The SwamPy Developer's Guide

Serialization

From Wikipedia:

> In computer science, in the context of data storage, serialization (or serialisation) is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment).

In the previous chapters of this guide, we discussed writing locations, CharacterClasses, Entities, and Items. In this chapter, we will discuss how `MuddySwamp` serializes in-game objects, and how you can hitch into this process with the `serialize` and `deserialize` methods. We treated `mudimport.py` as a blackbox, something that magically loaded in our files. While we can get away with this strategy for merely importing game data, we cannot do the same if we want to properly saves of our CharacterClasses, Entities, and Items.

So let us begin by discussing how `mudimport.py` works.

Importing vs. Loading

First, we must differentiate between the terms *import* and *load* in this context. When we say *import*, we mean "take something written by the game-designer, and bring it into the game world."

For instance, we can take `darktower.yaml` and import it into the game at runtime.

```
#TODO-docs: add code here
```

This is actually what `MuddySwamp.py` has been doing behind the scenes the entire time: *importing* all the location files at runtime.

Contrast that with loading / saving

A New View of the World

We've previously considered the world as something of a network. Each "node" is a location, linked to any number of other locations. However, if we consider the relationships between each type of in-game element, a clear hierarchy emerges. We start with locations, which are firmly planted in the world.

```
WORLD
  LOCATIONS
```

Characters are always stored in a location…

```
WORLD
  LOCATIONS
     CHARACTERS
```

…as are entities

```
WORLD
  LOCATIONS
     CHARACTERS
     ENTITIES
```

Items can be stored inside a location's inventory, or inside a character/entity.

```
WORLD
  LOCATIONS
     CHARACTERS
        ITEMS
     ENTITIES
        ITEMS
     ITEMS
```

This final tree suggests a possible order for importing game data. First we import locations since all characters, entities, and items must exist inside a location. Then, we can import characters, entities, and finally items.

However, this strict hierarchy neglects a few things. When we were planning out our world, we realized that we wanted a sort of "Soul Stone" that a player could hold in their inventory. When players use the soul stone, all their data is safely stored inside it, and they can safely be logged off.

How would this fit into our schema? Wouldn't this imply that a character could be inside an item? For that matter, why not have an entity inside an item? We could imagine a "mecha schematic" that produces some kind of mech.

In fact, we can barely differentiate between characters, entities, and items for the purposes of serialization. So let's adjust the model.

```
WORLD
  LOCATIONS
     CHARACTERS / ITEMS / ENTITIES
         OTHER CHARACTERS / ITEMS / ENTITIES
             ...
```

So, we've broken our beautiful hierarchy. Now we have a recursive mess of characters and entities inside items inside other entities and characters. This model is more complicated, but it offers more sophisticated game mechanics that we simply couldn't resist.

So, let us describe how to

The Prelude

First, we need to tell `MuddySwamp` how to interpret names like “DarkWizard” or “IronSword”. For this, we simply need to import