How to Write a Text Adventure in Python Part 3: Player Action – Let's Talk Data

By Phillip Johnson

This is an abbreviated version of the book [Make Your Own Python Text Adventure](#).

So far we've created a world and filled it with lots of interesting things. Now we're going to create our player and provide ways for the player to interact with the world. This will probably be the most conceptually challenging part of the game, so you may want to re-read this section a few times.

The Player

Time for a new module! Create `player.py` and include this class:

```
1
2   import items
3
4   class Player():
5       def __init__(self):
6           self.inventory = [items.Gold(15), items.Rock()]
7           self.hp = 100
8           self.location_x, self.location_y = world.starting_position
9           self.victory = False
10      def is_alive(self):
11          return self.hp > 0
12      def print_inventory(self):
13          for item in self.inventory:
14              print(item, '\n')
15
```

Now you can see some of the concepts that we previously templated have been made into reality. The player starts out with a few basic items and 100 hit points. We also load the starting location that was saved before and create a victory flag that will notify us if the player has won the game. The methods `is_alive` and `print_inventory` should be self-explanatory.

Adding Actions

Now that we have a player, we can start to give them actions. We'll start with moving around first.

```
1
    def move(self, dx, dy):
2
        self.location_x += dx
3
        self.location_y += dy
4
        print(world.tile_exists(self.location_x,
5   self.location_y).intro_text())
6   def move_north(self):
7       self.move(dx=0, dy=-1)
8   def move_south(self):
9       self.move(dx=0, dy=1)
10  def move_east(self):
11      self.move(dx=1, dy=0)
    def move_west(self):
12
        self.move(dx=-1, dy=0)
13
14
15
16
```

The player can move in four directions: north, south, east, and west. To avoid repeating ourselves, we have a basic `move` method that takes care of actually changing the player's position and then we have four convenience methods that use the common `move` method. Now we can simply refer to `move_south` without specifically trying to remember if `y` should be positive or negative, for example.

The next action the player should have is attack.

```
def attack(self, enemy):
    best_weapon = None
    max_dmg = 0
    for i in self.inventory:
        if isinstance(i, items.Weapon):
            if i.damage > max_dmg:
                max_dmg = i.damage
                best_weapon = i

    print("You use {} against {}!".format(best_weapon.name, enemy.name))
    enemy.hp -= best_weapon.damage
    if not enemy.is_alive():
        print("You killed {}!".format(enemy.name))
```

```
    else:
        print("{} HP is {}.".format(enemy.name, enemy.hp))
```

In order to find the most powerful weapon in the player's inventory, we loop through all the items and use `isinstance` (a built-in function) to see if the item is a `Weapon`. This is another feature we gain by having all of our weapons share a common class. If we didn't do this, we would need to do something messy like `if item.name=="dagger" or item.name=="rock" or item.name=="sword"...`. The rest of the method actually attacks the enemy and reports the result back to the user.

We now have behavior defined for certain actions. But within the game, we need some additional information. First, we need to bind keyboard keys to these actions. It would also be nice if we had a "pretty" name for each action that could be displayed to the player. Because of this additional "meta" information, we are going to wrap these behavior methods inside of classes. It's time for a new module called `actions.py`

```
import Player

class Action():

    def __init__(self, method, name, hotkey, **kwargs):

        self.method = method

        self.hotkey = hotkey

        self.name = name

        self.kwargs = kwargs

    def __str__(self):

        return "{}: {}".format(self.hotkey, self.name)


```

We're going to use the now-comfortable design of a base class with specific subclasses. For starters, the `Action` class will have a method assigned to it. This method will correspond directly to one of the action methods in the player class, which you will see shortly. Additionally, each `Action` will have a hotkey, the "pretty" name, and a slot for additional parameters. These additional parameters are specified by the special `**` operator and are named `kwargs` by convention. Using `**kwargs` allows us to make the `Action` class extremely flexible. We know all actions will require certain parameters, but there may be additional parameters that are different for certain actions. For example, we've already seen the `attack` method that requires an enemy parameter.

The following classes are our first wrappers:

```
class MoveNorth(Action):

    def __init__(self):

        super().__init__(method=Player.move_north, name='Move north',
```

```
hotkey='n')

class MoveSouth(Action):

    def __init__(self):

        super().__init__(method=Player.move_south, name='Move south',
hotkey='s')

class MoveEast(Action):

    def __init__(self):

        super().__init__(method=Player.move_east, name='Move east',
hotkey='e')

class MoveWest(Action):

    def __init__(self):

        super().__init__(method=Player.move_west, name='Move west',
hotkey='w')

class ViewInventory(Action):

    def __init__(self):

        super().__init__(method=Player.print_inventory, name='View
inventory', hotkey='i')
```

Notice how the method parameter actually points to a specific method in the Player class. Referring to methods as objects in a feature in Python and other languages with “first class” methods. Be sure that you do not include `()` after the method name. The code `Player.some_method()` will execute the method whereas `Player.some_method` is just a reference to the method as an object.

The attack method wrapper is very similar with one small difference:

```

class Attack(Action):

    def __init__(self, enemy):

        super().__init__(method=Player.attack, name="Attack", hotkey='a',
enemy=enemy)
```

Here we have included the “enemy” parameter as previously discussed. Since `enemy` is not a named parameter in the base `Action` class constructor, it will get bundled up into the `**kwargs` parameter.

Now that we have some actions defined, we need to consider how they will be used in the game. For example, the player should not be able to attack when no enemy is present. Conversely, they shouldn’t be able to calmly leave a room that

has an enemy! Actions should be available or unavailable based on the context of the situation. To handle this, we need to flip back to our tiles module.

Change your import statement to include the actions and world modules:

```
import items, enemies, actions, world
```

Next add the following methods to `MapTile`:

```
def adjacent_moves(self):
    moves = []
    if world.tile_exists(self.x + 1, self.y):
        moves.append(actions.MoveEast())
    if world.tile_exists(self.x - 1, self.y):
        moves.append(actions.MoveWest())
    if world.tile_exists(self.x, self.y - 1):
        moves.append(actions.MoveNorth())
    if world.tile_exists(self.x, self.y + 1):
        moves.append(actions.MoveSouth())
    return moves
def available_actions(self):
    moves = self.adjacent_moves()
    moves.append(actions.ViewInventory())
    return moves
```

These methods provide some default behavior for a tile. The default actions that a player should have are: move to any adjacent tile and view inventory. The method `adjacent_moves` determines which moves are possible in the map. For each available action, we append an instance of one of our wrapper classes to the list. Since we used the wrapper classes, we will later have easy access to the names and hotkeys of the actions.

Now we need to allow the `Player` class to take an `Action` and run the action's internally-bound method. Add this method to the `Player` class:

```
def do_action(self, action, **kwargs):
    action_method = getattr(self, action.method.__name__)
    if action_method:
        action_method(**kwargs)
```

That `getattr` rears its head again! We have a similar concept to what we did to create tiles, but this time instead of looking for a *class* in a *module*, we're looking for a *method* in a *class*. For example, if `action` is a `MoveNorth` action, then we know that its internal method is `Player.move_north`. The `__name__` of that method is "move_north". Then `getattr` finds the `move_north` method inside the `Player` class and stores that method as the object `action_method`. If `getattr` was successful, we execute the found method and we include the `**kwargs` in case that method needs additional objects (like the `attack` method).

Looking for something a little simpler? My book *Make Your Own Python Text Adventure* has a different approach to player actions that avoids the sometimes confusing `getattr` and `**kwargs`.

At this point, I decided to add one more action: flee. As an alternative to battle, the player can flee which causes them to a random adjacent tile. Here's the behavior for the `Player` in the `players` module:

```
import random
import items, world
class Player:
    def flee(self, tile):
        available_moves = tile.adjacent_moves()
        r = random.randint(0, len(available_moves) - 1)
        self.do_action(available_moves[r])
```

And here's our wrapper in the `actions` module:

```
class Flee(Action):

    def __init__(self, tile):

        super().__init__(method=Player.flee, name="Flee", hotkey='f',
tile=tile)
```

Similar to the attack action, the flee action requires an additional parameter. This time, it's the tile from which the player needs to flee.

Most of the tiles we have created so far can use the default available moves. However, the enemy tiles need to provide the attack and flee actions. To do this, we will override the default behavior of the `MapTile` class with our own version of the method in the `EnemyRoom` class.

```
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))

    def available_actions(self):

        if self.enemy.is_alive():

            return [actions.Flee(tile=self),
actions.Attack(enemy=self.enemy)]

        else:

            return self.adjacent_moves()

```

If the enemy is still alive then the player's only options are attack or flee. If the enemy is dead, then this room works like all other rooms.

Whew! That was a lot of new code. We're almost finished! All we need to do to wrap up is create and interface for the human player.

Click here for part 4