# How to Write a Text Adventure in Python Part 2: The World Space – Let's Talk Data

*By Phillip Johnson*

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

All games take place in some sort of world. The world can be as simple as a chess board or as complex as the *Mass Effect* universe and provides the foundation for the game as a whole. All elements of a game reside in the world and some elements interact with the world. In this post, you'll learn how to add items and enemies to your world.

### The coordinate plane

A text adventure usually involves a player moving through the world one section per turn. We can think of each section as a tile on an x-y grid. Note: in most game programming the x-y coordinate plane is different from the one you learned in algebra. In the game world, (0,0) is in the top left corner, x increases to the right, and y increases to the bottom.

### Creating tiles

Start by creating a module tiles.py with this class:

```
1
2
3
4
5
6
```

```
import items, enemies

class MapTile:

    def __init__(self, x, y):

        self.x = x

        self.y = y
```

The `import` keyword means "give this module access the 'items' module and 'enemies' module. We need this because we will want to put these elements inside some of our rooms.

The `MapTile` class is going to provide a template for all of the tiles in our world, which means we need to define the methods that all tiles will need. First, we'll want to display some text to the user when they enter the tile that describes the world. We also expect that some actions may take place when the player enters the tile, and that those actions change the state of the player (e.g., they pick something up, they win the game, something attacks them, etc.). Let's add those methods now.

```
1
2
3
4
5
```

```
def intro_text(self):

    raise NotImplementedError()

def modify_player(self, player):

    raise NotImplementedError()
```

We haven't talked about the code for the player yet, but that's OK. The `player` parameter will serve as a placeholder. As you might guess, these methods aren't going to do much in their current state. In fact, they will actually cause the program to crash! This might seem silly, but this behavior is to help us as programmers.

When thinking about our world, we don't want to have tiles that do *nothing*. We may want tiles of water, tiles in a spaceship corridor, tiles with other characters, or tiles with treasure, but not empty tiles. So this `MapTile` class is actually just a template that all other tiles will expand on.

In the last post we learned about base classes. `MapTile` is actually a specific flavor of a base class. We call it an ***abstract* base class** because we don't want to create any instances of it. In our game, we will only create specific types of tiles. We will never create a `MapTile` directly, instead we will create subclasses. The code `raise NotImplementedError()` will warn us if we accidentally create a `MapTile` directly.

Now on to our first tile subclass!

```
class StartingRoom(MapTile):

    def intro_text(self):

        return

    def modify_player(self, player):

        pass
```

This class extends `MapTile` to make a more specific type of tile. We **override** the `intro_text` and `modify_player` methods to implement the specific behavior that this tile should have. A method is overridden when a subclass has the same method name as a superclass. Because it's the starting room, I didn't want anything to happen to the player. The `pass` keyword simply tells Python to not do anything. You might wonder why the method is even in this class if it doesn't do anything. The reason is because if we don't override `modify_player`, the superclass's `modify_player` will execute and if that happens the program will crash because of `raise NotImplementedError()`.

Next, let's add a class for the tile where a player will find a new item.

```
class LootRoom(MapTile):

    def __init__(self, x, y, item):

        self.item = item

        super().__init__(x, y)

    def add_loot(self, player):

        player.inventory.append(self.item)

    def modify_player(self, player):

        self.add_loot(player)
```

Remember, we haven't created `player` yet, but we can guess that the player will have an inventory.

Let's define one more type of room: a room in which the player encounters an enemy.

```
class EnemyRoom(MapTile):

    def __init__(self, x, y, enemy):

        self.enemy = enemy
```

```
        super().__init__(x, y)

    def modify_player(self, the_player):
        if self.enemy.is_alive():
            the_player.hp = the_player.hp - self.enemy.damage
            print("Enemy does {} damage. You have {} HP
remaining.".format(self.enemy.damage, the_player.hp))
```

This constructor should look familiar to you now. It's very similar to the `LootRoom` constructor, but instead of an item, we are working with an enemy.

The logic for this room is a bit different. I didn't want enemies to respawn. So if the player already visited this room and killed the enemy, they should not engage battle again. Assuming the enemy is alive, they attack the player and do damage to the player's hit points.

Now that we have some basic types of tiles defined, we can make some even more specific versions. Here are some that I created:

```

class EmptyCavePath(MapTile):
    def intro_text(self):
        return
    def modify_player(self, player):
        pass
class GiantSpiderRoom(EnemyRoom):
    def __init__(self, x, y):
        super().__init__(x, y, enemies.GiantSpider())
    def intro_text(self):
        if self.enemy.is_alive():
            return
        else:
            return
class FindDaggerRoom(LootRoom):
    def __init__(self, x, y):
        super().__init__(x, y, items.Dagger())
    def intro_text(self):
        return
```

```
26
27
28
29
30
31
32
33
```

If you remember, I also created an `Ogre` enemy and `Gold` item. You may choose to create corresponding rooms too.

### Creating the world

We're going to close out this post by actually creating a world based on the tiles we've defined. This delves into some advanced features so it's OK if you don't follow everything. I'll explain everything briefly here, but I encourage you to read up on anything you're interested in learning more about.

Create a new module in the same directory called `world.py`. Next, make a folder called "resources" that is in the same directory as the "adventuretotrial" directory and create `map.txt` inside. We're going to build the world in this external file and load it into the game programatically.

I like to use a spreadsheet program and then copy the text into the map file, but you can just edit the file directly too. The goal is to lay out a grid of tiles whose names match the class names and are separated by tabs. Here's an example in a spreadsheet:

| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | | | | LeaveCaveRoom | |
| 2 | | Find5GoldRoom | EmptyCavePath | GiantSpiderRoom | |
| 3 | | | EmptyCavePath | | |
| 4 | | | EmptyCavePath | | |
| 5 | GiantSpiderRoom | EmptyCavePath | StartingRoom | EmptyCavePath | FindDaggerRoom |
| 6 | | | Find5GoldRoom | | |
| 7 | | | EmptyCavePath | | |
| 8 | | | SnakePitRoom | | |
| 9 | | | | | |

Remember, your map should not include MapTile, LootRoom, or EnemyRoom! Those are base classes that should not be created directly.

In the `world` module, add the following dictionary and method to parse the file you created.

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

```
_world = {}

starting_position = (0, 0)

def load_tiles():
    with open('resources/map.txt', 'r') as f:
        rows = f.readlines()
    x_max = len(rows[0].split('\t'))
    for y in range(len(rows)):
        cols = rows[y].split('\t')
        for x in range(x_max):
```

```
            tile_name = cols[x].replace('\n', '') # Windows users may need to replace
'\r\n'
            if tile_name == 'StartingRoom':
                global starting_position
                starting_position = (x, y)
            _world[(x, y)] = None if tile_name == '' else getattr(__import__('tiles'),
tile_name)(x, y)


```

The parsing method goes through each line of the file and splits the line into cells. Using a double `for` loop is a common way of working with grids. The `x` and `y` variables keep track of the coordinates. When we find the Starting Room, that position is saved because we will use it later. We use the `global` keyword to let us access the `starting_position` variable that lives outside of this method. The last line is the most interesting, but it's fine if you don't fully understand it.

The variable `_world` is a dictionary that maps a coordinate pair to a tile. So the code `_world[(x, y)]` creates the key (i.e. the coordinate pair) of the dictionary. If the cell is an empty string, we don't want to store a tile in it's place which is why we have the code `None if tile_name == ''`. However, if the cell *does* contain a name, we want to actually create a tile of that type. The `getattr` method is built into Python and lets us **reflect** into the tile module and find the class whose name matches `tile_name`. Finally the `(x, y)` passes the coordinates to the constructor of the tile.

**Looking for an easier approach to building your world? Check out Chapter 13 of *Make Your Own Python Text Adventure*.**

Essentially what we're doing is using some advanced features in Python as an alternative to something like this:

```

tile_map = [[FindGoldRoom(),GiantSpiderRoom(),None,None,None],
            [None,StartingRoom(),EmptyCave(),EmptyCave(),None]
           ]
```

That's hard to read and maintain. Using a text file makes changing our world easy. It's also a lot simpler to visualize the world space.

Keep in mind that the only reason we are able to do this is because all of our tile classes derive from the same base class with a common constructor that accepts the parameters x and y.

Let's add one more method to the world module that will make working with the tiles a little easier:

```

def tile_exists(x, y):
    return _world.get((x, y))
```