National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2024/2025

Mission 13
Hungry Games Training, Part I

Release date: 14 April 2025
Due: 15 May 2025, 23:59

Required Files

- mission13-template.py
- hungry_games.py

Background

For thousands of years, the sacred and magical land of Serpentis have remained undetected. Her inhabitants have lived peacefully, learning the ancient arts of rune readings, training magical turtles of different colours, and mastering the skills to read public transportation schedules.

Unfortunately, that was about to change. In the nearby state of Panem, researchers have managed to synthesise a new element, dubbed the Stone of Jordan. The element is able to absorb and store magical essences, and is able to produce an elixir that would reverse the effects of ageing.

Fuelled by the promise of immortality, the Capitol, the totalitarian dictator of Panem, have threatened to attack the sacred land of Serpentis, and sought to turn it into District M, whose sole purpose is to enslave Serpentists for their magical essences.

As the day of the Hungry Games draws near, the Grandmaster Ben made an agreement with The Capitol - they would send 2 volunteers for tributes to the Hungry Games, and should they emerge victorious, the Capitol would destroy all Stone of Jordans in existence, and leave Serpentis alone. The Capitol was happy to grant them this request - there is no way the sheltered mages would be able to survive the Hungry Games, and it would be a way to introduce the new District as one of their own.

You are tasked with preparing and training our volunteers (henceforth knows as tributes) for the Hungry Games.

Information

In this mission, you will be tasked with modelling some of the objects, namely weapons and supplies, that would be used in the Hungry Games. We will be using `hungry_games.py`, as a base from which you will expand and build your own classes. It will definitely be useful to look through all the class definitions provided in the file.

As usual, the template file `mission13-template.py` has been provided for your use. Do not copy the test code onto Coursemology.

This mission consists of **five** tasks.

Task 1: Weapons (6 marks)

In order to survive in the Hungry Games, our tributes would need to be well versed with different weapons. In order to do so, we would need a magical spell from which we can create weapons of varying capabilities.

a. Create a class, `Weapon`, inherited from the `Thing` class.

   The constructor should take in 3 parameters, the name of the weapon, min_dmg and max_dmg, which describes the damage capabilities of the weapon. Note that the name of the weapon is the type of the weapon in string, so multiple weapons may share the same name because they are of the same type.

```
>>> sword = Weapon("sword", 3, 10)
>>> isinstance(sword, Weapon)
True
>>> isinstance(sword, Thing)
True
```

b. Implement the methods, `min_damage()` and `max_damage()`, which would return the minimum and maximum damage for a weapon respectively.

```
>>> sword.min_damage()
3
>>> sword.max_damage()
10
```

c. Implement a method, `damage()`, which would return a value between min_dmg and max_dmg **inclusive**.

   HINT: You may use the `random.randint(min, max)` method to generate a random integer between min and max inclusive.

```
>>> sword.damage()  # Should return a value between 3 and 10
5
```

Task 2: Ammunition (4 marks)

As melee weapons require the user to be up close before they can damage the enemy, being only skilled with melee weapons would expose our tributes to additional threats. It is thus important that we have ranged weapons.

However, the biggest drawback of ranged weapons is that they require a supply of ammo in order to be effective. Sure, you can try and hit someone with your bow, but clearly that would not be very smart...

In this task, we will model the ammunition with the `Ammo` class.

a. Create a class `Ammo`, inherited from the `Thing` class. The constructor should take in 3 parameters, the name of the ammo, the weapon that this ammo is for, as well as the quantity of this ammo (which is the number of shots avaliable initially).

b. Create a method `get_quantity()`, that would return the number of shots available in this ammo object.

c. Create a method `weapon_type()`, that would return the **name** of the type of the weapon that this ammo is supposed to work with. Recall that a weapon's type is simply its name. Hence an ammo can work with different weapon objects as long as they share the same name.

d. Create a method `remove_all()`, that would set the number of shots in the ammo to be 0.

```
# Make the weapon - see next Task.
>>> bow = RangedWeapon("bow", 10, 20)

# Represent a stack of 5 arrows for a bow weapon
>>> arrows = Ammo("arrow", bow, 5)
>>> arrows.get_quantity()
5
>>> arrows.weapon_type()
"bow"
>>> arrows.remove_all()
>>> arrows.get_quantity()
0
```

Task 3: Ranged Weapon (8 marks)

Now that we have a way to create Ammo, it is now time to create a Ranged Weapon. The RangedWeapon class will be a subclass of the Weapon class.

a. Create a class `RangedWeapon`, which inherits from the `Weapon` class. The constructor should take in 3 parameters, the name of the weapon, min_dmg and max_dmg, which describes the damage capabilities of the weapon.

The RangedWeapon should have a property, `shots`, that keeps track of the current ammo supply. A newly created `RangedWeapon` start with 0 shots.

Note that since the type of a weapon is simply its name, it is not unusual to have multiple weapons that share the same name.

```
>>> bow = RangedWeapon("bow", 1, 4)
>>> isinstance(bow, RangedWeapon)
True
>>> isinstance(bow, Weapon)
True
```

b. Create a method `shots_left()`, that returns the ammo supply for the current ranged weapon.

c. Create a method `load(ammo)`, that would take in an `Ammo` object. If the ammo object is meant for the weapon, the shots count for the weapon would be increased by the ammo's quantity, and the ammo quantity will be reduced to 0 (Hint: you can use `remove_all()` to set the quantity of the `Ammo` to 0) However, if the ammo is of the wrong type, there is no effect on either the `Ammo` or the `RangedWeapon`.

```
>>> bow = RangedWeapon("bow", 10, 40)
>>> crossbow = RangedWeapon("crossbow", 15, 45)
>>> arrows = Ammo("arrow", bow, 5)
>>> bolts = Ammo("bolt", crossbow, 10)

>>> bow.load(bolts)
>>> bow.shots_left()
0
>>> bolts.get_quantity()
10

>>> bow.load(arrows)
>>> bow.shots_left()
5
>>> arrow.get_quantity()
0
```

d. Override the `damage()` method for `RangedWeapon` such that it behaves in the following manner:

   (i) Returns 0 if the ranged weapon does not have any shots left

   (ii) Otherwise, decrease the number of shots left by 1, and returns a value between min_dmg and max_dmg inclusive. You should call its superclass's `damage()` method here.

```
>>> crossbow.damage()
0
>>> crossbow.load(bolts)
>>> crossbow.shots_left()
10
>>  bolts.get_quantity()
0
>>> crossbow.damage()      # Between 15 and 45 inclusive
38
>>> crossbow.shots_left()
9
```

Task 4: Supplies (4 marks)

Clearly, having weapons is definitely not enough. In the Hungry Games, we will definitely need to have food and medicine in order to survive. We will now implement these objects.

a. Create a class `Food`, that inherits from the `Thing` class.

   The constructor should take in 2 parameters, `name`, and `food_value`, which describes how filling the food is. Your food object should have a `get_food_value()` method that returns the food value.

```
>>> apple = Food("apple", 4)
>>> apple.get_food_value()
4
```

b. In the Hungry Games, a Medicine is considered to be a kind of Food. You can consume medicine to increase your health, in addition, some medicine will also help cure hunger. Create a class `Medicine`, that inherits from the `Food` class.

   The constructor should take in 3 parameters, `name`, and `food_value`, which describes how filling the food is, and `medicine_value`, which describes how effective the medicine is. Your medicine object should have an additional method called `get_medicine_value()` which will return the medicinal value of the medicine.

```
>>> panadol = Medicine("paracetamol", 0, 5)
>>> panadol.get_food_value()
0
>>> panadol.get_medicine_value()
5
```

Task 5: Animals (6 marks)

Finally, we will need a way to model animals. As the Hungry Games takes place in the wildness, tributes would need to learn how to kill animals for their food.

a. Create a class `Animal`, inherited from the `LivingThing` class.

   The constructor should take in 4 parameters, `name`, `health`, which is the number of damage the animal can take before it dies, `food_value`, which describe how filling the `Food` of the `Animal` is, and an **optional** argument `threshold`, which determines how often the animal will move.

   If the last parameter `threshold` is not specified, the threshold of the animal object will be a random value between 0 and 4 **inclusive**.

   The random threshold value should have a chance to be different between 2 different instantiation of Animal. Therefore fixing the threshold to one single "random" value that stays the same between 2 different instantiations is incorrect.

   In addition, implement `get_food_value()`, which will return the food value of the `Food` object that will be dropped when the animal is killed.

```
>>> bear = Animal("bear", 20, 10, 3)
>>> bear.get_threshold()
3
>>> bear.get_food_value()
10

>>> deer = Animal("deer", 15, 6)
>>> deer.get_threshold()   # Between 0 and 4 inclusive
2
```

b. Using polymorphism, override the `go_to_heaven()` method of LivingThings such that when an animal really `go_to_heaven`, a `Food` object of name "<animal's name> meat" and food_value of `animal.get_food_value()` will be created at the place where the animal was. E.g. If an animal named "bear" was killed, a `Food` object with the name of "bear meat" will be produced.

   HINT: `go_to_heaven()` is already defined in the `LivingThing` class. Employ polymorphism to add additional functionality (namely adding `Food` object to LivingThing's `place`) before calling the `super()`'s `go_to_heaven()`.

```
>>> Base = Place("base")
>>> bear = Animal("bear", 20, 10)
>>> Base.add_object(bear)

>>> named_col(Base.get_objects())
['bear']
>>> bear.get_place()
Base
>>> bear.go_to_heaven()
bear went to heaven!
>>> named_col(Base.get_objects())
['bear meat']
```

Note: `named_col` is a helper function in `hungry_games.py` that prints out the name of objects stored in a list or tuple.