test plan

Desc	
Tags	Complete

Test Combat Functions

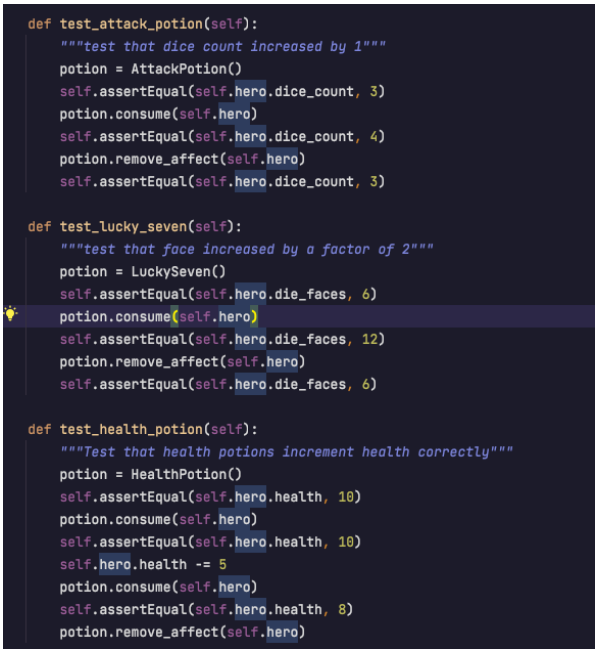
The combat function tests are to verify that health is actually being decremented when a character lands a hit. It was guaranteed a hit was going to be hit by modifying the combat rolls directly.

```python
def test_duel(self):
    """Test that combat is functioning properly"""
    self.hero._combat_rolls = [6, 6, 3]
    self.monster._combat_rolls = [3]

    self.assertEqual(self.monster.health, 3)
    self.assertTrue(duel(self.hero, self.monster, self.spooky, debug=True))
    self.assertEqual(self.monster.health, 2)

    self.heavy_hand.add_qty(3)

def test_with_potions(self):
    """Test that multiple potions still result in expected damage to monster"""

    # set increase damage and guarantee hit
    self.heavy_hand.consume(self.hero)
    self.pierce_shot.consume(self.hero)

    self.hero.combat_roll()

    # make sure we have all six
    for die in self.hero.combat_rolls:
        self.assertEqual(die, 6)

    self.monster._combat_rolls = [3]

    self.assertEqual(self.monster.health, 3)
    self.assertTrue(duel(self.hero, self.monster, self.spooky, debug=True))
    self.assertEqual(self.monster.health, 1)
```

As expected, all test cases returned what was expected of them

Test Consumable Objects

A consumable object is a directive of the loot class. The only difference, is that a consumable can be consumed to modify the hero for one turn. The tests are pretty straight forward. Test the hero's value that will be changed before consuming the consumable then, test afterwards. Then test that the affects wore off.



```python
def test_attack_potion(self):
    """test that dice count increased by 1"""
    potion = AttackPotion()
    self.assertEqual(self.hero.dice_count, 3)
    potion.consume(self.hero)
    self.assertEqual(self.hero.dice_count, 4)
    potion.remove_affect(self.hero)
    self.assertEqual(self.hero.dice_count, 3)

def test_lucky_seven(self):
    """test that face increased by a factor of 2"""
    potion = LuckySeven()
    self.assertEqual(self.hero.die_faces, 6)
    potion.consume(self.hero)
    self.assertEqual(self.hero.die_faces, 12)
    potion.remove_affect(self.hero)
    self.assertEqual(self.hero.die_faces, 6)

def test_health_potion(self):
    """Test that health potions increment health correctly"""
    potion = HealthPotion()
    self.assertEqual(self.hero.health, 10)
    potion.consume(self.hero)
    self.assertEqual(self.hero.health, 10)
    self.hero.health -= 5
    potion.consume(self.hero)
    self.assertEqual(self.hero.health, 8)
    potion.remove_affect(self.hero)
```

```python
def test_heavy_hand(self):
    """Test that damage is increased from heavy hand potion"""
    potion = HeavyHand()
    self.assertEqual(self.hero.damage, 1)
    potion.consume(self.hero)
    self.assertEqual(self.hero.damage, 2)
    potion.remove_affect(self.hero)
    self.assertEqual(self.hero.damage, 1)

def test_pierce_shot(self):
    """Test that pierce shot buff is enabled with potion"""
    potion = PierceShot()
    self.assertFalse(self.hero.pierce_shot)
    potion.consume(self.hero)
    self.assertTrue(self.hero.pierce_shot)
    potion.remove_affect(self.hero)
    self.assertFalse(self.hero.pierce_shot)
```

Test Environment

The two main pieces to check in this test is that initiative is performed correctly and that the environment holds loot dropped by the monsters in the room. To test them, I artificially "added" loot to the room to make sure the room is able to hold them and checked the the initiative returned a enum

Test Hero and Test Monster

These are straight forward. Determine if a hero and a monster are able to be instantiated and values modified through setters and getters.

Test Monster Controller

Monster controller is my favorite feature of this project. Monster controller spawns monsters based on the environment. Each environment has a set of monsters that can potentially spawn that way the dialog always matches when the "monsters" talk to you.

But since a requirement is to support the instructors dd_monster file, the mosnter_controller has to be able to adapt. If it doesn't find a environment or monsters that belong together it will just pull at random. Not as elegant, but it works.

As expected, the mosnter_controller is able to either spawn monsters based on environment or default to spawning random monster.

```
class TestMonsterController(TestCase):
    def setUp(self) -> None:
        self.env1 = Environment("Cave", "Scary", ["Imp", "Mad Cow"], 1)
        self.env2 = Environment("Cave", "Scary", level=5)

    def test_monster_one_count(self):
        """Test creating of 1 monster"""
        monster_controller = MonsterController(EnvironmentRecord(self.env1.level, self.env1.habitable))
        self.assertEqual(monster_controller.monster_count, 1)

    def test_monster_multi_count(self):
        """Increment level and get monster count"""
        monster_controller = MonsterController(EnvironmentRecord(self.env2.level, self.env2.habitable))
        self.assertEqual(monster_controller.monster_count, 3)

    def test_monster_deaths(self):
        """Test if the monsters are popped from the list"""
        monster_controller = MonsterController(EnvironmentRecord(self.env2.level, self.env2.habitable))
        self.assertEqual(monster_controller.monster_count, 3)

        monster_controller.monsters[0].is_dead = True
        monster_controller.monsters[2].is_dead = True
        monster_controller.clean_up()
        self.assertEqual(monster_controller.monster_count, 1)
        self.assertEqual(len(monster_controller.monsters), 1)
```