed in section 8.3.



### Users and Players JSON Data Representation

#### Users JSON Data Representation

This is an example of user representations in JSON. The *UserDB* class described in section 0 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* In addition to basic user information, the structure will include a list of player characters.
  + The list of characters will only include the character Id, and not the full player character data. The full character data is stored in a separate structure, which we'll see later.
  + The player characters are structured such that the characters are split different lists for each character type. So, in the example below, characters under the key "1" are knights, and characters under the key "2" are wizards. The numbers 1 and 2 are defined the *game\_types\_config.py* file we discuss in section 8.1.1.

{

"fttg": {

"login": "fttg",

"first\_name": "Fernando",

"last\_name": "Teixeira",

"characters": {

"1": [

"fttg.FernBeast"

],

"2": [

"fttg.FernWiz"

]

}

},

"finn": {

"login": "finn",

"first\_name": "Finn",

"last\_name": "Teixeira",

"characters": {

"2": [

"finn.FinnWizard"

]

}

}

}

#### Player Characters JSON Representation

Below is an example of players representations in JSON. The *PlayerCharacterDB* class described in section 0 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* The *type* field is an integer, but it will be mapped to easily recognizable names specified in the *game\_types\_config.py* configuration file.
* The id attribute for a player character does not follow the 5 character format we've seen used in other resources. That's because player characters will not be placed in specific locations in a map, and therefore that convention used for visualization is not necessary. So instead, we use a combination of the user id and a character name. Prefixing the id with the user name allows for different users to have characters with the same name. In the example below matches the characters that has been referred to in the User JSON representation we saw earlier.

{

"fttg.FernBeast": {

"id": "fttg.FernBeast",

"name": "FernBeast",

"pronoun": "he",

"type": 1,

"status": null,

"description": null,

"level": 1,

"max\_health": 1000,

"health": null,

"attack\_item": null,

"defense\_item": null,

"inventory": null,

"sword": null

},

"fttg.FernWiz": {

"id": "fttg.FernWiz",

"name": "FernWiz",

"pronoun": "he",

"type": 2,

"status": null,

"description": null,

"level": 1,

"max\_health": 1000,

"health": null,

"attack\_item": null,

"defense\_item": null,

"inventory": null

},

"finn.FinnWizard": {

"id": "finn.FinnWizard",

"name": "FinnWizard",

"pronoun": "he",

"status": null,

"description": null,

"type": 2,

"inventory": [],

"level": 1,

"max\_health": 1000,

"attack\_item": null,

"defense\_item": null,

"health": 1000

}

}

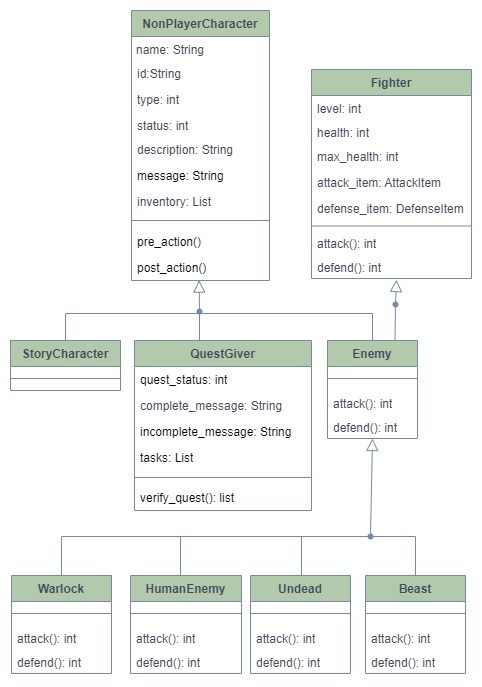
## Non-Player Character Hierarchy

A Non-Player Character, or NPC as it is commonly referred to as in games, is a character in the game play that is not the player. Essentially those are computer generated characters that play different roles in the story. They can have a very active role as an enemy that the player must fight, or can simply be an element of "flair", as a character that runs into the player and has a witty message. Programmers will have the ability to customize and add new types of NPCs to the game.

### Non-Player Character Classes

This is a subset of the overall class diagram covering the classes that define the user and non-player characters. The basic design is:

* The *NonPlayerCharacter* class defines attributes that are common for all types of playing NPCs.
  + All characters have a unique *id.*
  + All characters have a *type* attribute, defining their specific type, which currently can be a WARLOCK, HUMAN\_ENEMY, UNDEAD, BEAST, STORY\_CHARACTER or QUEST\_GIVER.
  + All characters have an inventory of items that can be used throughout the game.
  + All items have a description, that will be used for the user interface in the game
  + The base class *NonPlayerCharacter* implements an *\_\_eq\_\_()* method that defines NPCs being equal if they have the same id.
* Among NPCs, the *Enemy* sub-class differentiates those types of characters which will participated in battles with the player.
  + The *Enemy* class is derived from the *Fighter* class, which defines the attributes that are related to fighting actions.
  + As we saw earlier, the *Fighter* class is also inherited by the *PlayerCharacter* hierarchy, and that provides a consistent set of attributes across combatants in a battle.
* There are currently four types of enemy NPCs: warlocks, human enemies, undead, and beasts.
  + Having different types allows for customization of characters. For instance, the attack used by an undead enemy is different than the ones used by others. Instead of using some type of weapon to generate damage, an undead enemy drains a percentage of the player's health. Other specializations can be added in the future
* The other NPCs at this time are story characters and quest givers
  + Story characters are mostly passive characters the player may encounter in the story. At most they will provide and hint in the form of a message, or perhaps donate items.
  + Quest givers will give quests the player can undertake to win a reward. For example, a quest may require a player to collect certain items, and may reward the player with a key to advance to another area of the game. The *QuestGiver* class plays a very active role in controlling quest scenes, which we'll see in section 6.6.3.6.



#### The pre\_action and post\_action Methods

Much of the Item hierarchy is meant to support the ability to easily add new elements to the game, including new types of non-player characters. One of the ways customization can be added, is through the *pre\_action()* and *post\_action()* methods in the base class *NonPlayerCharacter*. These methods will be called during NPC encounter scenes, at their beginning and end. The methods are defined in the base class, but have no actual implementation there. However, derived NPC classes can override the methods if they want to implement any specialized behavior.

The complete details of how to create new non-player characters is described in section 8.4.

### Non-Player Characters JSON Data Representation

Below is an example of players representations in JSON. The *NonPlayerCharacterDB* class described in section 6.7.4 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* The *id* attribute for a non-player character follows the 5 character format we've discussed before. That allows is to place NPCs in specific locations in the map, as we saw in section 6.2.3.2. We also use the first character in the id to help identify the type of NPC, so we can display a different symbol for each.
* The *type* field is an integer, but it will be mapped to easily recognizable names specified in the *game\_types\_config.py* configuration file described in section 8.1.
* Note that different types of NPCs may have slightly different attributes depending on their types. In this example, we see:
  + The first two NPCs in the list are enemies, so they have attributes related to the *Fighter* class *Enemy* derives from: *health, max-health, level, attack\_item-Id*, and *defense\_item\_id.*
  + Even within enemies, you can see a difference in that Zombies are of type *Undead*, so they have an extra attribute *attack\_drain* to define how much (percentage) they'll drain an opponent's health during an attack
  + The last example, with id *N0001*, is a story character, so he does not have any fighter elements. Instead, he has a long message warning players of dangers, and an inventory of items that will be given to the player (see *EncounterScene* in section 6.6)

{

"E0101": {

"id": "E0101",

"name": "Lady Frankenstein",

"pronoun": "she",

"type": 4,

"status": null,

"description": "a slow-moving female zombie",

"level": 1,

"max\_health": 100,

"health": null,

"attack\_item\_id": null,

"defense\_item\_id": "a0001",

"inventory": null,

"message": "Brains ... \nI need brains ...",

"attack\_drain": 10

},

"E0201": {

"id": "E0201",

"name": "Martel the warlock",

"pronoun": "he",

"type": 2,

"status": null,

"description": "a warlock dressed in a red robe",

"level": 1,

"max\_health": 100,

"health": null,

"attack\_item\_id": "s0002",

"defense\_item\_id": "a0002",

"inventory": null,

"message": "You will dread the day you faced Martel."

},

"N0301": {

"id": "N0001",

"name": "Wise old man",

"pronoun": "he",

"type": 100,

"status": null,

"description": "an old man, with a friendly countenance",

"inventory": ["s0001", "a0001", "m0001", "p0001"],

"message": "Hello there young adventurer. \nIt has been many years since I've seen the likes of you entering this temple.\nThere are many dangers ahead.\nI suggest you look around and find some means of attacking and protecting yourself.\nHere, let me get you started with a couple of things. "

}

}

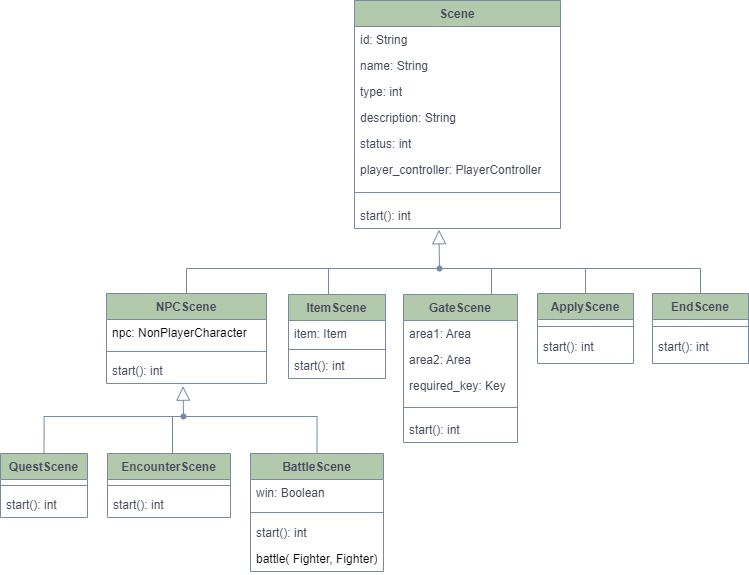
## Scene Hierarchy

"Scenes" are where most actions take place during the game. During a game session, a player will either be walking around through the world, or upon encountering a game element, will participate on a scene. Scenes can range from simple encounters with NPCs, to complex battles with enemies.

### Scene Classes

This is a subset of the overall class diagram covering the classes that define various scenes currently implemented in the game. The basic design is:

* All scenes inherit from the base *Scene* class, which define key characteristics for all items
  + Scenes have a unique *id.*
  + All scenes have a *type* attribute, defining their specific type (QUEST\_SCENE, BATTLE\_SCENE, APPLY\_SCENE, etc). We'll discuss the specific details of the different scene types in the next section.
  + All scenes have a description, that will be used for the user interface in the game
* There is a sub-class *NPCScene*, which encapsulates common actions and attributes for scenes that involve interactions with NPCs.
* Much of the scene hierarchy is meant to support the ability to expand the code for new types of scenes in the future. Note however, that because scenes are a tightly coupled with the mechanics of the game, new scenes can't be added by simple configuration changes as we do for Items and characters (discussed in section 8). That being said, it would not be hard to adapt the existing code base for it.



### Scene Activation

Scenes are activated by the *SceneController*, perform the necessary actions for the scene, then return control to the *SceneController* . In most cases (but not all), a scene is initiated when a player moves near on onto a specific map cell. As we discussed earlier, the *PlayerController* *evaluate\_position*() method will detect the special cell, and return the information to the *SceneController*. In the next sections we describe the various scene types, and what map element or action initiates it.

### Scene Types

There are multiple types of scenes in the game, and although they share some attributes seen above, they can have very different purposes and behaviors.

#### Apply Scene

This scene is activated when a player decides to use an applicable item (see discussion on applicable items in section 6.3.2.3). And applicable item does not have to be used immediately once it’s picked up, so a player can accumulate various items in its inventory. When a player decides to use an item (an option provided as part of game interactions), *ApplyScene* will guide the player through a selection of which applicable item (if any) they want to use. It will then invoke the *apply item*() operation on the selected item.

If the item in question is not reusable (which is the case with the *Potion*item currently implemented), then the *ApplyScene* class will handle removing it from the player's after it has been applied.

#### Gate Scene

This scene is activated when a player moves into a gate cell in the area map. That is an indication the player has reached a doorway into a different area. Each gate is unique, and represents a connection between two locations in two areas. Unlike some of the other scenes, a gate cell will map to a specific id for a gate scene. We'll see later how this is represented in JSON, and in the *GateScene* object.

The GateScene class handles the updating of the player's position in the world, such that they are essentially transported from one area map to another. A gate supports two-way transportation, meaning that the same gate is used to connect two areas in either direction.

A gate may require a key or not. If a key is required, the player will be able go through if they have the required key item in their inventory.

#### Item Scene

This scene is activated when a player moves into a cell in the area map which contains an item. The *ItemScene* class will implement the basic user interactions where an item is described to the player, and then added to the player's inventory if applicable.

Items are only added to the player's inventory if the character type can use the specific item type. *ItemScene* will invoke the *can\_use()* operation in the player, passing in the Item picked up, to determine that.

#### Encounter Scene

This scene is activated when a player moves into the proximity (adjacent cells) of a story character on the area map. The *EncounterScene* class will implement the basic user interactions where a story character message is presented to the player. If a story character has items in its inventory, the class will also add theses items to the player's inventory if applicable.

Items are only added to the player's inventory if the character type can use the specific item type. *ItemScene* will invoke the *can\_use()* operation in the player, passing in the Item picked up, to determine that.

#### Battle Scene

This scene is activated when a player moves into the proximity (adjacent cells) of an enemy character on the area map. The *BattleScene* class will implement the user interactions that will drive the combat between the player and the enemy from beginning to end.

The *BattleScene* is one of the most complex scene currently implemented. It performs these general steps:

1. Introduce the enemy encountered, and display the enemy's taunting message
2. Invoke the methods in the player and enemy opponent to equip the attack and defense items they will use in battle

* Enemies are assigned specific items in configuration
* Player's will have the option of selecting items available in their inventory

1. Begin the battle loop
   1. Player attacks the enemy
      * The attack is done by invoking the *attack()* operation on the player
      * The *attack()* operation will handle generating the attack, and applying the appropriate damage to the enemy opponent
   2. Player attack results and resulting enemy health are displayed
      * If the enemy's health is zero at this point, the battle ends, and the loop exists
   3. Enemy attacks the player
      * The attack is done by invoking the *attack()* operation on the enemy
      * The *attack()* operation will handle generating the attack, and applying the appropriate damage to the player
   4. Enemy attack results and resulting player's health are displayed
      * If the player's health is zero at this point, the battle ends, and the loop exists
2. After the end of the battle the *player\_won* attribute will be set to signal the result of the battle
3. If the player won, applicable items in the enemy's inventory are "looted" by the player, meaning they will be transferred to the player's inventory. The enemy is also removed from the map, so you will not keep fighting the same enemy over and over.

##### Battle Stats

At each turn of a battle, the *BattleScene* class will log stats for each attack (both for player and enemy). These are saved in a CSV file, under the logs folder. This is done strictly for analysis of data offline, and does not impact the game play in any way.

#### Quest Scene

This scene is activated when a player moves into the proximity (adjacent cells) of a quest giver character on the area map. The quest scene has some unique characteristics not share by other scenes, because a quest scene doesn't really complete in one visit. There are two different flows for quest scene, depending on the game situation. When the quest scene first starts, it immediately checks if there is an active quest going on. This is done by checking the global *GameSession* object.

If there is no active quest for this quest giver, the quest scene will follow this new quest flow:

1. The scene gets a "fresh" *QuestGiver* object for the non-player character managing this quest from the database.
   * Much of the information pertaining to the quest is actually stored in the *QuestGiver* class. From an object-oriented perspective, that makes sense, since it would be the quest giver that knows what the quest is all about.
2. With information taken from the *QuestGiver*, the *QuestScene* object will display the appropriate messages informing the player of what the quest is, and the tasks needed to be completed.
   * Tasks are either items that need to be collected, or enemies that need to be defeated
3. *QuestScene* adds this quest giver to the active list maintained in the global *GameSession* object.
4. The scene ends after providing quest information, but unlike other NPC encounters, the *QuestGiver* object is not removed from the map. That's because the player will need to return to the quest giver to complete the quest.
5. After the scene completes, the *SceneController* implements a simple counter "cool off" mechanism to make sure the player position won't immediately re-initiated a quest scene. The player will eventually need to return to the QuestGiver, but obviously not immediately after.

If the player moves around, and eventually returns to a quest giver, *QuestScene* will find the quest giver in the global *GameSession* object, and therefore it will know to follow the different returning quest flow below:

1. The scene gets a reference to the *QuestGiver* non-player character managing this quest from the global *GameSession* object.

* It is critical that the QuestGiver object is retrieved from the game session, and not fresh from the database, because the quest giver object tracks the progress of a quest. So, we want to retrieve the same quest giver that gave out the quest, and was saved in the session in the new quest flow above.

1. *QuestScene* will invoke the *verify\_quest()* method in the *QuestGiver* object, which will determine whether the quest was completed.

* *QuestGiver* will compare its tasks with the players inventory and the enemies defeated by the player

1. Depending on whether the quest was completed, there are two possible flows
   1. If the quest was completed, an appropriate success message is displayed, and the player is given whatever items were in the quest giver inventory
      * The *QuestGiver* object is now removed from the map, and from the active quests list in the *GameSession*
   2. If the quest was not completed, an appropriate message is played indicating the quest was not completed, and the player is told which remaining tasks are still unfulfilled
      * In this case, the *QuestGiver* object remains in the map, and in the *GameSession*
      * The player can return later after attempting to complete the tasks, and it would start this returning flow again

#### End Scene

This scene is activated when a player moves into a cell on or adjacent to the end of game cell. The *EndScene* class will implement the basic user interactions required when a player is approaching the end of the game. *EndScene* can be initiate in two distinct scenarios. First, if a player is adjacent to the end cell, it will simply display a suitable foreshadowing message. This is done just to allow for building some expectation. If the player then moves into the end game cell, EndScene will process the actual end game sequence.

In addition to displaying messages (with the expected congratulatory text),

### Scene JSON Data Representation

Most scenes do not require a JSON representation to be stored. That's because they are defined by the game element that activates them. For instance, for a Battle or Encounter scene, what happens is driven by the NPC that initiates the scene. The Quest Scene is managed by the quest giver. Same goes for an Item Scene or Apply Scene. At this time, the only scene that has to be configured in JSON is the gate scene.

#### Gate Scene JSON Data Representation

Below is an example of some item gate scene representations in JSON. The *GateSceneDB* class described in section 6.7.4 will convert the JSON representation to and from Python class objects. Some key points in the representation:

* The *id* value is what will be used in the area map to indicate we arrived at a gate, and should start a gate scene
* A gate connects two areas in both directions, so we have to specify these two areas in the *area1*  and *area2* attributes
* Both *area1* and *area2* specify not only an area id, but also a specific row/col location where the gate is located.
* If this gate requires a key, the *required\_key* attribute will be the id for the specific key item required to open the gate.

{

"G0001": {

"id": "G0001",

"name": "a0101-a0102",

"description": "a plain wooden door",

"status": null,

"area1": {

"id": "a0101",

"row": 2,

"col": 4

},

"area2": {

"id": "a0102",

"row": 9,

"col": 1

},

"required\_key": null

},

"G0003": {

"id": "G0003",

"name": "a0102-a0103",

"description": "an iron gate",

"status": null,

"area1": {

"id": "a0102",

"row": 3,

"col": 10

},

"area2": {

"id": "a0103",

"row": 9,

"col": 1

},

"required\_key": "k0002"

}

}

## Database Layer

Most of the elements that drive this application are meant to be persisted in a database. This includes users, characters, items, world area definitions, gates, and quests. Following the Object-Oriented principle of isolation, the database interface is separated into separate layer of objects. This layer will insulate the rest of the application from changes in the database engine being used.

### Data Representation

As we have discussed before, we have chosen to use JSON as the primary way to represent application data. This means that all our class objects will be loaded from JSON structures, and persisted as JSON structures. The main reasons for that are:

* Python class objects can easily be converted into JSON structures. That's because the data in a Python object is stored as a Python dictionary, and there is a very natural relationship between Python dictionaries and JSON structures.
* Likewise, JSON structures can be easily mapped into class objects.
* The standard json Python module will support the most common operations for manipulating JSON
* Last but not least, JSON structures work well with NoSQL databases such as AWS DynamoDB

### Converting to and from JSON Data

Since we have chosen to persist our data using a JSON representation, it is important that the application classes support this in a consistent manner. Although the database classes will drive the process of converting to and from JSON, following a good object-oriented design, it is the game element classes (like *Sword, Beast, Wizard,* etc) themselves that know how to convert their own data. With that in mind, the application defines a standard where any game object that needs to be persisted will support the following methods:

* **from\_dict**(dict) – This is a **static** method takes a Python dictionary as an input, and will return a new object of the class. The Python dictionary passed in, should be a direct mapping of a JSON structure, which can be easily created using the standard Python json library *loads*() function.

Remember that a static method is one that can be called directly without creating an object. So, you would call this static method to create an object from a Dictionary. We'll see and example next.

* **\_\_str\_\_()** – In Python, the *\_\_str\_\_()* method is a standard method a class can implement to specify how an object should be represented as a string. The \_\_str\_\_ method is invoked by Python if you apply a str() conversion to an object, or simply print it.

In our case, our standard is that the *\_\_str\_\_()* method should always call the standard json library *dumps()* method, which will convert the object into a JSON string. This will give us a JSON structure suitable for persisting our objects to a database. It is also useful to use for logging.

### Example: JSON Conversion

To help understand the process, let's look at an actual example for a game item. In this case, we will show the code in the *Treasure* item class. We will see in the next section how the *ItemDB* class will have retrieve JSON structure from the database, and load it into a Python dictionary. So the next step would be to create a Treasure object from that dictionary.

Looking at the code block below, note how the ***from\_dict*** method pulls in data from the dictionary, instantiates an object of the *Treasure* class, and returns it. It may appear strange that a method inside the *Treasure* class creates an object of the *Treasure* class, but remember that *from\_dict* is a static method. A static method can be used without an object, like a regular Python function. We place it under the *Treasure* class because it clearly supports functionality for that class, but technically speaking we could have implemented it as a stand along function as well.

The *\_\_str\_\_()* method code might be a little harder to visualize. First, it relies on the fact that Python automatically stores the data in a class object as a Dictionary. To get access to that dictionary, you invoke the *\_\_dict\_\_()* method. Again, that's something Python does already, and not something we had to implement. The "*default=lambda o: o.\_\_dict\_\_*" part tells the dumps function that if it finds a complex type inside our object (which will happen when an object contains another object), it should use that object's dictionary in the conversion as well. Again, that may look a little complicated, but the good news is that piece of code can be reused as is for all our object types.

**class Treasure**(Item):

def \_\_init\_\_(self, id=None, name=None, description=None, value=None):

*[... constructor code ommitted ...]*

**@static method**

# create object from a dictionary

def **from\_dict(item\_dict):**

return **Treasure**(id=item\_dict["id"], name=item\_dict["name"],

description=item\_dict["description"],

value=item\_dict["value"])

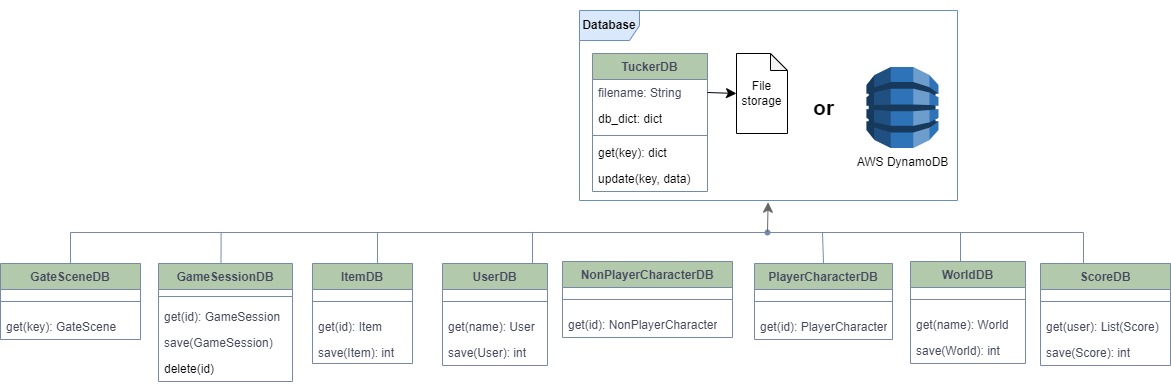
# return string representation of object, as a JSON structure

def **\_\_str\_\_(self):**

return **json.dumps**(self, default=lambda o: o.\_\_dict\_\_)

### Database Classes

The database classes are responsible for saving or retrieving data from the database, and returning object of the appropriate class. This is a subset of the overall class diagram covering the classes that define various scenes currently implemented in the game.



At this time, we have the following database classes implemented:

* **WorldDB** – Supports loading and saving world definition from the database. The methods will take in and/or return objects of the *World* class, including the *Zone* and *Area* objects in it.
* **UserDB** – Supports loading and saving users from the database. The methods will take in and/or return objects of the *User* class.
* **PlayerCharacterDB** - Supports loading and saving player characters from the database. The methods will take in and/or return objects of one of the specific derived *PlayerCharacter* classes, such as *Wizard* or *Knight*.
* **NonPlayerCharacterDB** - Supports loading and saving non-player characters from the database. The methods will take in and/or return objects of one of the specific derived *NonPlayerCharacter* classes, such as *Warlock,* *Beast, StoryChracter,* etc.
* **ItemDB** - Supports loading and saving items from the database. The methods will take in and/or return objects of one of the specific derived *Item* classes, such as *Spell,* *Potion, Armor,* etc.
* **GateSceneDB** – Supports loading and saving gate information from the database. The methods will take in and/or return objects of the *GateScene* class.
* **GameSessionDB** – Supports loading and saving game sessions to the database. This is used to support saving the progress of a game, which we discuss later. The methods will take in and/or return objects of the *GameSession* class.
* **ScoreDB** – Supports loading and saving game scores from the database. The methods will take in and/or return objects of the *Score* class.

#### Loading Data

Since most game elements in Tucker Quest are stored in a database, all DB classes will have a data retrieval method. For the sake of standards, all database classes will have a method with the following signature to retrieve data:

def get(self, id)

Where *id* is the id for the element type, and the return type is an object of that class type.

For example, the *ItemDB* class has a *get(self, key)* method that takes in an item id, and returns an object of the appropriate class for that key (a *Sword* object, *Spell* object, *Treasure* object, etc). When the get method is called, at a high level, all the database classes follow the same:

1. Retrieve the JSON structure for a given id from the database using a function in the database engine (see section 6.7.8).
2. Convert JSON object string into a Python Dictionary, using the standard json library *loads()* function
3. Based on the type of the object, call the *from\_dict(dict)* static method on the corresponding class to generate an object based on the dictionary data. How to identify the object type is something we discuss in section 6.7.6 below.

Retrieving a single element by id is standard, and found in every class. Depending on need, some database classes might offer additional *query* methods to retrieve data based on different conditions. In those cases, the retrieval from the database (step 1 above) might be different, but the creation of class objects will still work the save

#### Saving Data

Not every database class needs to save data, since elements like Items or Enemies are not created in real time during the game play. They are configured ahead of time. However, some things like users and player characters might be created inside the game, and will need to be persisted back into the database. For those instances, the database class will provide a *save* method. When the *save* method is called, at a high level, all the database classes follow the same:

1. Use the *\_\_str\_\_()* method in the class object to generate a JSON string corresponding to the object
2. Save the JSON string for the object, with the object id as the key.

### Saving Game Progress

Saving a game progress is an important feature in any game, particularly those which include a long story. This is a complex task, because it requires the application to remember everything that took place in the game up to that point. Fortunately, the *GameSession* object maintains all of the relevant data throughout a session. So, saving the game, will essentially amount to saving the current *GameSession* object. When it’s time to start a game, if we want to reuse an existing session, we will simply load the session object for that player from the database, and set as the starting session for the game.

As we saw earlier, the following data is maintained in the *GameSession* object:

* The current player character being used
* The player location in the world
* The enemies defeated by the player
* Current active quest givers in the game
* A reference to the world (including all areas) being played
* A session id, which will be unique within a player character

One item that may seem odd to need saving at first glance, is the *World* object, because one might think that the world doesn't change. That may be true at a high level, however the world object contains the areas being played, and the areas include two key pieces data. First, the area map maintains the position in the world of game elements such as items and NPCs. Remember that when a player picks up an item, or interacts with an NPC, that element is removed from the map. So we need the latest up to date map, or upon restoring a session, all the items previously found, and NPCs encountered, would be back in the map. The other important thing is the list of visited cells. Remember that as we walk through the world, cells become visible. Therefore, we need to have that list up to date as well.

#### Active Session and Session History

An important thing to note, is that although there is a session id in our *GameSession* object, this will not be used as the key for the database. That's because we only maintain one active session per player character, so use the player character id as the key. This means each player character can only have one saved session. For this type of game, this is normal. One character, will only be in one adventure at the time. The session id exists so that we can keep a historical record of the scores of multiple sessions over time. So it will actually be used as the key for the *ScoreDB* class.

The *GameSessionDB* class has a delete method as well, so if a user wants to start a fresh session with that character, they will be able to, and the application code will delete the existing session from the database. And note that we are talking about the player character id, not the user id. That means a user could create multiple characters, and play separately with them. Each would have its own saved session.

### Identifying and Creating Object Types

This particular topic might be one of the most import ones when it comes to the ability of this application to be easily expanded by a programmer. That's because although a programmer can create a new class to represent a new item, or character type, how would the existing *ItemDB* or *PlayerCharacterDB* class be aware of that type? One solution would be for the programmer to also create separate DB classes for every new type. However, this would require a lot of additional new classes, as well as new database tables. The later in particular could be harder, because creating and maintaining database tables can be a significant effort. So ideally, we want to maintain a consistent set of database tables for each game element type. The solution will leverage the standard *from\_dict()* method we saw earlier, which we require every class that needs to be persisted to the database to provide, along with some additional configuration steps.

For game elements that are not meant to be extended with new types, such as the *World*, *Scene*, and *User* classes, the solution is simple. The *WorldDB* class knows it's creating a *World* object. The *GateSceneDB* class knows it's creating a *GateScene* object. So, they can directly call the *World.from\_dict()* and *GateScene.from\_dict()* standard methods respectively. However, when working with Items, Player Characters and NPCs, we wanted the application framework to support easy extension by programmers. Therefore, the database classes like *ItemDB, PlayerCharacterDB and NonPlayerCharacterDB* cannot have hardcoded class object names in there, since a programmer might create a new class type.

We discuss the framework for extending the application in detail in section 8, but for the purposes of this discussion, the key element is the use of a Dictionary in a configuration file that maps a class type to a *from\_dict()* static method in the class. These config files can be updated by the programmer, and we will discuss that process in detail later (section 8.1).

#### Example: Identifying and Creating an Object

Let's walk through one example. For instance, how would the *ItemDB* class create an appropriate *Potion* object after retrieving a JSON structure associated with a potion? Assume the application code makes the call *ItemDB.get("h0002"),* where "h0002" is the id of this Potion item. Bear in mind that the process we describe below is the same one used for identifying and creating NPCs and player characters in a generic fashion.

First, the following JSON structure would be retrieved by the database engine for the provided id.

{

"id": "h0002",

"name": "Medium healing potion",

"description": "a bottle labeled \"Healing Potion - Power 100\"",

"type": 400,

"power": 100,

"reusable": false

}

For every item type, we will need to map the type value to a corresponding *from\_dict*() static method, which knows how to create an object for that specific type from a dictionary representation of the JSON structure.

That mapping is stored in the in the *item\_from\_dict\_map* variable, located in the *item\_config.py* file. Here is what that looks like:

item\_from\_dict\_map = {

ItemType.TREASURE\_ITEM: Treasure.from\_dict,

ItemType.SWORD\_ITEM: Sword.from\_dict,

ItemType.SPELL\_ITEM: Spell.from\_dict,

ItemType.ARMOR\_ITEM: Armor.from\_dict,

ItemType.PROTECTION\_SPELL\_ITEM: ProtectionSpell.from\_dict,

ItemType.KEY\_ITEM: Key.from\_dict,

ItemType.POTION\_ITEM: Potion.from\_dict

}

# This method can create a specific item type from a dictionary

def item\_from\_dict(item\_dict):

# get reference to the from\_dict function for this item type

item\_type = item\_dict.get("type")

obj\_from\_dict = item\_from\_dict\_map[item\_type]

# create and return appropriate object from dictionary

return obj\_from\_dict(item\_dict)

The first part of the file has a simple map (Python dictionary) that can match an item type to a specific static method that knows how to create that type. The second part is a utility function *item\_from\_dict*, that can take a JSON, and invoke the correct static function based on the map. As we'll see below, this is what the DB class will use to dynamically create the correct item type.

You will notice that instead of the type number *400*, we are referring to "*ItemType.POTION\_ITEM*". That's simply a good programming practice. We could have used these type numbers across the application, and that would work. However, it's a good programming practice to map numerical constants to descriptive names. In the *game\_type\_config.py* configuration file, you'll find a mapping of types used across the application to nicer descriptive names. Here is the mapping in that file for the various item types:

class ItemType:

TREASURE\_ITEM = 1

SWORD\_ITEM = 100

SPELL\_ITEM = 101

ARMOR\_ITEM = 200

PROTECTION\_SPELL\_ITEM = 201

KEY\_ITEM = 300

POTION\_ITEM = 400

So now we have that *item\_from\_dict\_map* dictionary available, which we can use to generically pass in an item type, and retrieve a *from\_dict()* static method specific for that type.

The text box below shows a simplified version of the *ItemDB.get(id)* code (removed error checking and logging for easier readability). Here are the key points:

* **Line 1** - We import the *item\_from\_dict* from the *item\_config.py*module we just saw above
* **Line 5** – We retrieve the JSON structure for that id from the database, already converted into a Python dictionary by the database engine (we'll look at the database engine in the next section). Now we will have the *item\_dict* dictionary storing the data equivalent to the JSON structure.
* **Line 8** – We use the *item\_from\_dict* function we imported from *item\_config.py* to create the correct item type.
* **Line 11 –** We return the specific item object

1:

2:

3:

4:

5:

6:

7:

8:

9:

10:

11:

from item\_config import item\_from\_dict

def get(self, id):

# get item dictionary from database

item\_dict = ItemDB.tucker\_db.get(id)

# create appropriate item object from dictionary

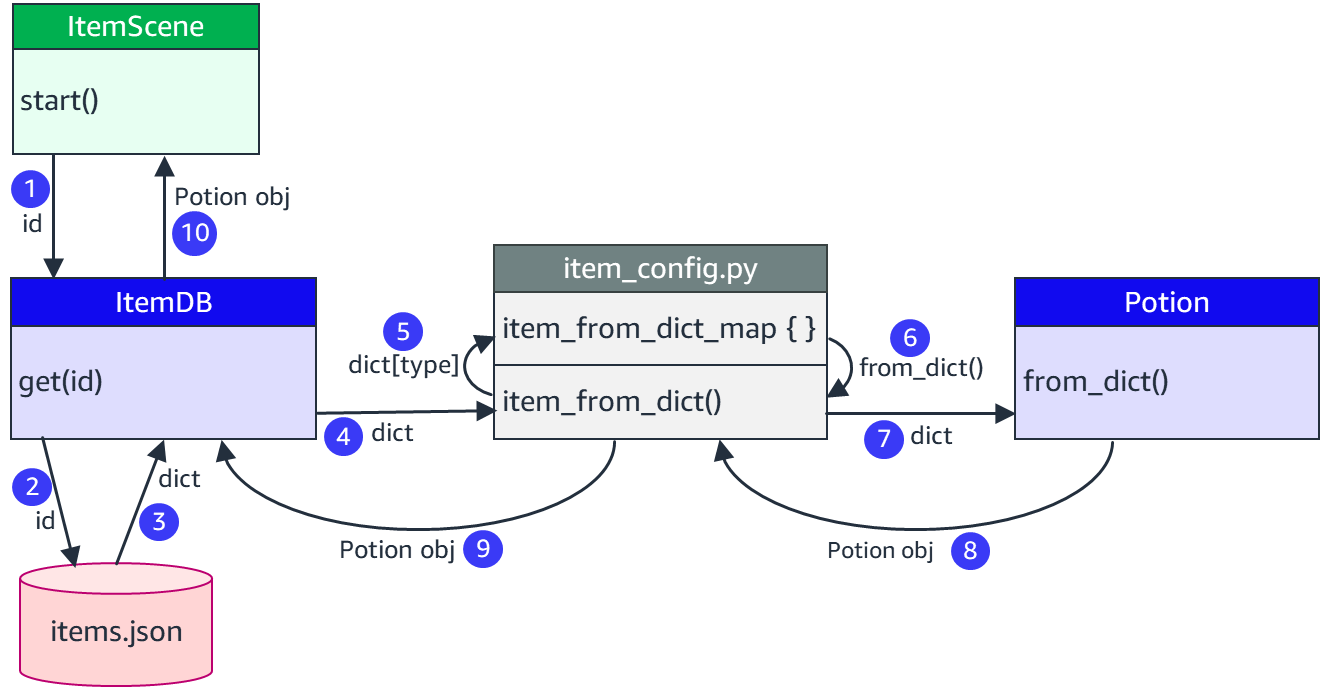
item\_obj = item\_from\_dict(item\_dict)

# return the appropriate object base on the item type

return item\_obj

The important part to note, is that *ItemDB* never referred to a specific item type such as *Treasure* or *Sword* or *Potion.* The only place where those specific classes were mentioned, was in the configuration files.

There were a lot of moving parts here, so the following diagram summarizes the end to end process:



1. Some part of the code, an *ItemScene* object for instance, needs to retrieve a *Potion* object based on the id, so it calls the *get* method on *ItemDB*
2. *ItemDB* passes the id to the database engine (currently *TuckerDB*) ...
3. ... and receives back a Python dictionary representing of the potion's JSON data
4. *ItemDB* then calls the *item\_from\_dict*() method in the *item\_config.py* module, passing the dictionary
5. *item\_from\_dict*() looks up the *item\_from\_dict\_map*, passing the type of the item as the key ...
6. ... and receives back a reference the to the *from\_dict*() static method for *Potion*
7. *item\_from\_dict*() then calls that method, passing the item JSON dictionary
8. The *item\_from\_dict*() method in *Potion* will know hoe to create a *Potion* object from the JSON dictionary, and returns that back.
9. *item\_from\_dict*() passes the *Potion* object back to *ItemDB* ...
10. ... which passes it back to the original caller, *ItemScene* in this example.

Once again, it bears repeating that this overall process is the same used to resolve items, player characters and NPCs.

### Database Engine

We've has a long discussion up to this point about the database layer, but haven't yet discussed which database engine we are using. That's because with the strong isolation principle we are following, where we are restricting all database access to the database classes, the actual database engine is not overly important from a coding perspective. We could completely change database engines, and the application code impact would be limited strictly to these database classes.

Please bear in mind that does not mean a database engine is not important. A good database engine is critical for performance, scalability and cost. These however are architectural topics, and our focus here is in the code. Within the AWS family of services, my choice for a database for Tucker Quest would have been AWS DynamoDB. DynamoDB is an enterprise level NoSql database, which supports the type of JSON storage we need here.

However, in keeping with the tenants discussed earlier in this document, this application had to be implemented such that it can be developed and executed in the standard *Development Fundamentals 2* practice environment. At this time, DynamoDB is not available, primarily because it will be a topic for *Development Fundamentals 3.* With that in mind, a custom database was created for the use of this application.

#### The TuckerDB Database

*TuckerDB* is a makeshift database created strictly to support this application. It's designed for simplicity, and not performance or scalability. We will not go over the design and coding of it in detail, because in a normal production application, we would be using a real database engine as I mentioned earlier.

*TuckerDB* stores its data in regular flat JSON files. In fact, what we call a "table" in *TuckerDB*, is really only a JSON file. When you create an instance of *TuckerDB*, you simply pass the full path to the JSON data file name in the constructor (the *\_\_init\_\_* method). So, for each of our database access classes, there is a corresponding JSON data file provided to *TuckerDB*. We will discuss these data files in the next section.

In most commercial databases, individual rows or items are saved and retrieved separately. *TuckerDB* will provide methods to get and save items individually, but that's largely an "illusion". In reality, for the sake of simplicity, *TuckerDB* saves all elements in a single JSON file, and loads/saves them all together. For instance, when *ItemDB()* creates a TuckerDB object to access the database, it will pass an "*item.json*" file as the JSON file name. All items in the game are saved into this one file. *TuckerDB* will read all the items in the game in one shot, and save it into internal memory. When *ItemDB* later retrieves a specific item from the *TuckerDB* object by passing a key, *TuckerDB* will simply return an item from its internal dictionary.

This is internal implementation of *TuckerDB*, where all the items are pre-loaded into memory, is not something the database classes need to be aware off. In fact, it's important that they don't, because if in the future we were to switch our database engine to something like DynamoDB, the individual items are in fact stored and retrieved one at a time. Once again, that's the principle of isolation in action.

### Tucker Quest Data Files

As we've just seen, working with TuckerDB our game data will be store in JSON data files. Therefore, when we want to add new elements to the game, we simply need to update them in the JSON data files. JSON is very readable format, so we can update these files with a regular text editor, and we just need to be careful not to break the proper JSON format.

If in the future Tucker Quest is changed to use a real database engine like DynamoDB, the same JSON structures would be saved in DynamoDB tables. Updating game contents would still be just a matter of updating JSON structures, since DynamoDB is very much JSON based.

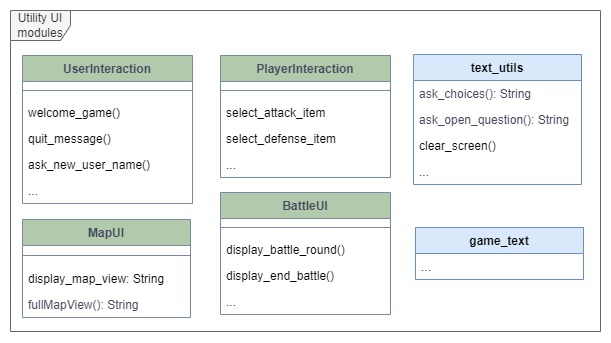
At this time, the following data files are used to store data for our various database classes. Each of these files are essentially the equivalent of a database table, and if we were to migrate to DynamoDB or another database engine in the future, we would likely have tables matching these files. All of these files are located under the *data* folder.

* **worlds.json** – stores the world/zone/map game data managed through the *WorldDB* class.
* **users.json** – stores all the user information data managed through the *UserDB* class
* **items.json** – stores all the game items managed through the *ItemDB* class.
* **player\_characters.json** – stores all the player characters managed through the *PlayerCharacterDB* class.
* **npcs.json** – stores all the non-player characters managed through the *NonPlayerCharacterDB* class.
* **gates.json** - stores all the gates managed through the *GateSceneDB* class.
* **game\_sessions.json** - stores the saved game sessions managed through the *GameSessionDB* class.
* **scores.json** - stores the game scores managed through the *ScoreDB* class.

The format of these files is what we've already shown in early sections when you provided examples of the "JSON Data Representation". In fact, the examples shown in those sections were copied directly from these files.

## User Interface Module

The User Interface module is not class hierarchy as we've seen in other areas. It's really a collection of classes, utility functions, and string constants that help in interfacing with the user. In our class diagram, they are shown as the auxiliary module below. The diagram only shows a very small number of operations as examples, but the actual classes and modules will have much more.



Why do we separate these components from the rest of the code? Why do we need a separate *MapUI* class to display a map, instead of simply putting that code into the existing *Map* and *Area* classes? The answer is once again the principle of isolation.

At this time, Tucker Quest is a text-based game. Input and output for a console-based text interfaces involve a lot of print statements and input prompts. Any type of graphical representation will be based on rudimentary ASC-II or Unicode based characters. This is something very specific, with its own features and limitations. However, in the future we could decide to implement Tucker Quest desktop-based application, which would then open the door to use a number of Python graphical libraries. Likewise, we could choose to deploy the game as a web application, in which case we could take advantage of yet another type of user interface capability. Regardless of our user interface capabilities, the essential framework of the game, which we've been discussing in all the previous sections, does not change. Therefore, by isolating the user interface functionality, we make it a lot simpler to change the presentation layer of this application in the future.

Most interactions in the game are handled through basic text input and output. There are couple of instances, such as displaying the game map and battle scenes, where a slightly more complex interaction is needed. These different types of interfaces are detailed in the next few subsections.

### Basic Text Interaction

#### UserInteraction and PlayerInteraction Classes

These classes encapsulate most of the interactions with users throughout the game, such as asking questions, and presenting text. All the methods in these classes are static, so that you don't need to explicitly create an object to invoke them. They are grouped under a class primarily to provide a clean name space.

The methods in this class can be as simple as *print\_item\_found()*, which prints a message when an item is found, or *ask\_new\_user\_name()* which asks for the name of a new user. Many of them will take arguments which are included in the user interaction. For instance, when we are printing that an item was found as mentioned earlier, we would pass in a description of the item (taken from the *Item* object), and the description would be incorporated into the message to the user.

#### The text\_utils module

This file has a collection of functions to perform common actions for a text-based interface. For instance, it will have functions to clear a console screen, present a numbered list of options, ask for simple text input, etc. These are generally used by the *UserInteraction* and *PlayerInteraction* classes above.

#### The game\_text file

This is a simple file, but it represents yet another important aspect of isolation. The file is simply a list of constant strings for nearly every message shown in the display during game play. Nearly every interaction driven by the user interface classes above refers to string constants in *game\_text*. Some of them will be 100% static text, while others will have "{}" place holders, which can be filled in using standard Python format strings. He is an example of a couple of the constants found in that file:

...

ASK\_NEW\_USER\_LNAME = \

"Please enter your last name: "

ITEM\_FOUND = \

"You look down, and you see something on the ground.\n"\

"It's {}."

...

There are two good reasons to isolate these string constants into a separate file, and that's a common technique in large applications:

* In many cases, the resources responsible for written conversational messages are not IT resources. The fact is, IT skills and language skills are very different things. Someone may be an excellent developer, but not necessarily a good writer. Therefore, this separation allows for the "messaging SME" to prepare the messages, separate from the development code.
* Natural language support. If an application needs to be ported to a different language, it's very important that all language interactions are isolated in one location, so that human or machine translators can operate outside of application code.

Bear in mind that in the "real world", there are libraries and frameworks to support the isolation described here. Nonetheless, the principle and the motivation are the same.

### The BattleUI Class

The *BattleUI* class handles the interactions that play out during a Battle scene in the game. A battle is probably the most active seen in a game, and involves multiple attacks and defenses by players and enemies. The class tries it's best to show the progress of battle, given the limitations of a text interface.

### The MapUI Class

The MapUI class is the most complex of the UI elements. As we recall from section 6.2.3.2, the following is an example of the data structure representing an area map:

"map":[

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","G0001","WWWWW"," "," "," ","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW"," "," ","E0201","WWWWW"," ","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW"," ","BBBBB"],

["BBBBB","t0001"," "," ","m0003"," "," ","s0003"," ","BBBBB"],

["BBBBB","WWWWW","WWWWW"," ","WWWWW","WWWWW","WWWWW","WWWWW","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW"," "," ","p0002"," ","h0002","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW","a0002","WWWWW","WWWWW","WWWWW"," ","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW"," ","N0301","E0101","WWWWW","BBBBB"],

["BBBBB","WWWWW","WWWWW","WWWWW","WWWWW","SSSSS","WWWWW","WWWWW","WWWWW","BBBBB"],

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"]

]

This is reasonable for a developer creating a map, but certainly not a visually pleasing or easy to understand. The MapUI class will turn that JSON data into something like this:

gate symbol

potion symbol

player symbol

sword symbol

enemy symbol

▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓

▓▓▓████████████████████████▓▓▓

▓▓▓█████████ Ώ ███ . . . ▓▓▓

▓▓▓█████████ . . Ӝ ███ . ▓▓▓

▓▓▓█████████████████████ . ▓▓▓

▓▓▓ $ . . Ф . . † . ▓▓▓

▓▓▓██████ . ███████████████▓▓▓

▓▓▓██████ . . § . Ϫ ███▓▓▓

▓▓▓██████ Ѡ █████████ . ███▓▓▓

▓▓▓████████████ . Ÿ Ӝ ███▓▓▓

▓▓▓████████████ ▲ █████████▓▓▓

▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓

Some of the symbols representing different game elements are annotaded in the example above, but there are many others. We'll see how these symbols are created next.

#### Map Display Characters

Most of the symbols seen above, including the walls, are represented using Unicode characters. Unicode characters are an extension of the standard ASC-II character set, and are supported by most operating systems. They can be used to represent character sets in different languages, block symbols, and a variety of "fancier" characters including emojis. However, not every OS and terminal application supports all the same Unicode characters. There are some more sophisticated characters we could have used in Tucker Quest, but upon testing, some of them were not supported in all terminals. The ones currently used appear to work properly on both the Cloud9 bash terminal, as well as Windows based command and PowerShell.

The choice of symbols to use was mostly personal and arbitrary, and they can be easily change. The configuration file *map\_config.py* has definitions for what character to use for different cell types. For example, in that file you'll see these definitions among many others:

...

GATE\_BLOCK\_STR = " " + u"\u038f" + " "

POTION\_BLOCK\_STR = " " + u"\u03ea" + " "

...

If you do an internet search for the Unicode symbol "038f", you'll find that it represents the Greek Omega Symbol, **Ώ**, which is what I used for gates. If you look up the Unicode symbol "03ea", you will see that represents the Coptic Capital Letter Gangia , **Ϫ**, which is what I used for potions. As you might guess, the fact that this is a symbol in the Coptic Alphabet is completely irrelevant. I simply choose the symbol based on appearance, so anyone is welcome to choose different symbols and update the value in the *map\_config.py* file.

#### The Map in Game

If you have tried out the game, which is highly recommended for anyone programming with it, you will have noticed that you do not actually see the fully drawn map as represented above. That text was actually copied from the log files, where we can display the full map for debugging purposes. During in game interactions, players only see cells that are immediately surrounding them, or cells that have already been seen. For instance, this is an example of a view of the same map above, but taken from an actual in game output.

▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓

▓▓▓░░░░░░░░░░░░░░░░░░░░░░░░▓▓▓

▓▓▓░░░░░░░░░░░░░░░░░░░░░░░░▓▓▓

▓▓▓░░░░░░░░░░░░░░░░░░░░░░░░▓▓▓

▓▓▓████████████░░░░░░░░░░░░▓▓▓

▓▓▓ $ . Ÿ Ф ░░░░░░░░░░░░▓▓▓

▓▓▓██████ . ███████████████▓▓▓

▓▓▓░░░███ . . . . . ███▓▓▓

▓▓▓░░░███ Ѡ █████████ . ███▓▓▓

▓▓▓░░░░░░░░░███ . . . ███▓▓▓

▓▓▓░░░░░░░░░███ ▲ █████████▓▓▓

▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓

This is done primarily to add an element of "mystery and unknown" to the game. We want the player not to know everything that it's coming up. To implement this the *Area* class maintains a parallel grid that tracks which cells in the map have already been seen. That's the *visited*attribute, which was mentioned when we introduced the *Area* class earlier. The *MapUI* class will use that field to determine whether to show a regular cell symbol, or the default light grey block representing an unknown cell.

# Standard Game Customization

In the section 8 we will discuss the programmatic way by which the game can be expanded. This will include some level of coding, even if the framework makes these changes accessible. It is important to note however that even without programming, this application is meant to be easily expanded by simple updates to the application data. We many of the earlier sections we described the JSON structure for the various game elements, and in section 6.7.9 we describe the data files where these JSON structures are stored. Therefore, new items or characters of existing types can be added simply by updating these files with new elements. We will look at some examples below.

## Adding a New Item

The JSON data for items is stored in the current *TuckerDB* database in the *items.json* file. Therefore, if we want to add an item of an existing type to the game, we just need to add the JSON for that item in the file. For example, let's say we wanted to add a new "extra strong" sword to the game, called "Diamond Sword". We would just simply add the following few lines to the *items.json* file, with the appropriate attribute values of our choice:

{

... other items ...

"s0202": {

"id": "s0202",

"name": "Diamond sword",

"description": "a shiny and strong diamond sword",

"type": 100,

"min\_power": 120,

"max\_power": 150

}

... other items ...

}

Obviously, we would need to specify the attributes that the particular item requires. However, this is easily done by simply copying an existing item of the same type. We need to make sure that we use a unique id. The current id scheme supports 10,000 possible unique ids per item type, so finding a new unique id should not be a problem.

Finally, if we want this item to really be in the game, we would have to add this item id somewhere in the game map. We discuss the map updates shortly.

## Adding a New Non-Player Character

The JSON data for non-player characters is stored in the current *TuckerDB* database in the *npcs.json* file. Therefore, if we want to add an NPC of an existing type to the game, we just need to add the JSON for that character in the file. For example, let's say we wanted to add a new beast enemy to the game, called "Giant Black Spider". We would just simply add the following few lines to the *npcs.json* file, with the appropriate attribute values of our choice:

{

... other non-player characters ...

"E4387": {

"id": "E4387",

"name": "Giant Black Spider",

"pronoun": "it",

"type": 3,

"status": null,

"description": "a large black spider",

"level": 1,

"max\_health": 100,

"health": null,

"attack\_item\_id": null,

"defense\_item\_id": "a1002",

"inventory": null,

"message": "You fell into my web"

},

... other non-player characters ...

}

Obviously, we would need to specify the attributes that the particular NPC type requires. However, this is easily done by simply copying an existing item of the same type. We need to make sure that we use a unique id. The current id scheme supports 10,000 possible NPC ids per NPC type, so finding a new unique id should not be a problem.

Finally, if we want this NPC to really be in the game, we would have to add this id somewhere in the game map. We discuss the map updates in the next section.

## Updating the World, and Area Maps

We discussed the composition of the world, zones, areas and maps in detail in section 6.2. Once again, updating the world to add areas, or simply updating an existing an area map, is just a matter of updating the JSON specification, which is found in the *worlds.json* file.

Note that we're not discussing adding a new zone, because that's very simple. As we discussed earlier, a zone is more of a logical division of the world, like a continent. It just has a description, and a list of areas. That can be easily updated in the *worlds.json* file.

### Adding a new Area to the World

Adding a new area to the world would involve adding the basic area attributes, but most important, designing a map for that area. Here is an example of adding a very simple new area, as small chamber, with a 6 x 6 map.

{

... other areas ...

"a0505": {

"id": "a0503",

"name": "Dungeon – small chamber",

"description": "This is a small chamber located in the bottle of the temple. The only way in and out seems to be the one you came in on",

"zone\_id": "z0100",

"map": [

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

["BBBBB"," "," "," "," ","BBBBB"],

["BBBBB"," "," "," "," ","BBBBB"],

["BBBBB","WWWWW"," "," ","WWWWW","BBBBB"],

["BBBBB","WWWWW","G0105","WWWWW","WWWWW","BBBBB"],

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

],

"visited": null

}

... other areas ...

}

A couple of key things to note:

* Entering/exiting a new area will always require a gate, so note that we have a gate id ("G0105") in there. The gate itself needs to be similarly defined in the *gates.json* file, which we are not showing here.
* Don't forget the important rule describe in section 6.2.3.2, that any game map must always be fully surrounded by border cells ("BBBBB" cells).

## Adding Game Elements to the World

In the previous sections we showed examples of creating a new item (the Diamond Sword) and a new Beast enemy (the Giant Black Spider). In order to add these elements to the world, we would simply add their ids to the map where we want the player to encounter them. Using the area that we just created above, we could add them as seen below (you can match the ids with the ones used in the previous examples).

{

... other areas ...

"a0505": {

"id": "a0503",

"name": "Dungeon – small chamber",

"description": "This is a small chamber located in the bottle of the temple. The only way in and out seems to be the one you came in on",

"zone\_id": "z0100",

"map": [

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

["BBBBB"," "," "," ","E4387","BBBBB"],

["BBBBB","s0202"," "," "," ","BBBBB"],

["BBBBB","WWWWW"," "," ","WWWWW","BBBBB"],

["BBBBB","WWWWW","G0105","WWWWW","WWWWW","BBBBB"],

["BBBBB","BBBBB","BBBBB","BBBBB","BBBBB","BBBBB"],

],

"visited": null

}

... other areas ...

}

Placing the id of an element in a specific map cell, we are telling the game engine that the player will encounter that element (an item, an NPC, a gate, etc) when he/she reaches that cell.

# Custom Application Extension

Customizing the application through data files is a nice feature, but it does not exercise our super powers as developers. As discussed earlier, one of the key tenants of this application was to allow learners to extend it as much as possible. That means that beyond simply creating new elements of existing types, we want to allow programmers to create new items and characters with their own unique behaviors. This means writing new Python classes, but still being able to leverage the basic application framework. Much of the design of this application was driven by this goal. We've already discussed how some of this customization works when discussing the database layer, but in this section we will look at the process end to end, discuss the key components, and provide specific examples.

The three main things programmers can add to the game are new item types, new player character types, and new non-player characters types. This involve two main things:

* Creating the new classes that define what is being added, and the code to implement the new behavior
* Incorporating that new classes to the application framework, by updating key configuration files

## Configuration Files

The application has multiple files that will define type constants, and mappings between element types. All of them are located under the *config* folder. Updating these configurations will allow programmers to "wire" their new custom classes to the application framework. We'll look at each file next.

### The game\_types\_config.py File

This is probably the most important configuration file in the game, as it defines the names constants for the various elements of the game. Many programming languages have "enumerated types" to define numerical constants with a specific name space. Python doesn't directly support that, so an equivalent mechanism is to define a class, and the numerical constants as class variables. Using that technique, the file defines the following groups of constants:

* **ItemType** – Defines all the valid item types in the game, as well as item categories
* **NPCType** – Defines all the valid non-player character types in the game
* **PlayerType** – Defines all the valid player types in the game
* **SceneType** – Defines all the valid scene types in the game

### The item\_config.py File

This purpose of this file was explained in low level details in section 6.7.6, when we discussed how the data layer was able to generically create different object types. To quickly recap, this file maintains a map (using a Python dictionary) of the different item types, to the static methods used to create them.

### The non\_player\_character\_config.py File

Very similar to the *item\_config.py* file, this file maintains a map (using a Python dictionary) of the different NPC types, to the static methods used to create them.

### The player\_character\_config.py File

Likewise, this file maintains a map (using a Python dictionary) of the different player character types, to the static methods used to create them.

### The map\_config.py File

The *map\_config.py* file maintains configuration values that control the display of game elements in the screen map, and how they map to different item, scene and character types. You will need to update this file if you want to change the screen representation of one of these elements, or if you add a new item type.

## Adding a New Item Type

Assume the programmer wants to add a new type of item to the game. For the sake of this example, let’s say we want to add a new type of attack item: an axe. The idea would be that perhaps this new “axe” weapon would have different special behaviors during an attack. For instance, assume an axe is very powerful, but because it's hard to maneuver, you can only use it every other turn. You would need some custom code to implement that. In the next sub-sections, we will see the different steps to be taken to add this new item to our game.

### Define Item Attributes and JSON Format

You start by deciding what attributes will define your item. Remember that your item will be injected into the *Item* class hierarchy, so at a minimum it will inherit attributes from *Item*. Depending on the item you are adding, it may also be inheriting attributes and methods from a sub-class. For instance, for our axe example, our item would inherit from the *AttackItem* class.

You will add this JSON definition to the database. In our current implementation, that means we will add it to the *items.json* file. For example, for our axe item, the JSON format could look like:

"x0001": {

"id": "x0001",

"name": "Heavy axe",

"description": "a powerful axe, but hard to lift",

"type": 105,

"min\_power": 120,

"max\_power": 180,

"attack\_frequency": 2

}

The format above looks largely like the one used in the *Sword* item se saw in section 6.3.3, but we simply added a new “*attack\_frequency*” attribute. This is what we would use for our new behavior. The attribute defines how often the item can generate an attack. So our heavy axe can only be used once every 2 attacks.

As you define the new type and attributes, remember you will need to pick a type value and an id prefix that have not been used yet. You can find the ones that have already been used in the configuration files you'll need to update (we'll see that later).

### Create Item Specific Classes

This is where your personal coding comes in. We need to implement a class for your new Item, with the following minimum requirements.

* Create the class with a descriptive name, in its own file. For our example, this would be an *Axe* class, in a *axe.py* file
* The class must be derived from *Item* Class, or an appropriate sub-class. In our example, the *Axe* class is meant to be an attack item, so it would derive from *AttackItem* which is a sub-class of *Item.*
* The class *\_\_init\_\_()* method must set the Item type attribute, using the type value defined in the *game\_types\_config.py* file. In our example, it would set the item type to *AXE\_ITEM.*
* It may also need to set the item category, unless it derives from a sub-class of *Item*, and the sub-class already specifies it. For instance, in the case of our *Axe,* the *AttackItem* constructor already sets the category to *ATTACK\_ITEM.*
* It must include a *from\_dict()* static method, that knows how to create an object of the class from a Python dictionary that corresponds to the JSON. Without this, the *ItemDB* class would not be able to retrieve our new item type.
* It must include a *\_\_str\_\_()* method that generates a JSON string for the object data

This is the minimum you need to make the class work, but if you want to add some special behaviors, then you just need to override a method in the base class. For example, to give your *Axe*  its own special way of generating an attack power, you would override the *generate\_attack()* method, which is defined in the parent class *AttackItem.* We're not going to show the code for the class here, since that would be completely dependent on the type of behavior you are looking to add. However, if you want to see an example of a possible *Axe* implementation, I've included the file in Appendix A .

### Config file changes

After your class is written, there are a few configuration files you need to update to “wire” your item to the game framework. Let’s look at the updates required in each file. We will not show code samples here, because for all of those, the easiest thing to do is to look at the format of existing items, and copy it.

**game\_types\_config.py**

* Add a unique type value for your item under *ItemType* class. For our example, you would add an *AXE\_ITEM* entry with a value of 105 (this what we used in our JSON structure).

**item\_config.py**

* Import your new class file on top. For our example we would import the *Axe* class from the *axe.py* file.
* Add mapping in the *item\_from\_dict\_map* to the *from\_dict()* static method you created for your item. You can see examples in the file.

**player\_character\_config.py**

* Add item to *character\_can\_use\_map* to indicate if it’s usable for each player character type. For example, in our existing player characters, an axe would likely be usable by a *Knight*, but not a *Wizard*. If we don't do that, we would be able to add our item to the world, but players would not be able to use it.

**map\_config.py**

* Add a unique cell type value for your item. For our example, we would use *AXE\_CELL* as the constant name.
* Create a block string to control how your item is shown in map. You can copy an existing one if you want, or look for a Unicode character that resembles your item.
* In CELL\_TYPE\_TO\_STR\_MAP, add a mapping between the cell type and the cell type block string you created above.
* In CELL\_TYPE\_MAP, add a mapping of the first character of the item id to a cell type. As always, you can see examples from the existing types in the file.
* In CELL\_TYPE\_TO\_SCENE\_MAP, add a mapping from the new item cell type to a scene type. See examples in the file.

For our axe, unless you are doing something extra special, the scene type would be ITEM\_SCENE. We're just picking up the item. Later on, that axe would be available in the player's inventory to use in a battle.

### Testing the New Item Type

After changing the code, make sure you test it in any way your new item might be used.

#### Add an Item of the New Type in the Database

Remember that the code changes only create the code to support that new item. The item will only exist in the game, when you create an instance of it in the database. As we discussed extensively in section 7.1, this will simply be a matter of adding an instance of that JSON structure we defined in the *items.json* file.

#### Add an Instance of the New Item to the World

Now we just need to add an instance of our new item to the world. As we discussed in section 7.4, we will just need to add the id for the item to a map location defined in *worlds.json*. For the purpose of testing, it would be good to initially add the item very early in the map, that way you can immediately see if there were any coding problems.

#### Play the Game with the New Item

There is no better way to test your new item than playing with it. Remember to test not only that it can get picked up, and displayed properly, but also that it can be used. In the case of our axe, it would critical to test it in a battle, and verify that our custom attack actually works.

## Adding a New Player Character Type

Assume the programmer wants to add a new type of player character to the game. For the sake of this example, let's use the most cliché of fantasy creatures: an elf. We want to give our elf some kind of special characteristic so we can test. This could be as simple as the ability of using a new type of attack or defense item. However, to force some actual code changes, we will say that because our elf is a smaller creature, it's attack power will be reduced by 25%. However, because it is faster, its defense power will be increased by 25%, because it can dodge better. In terms of coding, this means our new *Elf* class we will have the override the *attack* and *defend* methods which are implemented in the base class *Fighter.*

### Define Player Character Attributes and JSON Format

Again, we start by deciding what attributes will define the new character. Remember that your character will be injected into the *PlayerCharacter* class hierarchy, which in turn inherits from the *Fighter* class, so at a minimum it will inherit attributes from both of these classes.

You will add this JSON definition to the database. In our current implementation, that means we will add it to the *player\_characters.json* file. For example, for our elf player, the JSON format could look like:

"fttg.Fernlorian": {

"id": "fttg.Fernlorian",

"name": " Fernlorian",

"pronoun": "he",

"type": 3,

"status": null,

"description": null,

"level": 1,

"max\_health": 1000,

"health": null,

"attack\_item": null,

"defense\_item": null,

"inventory": null,

"attack\_loss": 25,

"defense\_gain: 25

}

The format above looks largely like the one used in the *Knight* we saw in section 6.4.2.2, but we simply added two new attributes for “*attack\_loss*” and "*defense\_gain*". This is what we would use for our new behavior to add and reduce attack and defense power (we don't have to always use 25%).

As you define the new type and attributes, remember you will need to pick a type value that has not been used yet. You can find the ones that have already been used in the configurations files you'll need to update (we'll see that later).

### Create Player Specific Classes

This is where your personal coding comes in. We need to implement a class for your new player character, with the following minimum requirements.

* Create the class with a descriptive name, in its own file. For our example, this would be an *Elf* class, in a *elf.py* file
* The class must be derived from *PlayerCharacter* Class, or an appropriate sub-class
* The class *\_\_init\_\_()* method must set the *PlayerCharacter* type attribute, using the type value defined in the *game\_types\_config.py* file.
* It must include a *from\_dict()* static method, that knows how to create an object of the class from a Python dictionary that corresponds to the JSON.
* It must include a *\_\_str\_\_()* method that generates a JSON string for the object data

This is the minimum you need to make the class work, but if you want to add some special behaviors, then you just need to override a method in the base class. As we discussed, in our example we will override the *attack* and *defense* methods to modify the power generated.We're not going to show the code for the class here, since that would be completely dependent on the type of behavior you are looking to add. However, if you want to see an example of a possible *Elf* implementation, I've included the file in Appendix A .

### Config file changes

After your class is written, there are a few configuration files you need to update to “wire” your player character to the game framework. We will not show code samples here, because for all of those, the easiest thing to do is to look at the format of existing items, and copy it.

**game\_types\_config.py**

* Add a unique type value for your player character under *PlayerType* class. For our example, you would add an *ELF* entry with a value of 3 (this what we used in our JSON structure).

**player\_character\_config.py**

* Import your new class file on top. For our example we would import the *Elf* class from the *elf.py* file.
* Add mapping in the *player\_from\_dict\_map* to the *from\_dict()* static method you created for your player. You can see examples in the file.
* Add mapping in the *player\_object\_map* to an empty newly created object of your new class. You can see examples in the file. This is something you need to provide for player characters that you do not need to do for items or NPCs. That's because player characters aren't always existing ahead of time. They might be newly created by users during the game, so we need empty versions to use in those cases.
* Add an entry for your new player type in the *character\_can\_use\_map* to indicate what item types it can use.
* Add an entry for your new player type in the *player\_character\_to\_str\_map* to provide a short name that describes your player type.

### Testing the New Player Type

After changing the code, make sure you test it in any way your player type might be used. Testing a new player type does not require additional data setup as a new item did. That's because new player characters are created in game, by code already existing in *GameController*. Provide that you configured the files properly, the new player type should fit in with the existing code.

So just go ahead and play the game, creating a character of the new type, and verifying that it works properly. In the case of our elf, it would critical to test it in a battle, and verify that our custom attack actually works. The visual display will not show the modified attack numbers, so make sure you add some log statements that you can inspect to confirm the result.

## Adding a New Non-Player Character Type

Assume the programmer wants to add a new type of non-player character to the game. For the sake of this example, let’s say we want to add a new type of NPC that called a thief. As the name implies, a thief is a character that will rob the player. Specifically, a thief will steal one random item from the player's inventory. In the next sub-sections, we will see the different steps to be taken to create this new NPC type

### Define NPC Attributes and JSON Format

You start by deciding what attributes will define your new NPC. Remember that your NPC will be injected into the *NonPlayerCharacter* class hierarchy, so at a minimum it will inherit attributes from it. Depending on the NPC you are adding, it may also be inheriting attributes and methods from a sub-class. For instance, if we were adding an enemy, the class would also inherit attributes from the *Enemy* class. Our thief however will not be taking part in battles, so it will inherit directly from *NonPlayerCharacter*.

You will add this JSON definition to the database. In our current implementation, that means we will add it to the *npcs.json* file. For example, for our *Thief* character, the JSON format could look like:

"T0101": {

"id": "T0101",

"name": "Sneaky Jane",

"pronoun": "she",

"type": 300,

"status": null,

"description": "a young lady, with mischievous look in her eyes",

"inventory": null,

"message": "Hi there adventurer. What's that behind you?"

}

The format above looks exactly like the one used in the *StoryCharacter* NPC we saw in section 6.5.2. The difference is not in the data, but the special behavior we will code in the class. Other new NPCs might have additional attributes. It's all up to the developer.

As you define the new type and attributes, remember you will need to pick a type value and an id prefix that have not been used yet. You can find the ones that have already been used in the configurations files you'll need to update (we'll see that later).

### Create NPC Specific Classes

This is where your personal coding comes in. We need to implement a class for your new NPC, with the following minimum requirements.

* Create the class with a descriptive name, in its own file. For our example, this would be a *Thief* class, in a *thief.py* file
* The class must be derived from *NonPlayerCharacter* Class, or an appropriate sub-class
* The class *\_\_init\_\_()* method must set the *NonPlayerCharacter* type attribute, using the type value defined in the *game\_types\_config.py* file.
* It must include a *from\_dict()* static method, that knows how to create an object of the class from a Python dictionary that corresponds to the JSON.
* It must include a *\_\_str\_\_()* method that generates a JSON string for the object data

This is the minimum you need to make the class work, but if you want to add some special behaviors, then you just need to override a method in the base class. For example, to implement the special functionality of our *Thief* class, we would need to override the *post\_action()* method. The method would retrieve the player from the current game session, and steal and random item from their inventory. If you want to see an example of a possible *Thief* implementation, I've included the file in Appendix A .

### Config file changes

After your class is written, there are a few configuration files you need to update to “wire” your new NPC to the game framework. Let’s look at the updates required in each file. We will not show code samples here, because for all of those, the easiest thing to do is to look at the format of existing items, and copy it.

**game\_types\_config.py**

* Add a unique type value for your NPC under *NPCType* class. For our example, you would add a *THIEF* entry with a value of 300 (this what we used in our JSON structure).

**non\_player\_chracter\_config.py**

* Import your new class file on top. For our example we would import the *Thief* class from the *thief.py* file.
* Add mapping in the *npc\_from\_dict\_map* to the *from\_dict()* static method you created for your NPC. You can see examples in the file.

**map\_config.py**

* Add a unique cell type value for your new NPC. For our example, we would use *THIEF\_CELL* as the constant name.
* Create a block string to control how your NPC is shown in map. You can copy an existing one if you want, or look for a Unicode character that resembles your NPC.
* In CELL\_TYPE\_TO\_STR\_MAP, add a mapping between the cell type and the cell type block string you created above.
* In PROXIMITY\_CELL\_TYPES, add the cell type for your new NPC. This list identifies cell types which initiate an action when the player is next to them in the map, as opposed to on top (like items). NPCs fall under that category, so we need to add the new type here.
* In CELL\_TYPE\_MAP, add a mapping of the first character of the item id to a cell type. As always, you can see examples from the existing types in the file.
* In CELL\_TYPE\_TO\_SCENE\_MAP, add a mapping from the new cell type to a scene type. See examples in the file. For our thief, unless you are doing something extra special, the scene type would be ENCOUNTER\_SCENE. The regular encounter scene logic automatically invokes the *pre\_action* and *post\_action* methods on any NPC encounter.

### Testing the New NPC Type

After changing the code, make sure you test it in any way your new non-player character might interact in the game.

#### Add an NPC of the New Type in the Database

Remember that the code changes only create the code to support that new NPC type. The NPC will only exist in the game, when you create an instance of it in the database. As we discussed extensively in section 7.1, this will simply be a matter of adding an instance of that JSON structure we defined above in the *npcs.json* file.

#### Add an Instance of the New NPC to the World

Now we just need to add an instance of our new NPC to the world. As we discussed in section 7.4, we will just need to add the id for the NPC to a map location defined in *worlds.json*. For the purpose of testing, it would be good to initially add the NPC early in the map, that way you can immediately see if there were any coding problems. Remember thought, that to properly test our thief, our player should have at least one item in their inventory.

#### Play the Game with the New NPC

There is no better way to test your new NPC than playing with it. Remember to also test error scenarios. In the case of our Thief, what would happen if the thief tries to steal an item from a player that has nothing in their inventory?

## Specialized Extensions

Not every extension has to be a very general new item, player character, or NPC. Sometimes we may just want to have a one-off type of resource, with very special characteristics. To do that, one can still use a combination of the same strategies used in the sections above. They key object-oriented techniques will still be in the ability to derive from certain classes, and then override specific methods. You will also likely need to use the configuration files to "wire" your creation into the game. As an example, we will look at combination enemy and item created by one of our learners.

### Example: Evil Sherri

Sherri, one of our learners, envisioned a new enemy type: Evil Sherri. Evil Sherri has great power, and the only way a player might be able to survive her attacks and inflict any damage, is if that player possesses Holy Water. This idea adds a dependency between an NPC, and specific item type. That's a different twist on the customizations we discussed earlier, but can be done using a combination of the same techniques. We'll breakdown each individual step.

#### The New Holy Water Item

Holy Water will be a new type, so the technique fill follow generally steps as we saw in Section 8.2. As designed by Sherri, the Holy Water was not an attack item. It was meant to be a type of potion the player would possess, which would provide protection against the extra power of the enemy "Evil Sherri".

##### JSON Structure

We added this JSON definition to the database. In our current implementation, that means we will add it to the *items.json* file. For example, the JSON format looks like:

"h0004": {

"id": "h0004",

"name": "Holy Water of Power",

"description": "a sacred holy potion that grants immense power",

"type": 401,

"power": 150,

"reusable": false

}

Note that unlike what we did earlier, where we defined a new prefix for the new item id, in this case we are still using the same "h" that is used for the regular potion class. That is fine, because for this special case, we didn't need any specialized behavior in the map. The Holy Water will look like a regular potion. But it does have a different type value (401 as seen above), and that difference is important for specialization later.

##### HolyWater Class

The *HolyWater* class is derived from the *Potion* Class, and doesn't need to specialize any methods. However, its constructor (the \_\_init\_\_ method) will set the type of the item to a new HOLY\_WATER\_ITEM type. So, although it generally behaves like a potion, the different type makes it distinguishable from the regular potions.

##### Configuration Files

The changes here are the ones we described in Section 8.2.3, but we may not need all of them for this case.

**game\_types\_config.py**

* Added HOLY\_WATER\_TYPE = 401 in the *ItemType* class.

**item\_config.py**

* Add mapping in the *item\_from\_dict\_map* to the *from\_dict()* static method of the *HolyWater* class.

**player\_character\_config.py**

* Added the new HOLY\_WATER\_ITEM type to *character\_can\_use\_map* to indicate if it’s usable for every player character type.

**map\_config.py**

* This is where we would normally make all the configurations related to how an item would present in the map. However, since we wanted the Holy Water to look like a standard potion (which is why we re-used the existing prefix "h"), no changes are needed.

#### The new Evil Sherri Enemy

This one is a new NPC type, so the changes largely follow the example provided in Section 8.4.

##### JSON Structure

We added this JSON definition to the database. In our current implementation, that means we will add it to the *npcs.json* file. For example, the JSON format looks like:

"E0103": {

"id": "E0103",

"name": "Evil Sherri",

"pronoun": "she",

"type": 300,

"status": null,

"description": "a menacing sorceress with a sinister smile",

"level": 5,

"max\_health": 200,

"health": null,

"attack\_item\_id": "s0004",

"defense\_item\_id": "a0004",

"inventory": null,

"message": "Your journey ends here, mortal.\nAnd it will be swift if you are not carrying any holy water.\nFace the wrath of Evil Sherri!"

}

The format above looks exactly like the one used in the *HumanEnemy* NPC we saw in section 6.5.2. The difference is not in the data, but the special behavior we will code in the class.

##### EvilSherri Class

The basic functions of the *EvilSherri* class were much the same as the *HumanEnemy* class, so the standard *\_\_init\_\_, \_\_str\_\_* and *from\_dict* methods were largely copied from there. The special behavior for *EvilSherri*, and what required a new class, was in overwriting the standard *attack* and *defense* methods of the *Fighter* base class. The default base class methods, which is used by most enemy classes and player character classes, will generate attack power and defense powers based on the attack items and defense items equipped by the characters. This is fine in most cases. However, as designed by Sherri, the "Evil Sherri" character has unlimited power ... unless the player is holding holy water. To implement this functionality, we do the following:

* The *EvilSherri* class has a utility method *player\_has\_holy\_water*() that will check the current player's inventory in the game session, and return True or False, depending on whether the player has holy water or not. This is why it's important for *HolyWater* to have its own item type. Having any old potion won't protect the player, so the method needs to check for that specific HOLY\_WATER\_ITEM type.
* The *attack*  method is overwritten so that it first uses *player\_has\_holy\_water*()to check if the player has holy water. If they don't, the method will generate a very large attack that will guarantee an immediate death to the player. If the player has holy water, the method simply calls the standard *attack* method in the base class which will work as usual.
* The *defense*  method is overwritten so that it first uses *player\_has\_holy\_water*()to check if the player has holy water. If they don't, the method will generate a defense that will completely neutralize the player's attack. If the player has holy water, the method simply calls the standard *defense* method in the base class which will work as usual.

##### Configuration Files

The changes here are the ones we described in Section 8.4.3, but we may not need all of them for this case.

**game\_types\_config.py**

* Added the NPCType EVIL\_SHERRI

**non\_player\_chracter\_config.py**

* Added the mapping to the *from\_dict()* static method for *EvilSherri*

**map\_config.py**

* *EvilSherri* will look like a regular enemy in the game map. Therefore, no changes are required here.

# Conclusion

This document should have provided all the basic steps to get started programming with Tucker Quest. Remember, the goal here is not to write the next award winning computer game, put to learn Python object-oriented programming while having a little fun.

1. - Sample code for new classes

Attached are sample codes for new classes added in the examples of customization in section 8.

**Axe Item Class:**

import json

from attack\_item import AttackItem

from game\_types\_config import ItemType

class Axe(AttackItem):

def \_\_init\_\_(self, id=None, name=None, description=None, min\_power=None, max\_power=None,

attack\_frequency=None):

# initialize base class

super().\_\_init\_\_(id, name, description, min\_power, max\_power)

# initialize type in base class

self.type = ItemType.AXE\_ITEM

# initialize attack frequency, and the counter of attacks

self.attack\_frequency = attack\_frequency

self.attack\_counter = 0

# generate a random attack power between min\_power and max\_power

def generate\_attack(self):

# if my counter is at zero, means we're ready to attack

if (self.attack\_counter == 0):

# first reset my counter to the frequency - 1 (to account for the current attack)

self.attack\_counter = self.attack\_frequency - 1

# generate an attack using standard mechanism

return super().generate\_attack()

# else, we are not ready to attack yet

else:

# reduce the attack counter by 1

self.attack\_counter -= 1

# return an attack power of 0, since we're not ready to attack

return 0

@staticmethod

# create an object from a dictionary

def from\_dict(item\_dict):

return Axe(id=item\_dict["id"], name=item\_dict["name"],

description=item\_dict["description"],

min\_power=item\_dict["min\_power"], max\_power=item\_dict["max\_power"],

attack\_frequency=item\_dict["attack\_frequency"])

# return string representation of object, as a JSON structure

def \_\_str\_\_(self):

return json.dumps(self, default=lambda o: o.\_\_dict\_\_)

**Elf Player Class:**

from player\_character import PlayerCharacter

import json

from game\_types\_config import PlayerType

from fighter import Fighter

class Elf(PlayerCharacter):

# defaults max health

DEFAULT\_ELF\_MAX\_HEALTH = 1000

DEFAULT\_ELF\_ATTACK\_LOSS = 20

DEFAULT\_ELF\_DEFENSE\_GAIN = 20

DEFAULT\_ELF\_MIN\_DEFENSE = 10

def \_\_init\_\_(self, id=None, name=None, pronoun = None, status=None, description=None,

level=None, max\_health=None, attack\_loss=None, defense\_gain=None):

# initialize the PlayerCharacter

PlayerCharacter.\_\_init\_\_(self, name=name, pronoun = pronoun, status=status, description=description, id=id,

level=level, max\_health=max\_health)

# initialize character type

self.type = PlayerType.ELF

# initialize values for this class

self.attack\_loss = attack\_loss

self.defense\_gain = defense\_gain

# this is a custumized version of the attack method for this character type.

def attack(self, opponent):

# get attack power for this fighter (use default if no attack item is equipped)

attack\_power = Fighter.DEFAULT\_ATTACK\_POWER

if (self.attack\_item is not None):

attack\_power = self.attack\_item.generate\_attack()

# set attack loss based on attrinute or default

attack\_loss = self.attack\_loss if self.attack\_loss is not None else Elf.DEFAULT\_ELF\_ATTACK\_LOSS

# reduce attack power by the attack loss percentage)

attack\_power = int(attack\_power \* (1 - attack\_loss / 100))

# invoke enemy's defense function to determine damage

defense\_power = opponent.defense(attack\_power)

# calculate damage of this attack (if defense is greater than attack, set damage to 0)

damage = (attack\_power - defense\_power) if (defense\_power < attack\_power) else 0

# reduce openents health by the damage

opponent.decrease\_health(damage)

# return tuple to indicate result of the attack

return (attack\_power, defense\_power, damage)

# this is a custumized version of the defense method for this character type.

def defense(self, attack\_power):

defense\_power = Elf.DEFAULT\_ELF\_MIN\_DEFENSE

if (self.defense\_item is not None):

defense\_power = self.defense\_item.generate\_defense()

# set defense gain based on attrinute or default

defense\_gain = self.defense\_gain if self.defense\_gain is not None else Elf.DEFAULT\_ELF\_DEFENSE\_GAIN

# increase defense power by the defense gain percentage

defense\_power = int(defense\_power \* (1 + defense\_gain / 100))

return defense\_power

@staticmethod

# create an object from a dictionary

def from\_dict(char\_dict):

return Elf(id=char\_dict["id"], name=char\_dict["name"], pronoun=char\_dict["pronoun"],

status=char\_dict["status"],

description=char\_dict["description"], level=char\_dict["level"],

max\_health=char\_dict["max\_health"],

attack\_loss=char\_dict["attack\_loss"], defense\_gain=char\_dict["defense\_gain"])

# return string representation of object, as a JSON structure

def \_\_str\_\_(self):

return json.dumps(self, default=lambda o: o.\_\_dict\_\_)

**Thief NPC Class**

from non\_player\_character import NonPlayerCharacter

import json

from game\_types\_config import NPCType

import game\_globals as gg

import random

# These strings should preferrably go into game\_text.py file for consistency

THIEF\_NO\_ITEMS = "Arg! You have no items for me to steal. You're lucky!"

THIEF\_STOLE\_ITEMS = "Ah ha! I hope you didn't want this {}, because I stole it."

class Thief(NonPlayerCharacter):

def \_\_init\_\_(self, id = None, name = None, pronoun = None,

status = None, description = None,

inventory = None, message = None):

# initialize the Character elements

NonPlayerCharacter.\_\_init\_\_(self, name = name, pronoun = pronoun, status = status,

description = description, id = id, inventory = inventory, message = message)

# initialize character type

self.type = NPCType.THIEF

# initialize values set later

self.inventory\_size = 0

# Remove a random item from the player's inventory.

# The current player can be retrieved from the game session

def post\_action(self):

# get a reference to the current player

player = gg.game\_session.player

# get a reference to the player's inventory

player\_inventory = player.inventory

# if the player had no items, return a general message

if player\_inventory is None or len(player\_inventory) == 0:

return THIEF\_NO\_ITEMS

# pop a random item from the player's inventory

item\_idx = random.randint(0, len(player\_inventory) - 1)

item = player\_inventory.pop(item\_idx)

# return message indicatin item taken

return THIEF\_STOLE\_ITEMS.format(item.name)

# create an object from a dictionary

@staticmethod

def from\_dict(char\_dict):

return Thief(id=char\_dict["id"], name=char\_dict["name"], status=char\_dict["status"],

pronoun=char\_dict["pronoun"],

description=char\_dict["description"], inventory = char\_dict["inventory"],

message=char\_dict["message"])

# return string representation of object, as a JSON structure

def \_\_str\_\_(self):

return json.dumps(self, default=lambda o: o.\_\_dict\_\_)

1. Tucker Quest, as well as other references to “Tucker”, are a homage to my dog Tucker [↑](#footnote-ref-1)