How to Write a Text Adventure in Python Part 1: Items and Enemies – Let's Talk Data

By Phillip Johnson

This is an abbreviated version of the book Make Your Own Python Text Adventure.

Typically, a text adventure game involves the player exploring and interacting with a world to tell a story. For this tutorial, I wanted the game to take place in a cave with the goal of escaping alive. You can of course use this idea too, but feel free to use your own idea! Almost all parts of this tutorial are interchangeable with your own custom elements. We're going to start by defining some items and enemies.

Items

Start by creating a new directory called `adventuretutorial` and create a blank file called `__init__.py`. This tells the Python compiler that adventuretutorial is a Python package which contains modules. Go ahead and create your first module in this same directory called `items.py`.

The first class we are going to create is the `Item` class. When creating a class, consider the attributes that class should have. In general, it will be helpful for items to have a name and description. We can use these to give the player information about the item. Let's also add a value attribute that will help players to compare items within an economy.

```
1
2
3
4
5
6
7
8
9
```

```
class Item():

    def __init__(self, name, description, value):

        self.name = name

        self.description = description

        self.value = value

    def __str__(self):

        return "{}\n=====\n{}\nValue: {}\n".format(self.name,
self.description, self.value)
```

This class has just two methods, but we're going to see them pop up a lot. The `__init__` method is the **constructor** and it is called whenever a new object is created. The `__str__` method is usually a convenience for programmers as it allows us to print an object and see some useful information. Without a `__str__` method, calling `print()` on an object will display something unhelpful like `<__main__.Item object at 0x101f7d978>`.

While we *could* create an `Item` directly, it wouldn't be very interesting for the player because there is nothing special about it. Instead, we're going to **extend** the `Item` class to define more specific items.

One of my favorite parts of games is finding treasure so let's go ahead and create a `Gold` class.

```
class Gold(Item):

    def __init__(self, amt):

        self.amt = amt

        super().__init__(name="Gold",

                         description="A round coin with {} stamped on the
front.".format(str(self.amt)),

                         value=self.amt)
```

The `Gold` class is now a **subclass** of the **superclass** `Item`. Another word for a subclass is **child class** and superclasses may be called **parent** or **base** classes.

The constructor here looks a little confusing but let's break it down. First you'll notice an additional parameter `amt` that defines the amount of this gold. Next, we call the superclass constructor using the `super().__init__()` syntax. **The superclass constructor must always be called by a subclass constructor.** If the constructors are exactly the same, Python will do it for us. However, if they are different, we have to explicitly call the superclass constructor. Note that this class doesn't have a `__str__` method. If a subclass doesn't define its own `__str__` method, the superclass method will be used in its place. That's OK for the `Gold` class because the value is the same as the amount so there's no reason to print out both attributes.

I mentioned earlier that this game is going to have weapons. We could extend `Item` again to make some weapons, but weapons all have something in general: damage. Whenever a lot of specific classes are going to share the same concept, it's usually a good idea to store that shared concept in a superclass. To do this, we will extend `Item` into a `Weapon` class with a `damage` attribute and then extend `Weapon` to define some specific weapons in the game.

```
class Weapon(Item):

    def __init__(self, name, description, value, damage):

        self.damage = damage

        super().__init__(name, description, value)

    def __str__(self):

        return "{}\n=====\n{}\nValue: {}\nDamage: {}".format(self.name,
self.description, self.value, self.damage)

class Rock(Weapon):
```

```
    def __init__(self):
        super().__init__(name="Rock",
                         description="A fist-sized rock, suitable for bludgeoning.",
                         value=0,
                         damage=5)

class Dagger(Weapon):
    def __init__(self):
        super().__init__(name="Dagger",
                         description="A small dagger with some rust. Somewhat more dangerous than a rock.",
                         value=10,
                         damage=10)


```

This should feel very familiar as we are following the same process of creating subclasses to define more specific elements in the game.

Enemies

Now that you're an expert at extending objects, creating the enemies will be a breeze. Go ahead and create a new module called `enemies.py`.

Our base class `Enemy` should include a name, hit points, and damage the enemy does to the player. We're also going to include a method that allows us to quickly check if the enemy is alive.

```

class Enemy:
    def __init__(self, name, hp, damage):
        self.name = name
        self.hp = hp
        self.damage = damage
    def is_alive(self):
        return self.hp > 0

```

Next, create a few subclasses of `Enemy` to define some specific enemies you want the player to encounter. Here are mine, but feel free to use your own ideas instead.

```
1
2
3
4
5
6
7
class GiantSpider(Enemy):
    def __init__(self):
        super().__init__(name="Giant Spider", hp=10, damage=2)
class Ogre(Enemy):
    def __init__(self):
        super().__init__(name="Ogre", hp=30, damage=15)
```

Notice that we don't have to include `is_alive()` in the subclasses. This is because subclasses automatically get access to the methods in the superclass.

If you are completely new to programming, some of this may be confusing. My book *Make Your Own Python Text Adventure* is great for beginners because it does not assume any knowledge of programming concepts.

Click here for part 2