# CS 115 Homework 5: Robots versus Zombies

In this homework, you will be designing and implementing a classic turn-based role-playing game.

Like Homework 1, while there will be some technical requirements, we will generally leave you a large amount of creative freedom. Unlike Homework 1, however, this game will have many characters and items, each of which will be implemented via code, taking advantage of Object-Oriented Programming.

## Preliminaries

Before we go over what you have to do, we should begin by looking at the end goal, to see what we want to be able to accomplish.

Our end goal is to have an RPG, where a player character can fight a variety of enemies. One (incomplete) sample output of our final game follows:

```
Main character info: player (15 HP)
Here is your inventory:
- rusty axe (x1)

 Here are your enemies:
little cubebot (1 HP)
armor cube (2 HP)

 What is your next move?
Enter item name: rusty axe
Enter enemy number: 0
Your stats: player (15 HP)
Main character info: player (15 HP)
Here is your inventory:
- rusty axe (x2)

 Here are your enemies:
0: armor cube (2 HP)

 What is your next move?
Enter item name:
```

As you can see, the game will feature several characters. And we want our game to be able to handle arbitrary amounts and kinds of characters, so we will want to use objects to represent them. This leads to the `Character` class, where each instance of it will be used to store and represent one such character in the game.

Each `Character` will have their own distinct properties, but we want to be able to handle them uniformly. In particularly, while different characters might make different attacks, we want to be able to represent these attacks the same way, calling the same functions with the same interface. This lets us have just one piece of code that can handle vastly different types of characters, without having to write code to handle each type and interaction in a different unique way.

We accomplish this in two ways. The first is by creating a unified notion of `Move`s, which each use `Item`s.

Every `Character` has an `inventory` made up of `Item`s. Each item is some in-game tool that they can use during the encounter. This could be a weapon, a medkit, or anything else. We standardize this as the `Item` class, which is expanded further below. Furthermore, this inventory also doubles as loot: When a character is defeated, the character who defeated them gets to take the contents of the defeated character's inventory and add it to their own.

During each "turn", a `Character` chooses which `Item` to use, and who to use it on. To further standardize our behaviour, we encode this using the `Move` class. A `Move` specifies what `Item` is being used on which `Character`.

Finally, we further help standardize our game through the use of polymorphism: While different types of `Character` might have different properties, they generally share the same overarching behaviour. Thus, we choose to make these subclasses of our `Character` class, so that they can inherit the shared behaviour, while changing only a few small things.

**A note on provided code:**

To keep this homework at a reasonable scale, we have provided you with a bunch of starter code: We have given you the frameworks for a the three main classes, the entire implementation of the `Move` class, and the helper method `spawn_enemies`.

For a number of those methods, **you should not edit them**. The methods where the docstring is in all-caps should be kept as-is.

We also provide you the `main_game_loop` method. When this method is run, it actually runs the game for you. However, in order to get the right functionality, you need to have all the other methods implemented, as well as all the other features we ask for in this homework. But once you are done, you will be able to test your program and actually play it, using the method we have provided.

## Task 1: Completing the `Item` Class

Your first task is to finish the `Item` class. While we have already provided a lot of starter code for you, we still require you to finalize a few more parts of the class.

An `Item` has the following attributes, with the appropriate types:

- `name: str`
- `damage_points: int`
- `damage_type: str` ("physical", "electrical", "viral")
- `regeneration_points: int`
- `is_consumable: bool`

Your first task is to implement the constructor (`__init__`) for the `Item` class.

For example, the following code should instantiate a "rusty axe" with 1 damage point, 0 regeneration points, damage type "physical" and which is not consumable: `basic_weapon = Item("rusty axe", 1, 0, "physical", False)`.

Next, override the `__str__` implementation of `Item` to output a more descriptive string describing the item. The only requirements we have of your `__str__` method is that it must include 1) the `name` of the item, and 2) the number of damage points.

**Report question**

1. Instantiate a healing item of your choice using the Item constructor. Show how you would do that here in the report.
2. Copy and paste the result of calling `print` on your item with an overridden `__str__` into the report.

## Task 2: The `Character` class

Next, your job is to implement the `Character` class, representing a basic character in our game.

The `Character` class must have the following attributes:

- `name: str`: The name of the character.
- `health_points: int`: Their remaining health points.
- `inventory: dict`: A mapping of items of type `Item` to their quantities. By default, characters should start with a single rusty axe (as defined above).

Start by implementing the constructor of the `Character` class. This should takes in their starting health points `max_health_points` and their name `name` as input. For example, `small_enemy = Character("cubebot", 10)` results in a character named `"cubebot"` with 10 health points.

Next, you should override the `__str__` method of the class, such that it displays more useful information. In particular, your `__str__` class must display at least the following (although you may include more):

- The character's `name`.
- Their current number of `health_points`.

Afterwards, we want you to `override` the `__lt__` method. This is the magic method that is called when a "less than" comparison is made. In other words, `char1 < char2` results in a call to `char1.__lt__(char2)`.

You should implement this method such a character is "less" than another character if they have fewer *remaining* health points. You may reference `Item` as an example of how to do this.

As an aside, this also ensures that `min(characters)`, when called on a list of characters, returns the character with the lowest number of remaining health points.

After you have implemented these magic methods, we also need you to implement the following method. For each method, start by writing the docstring before implementing the function.

- `transfer_loot(self, loot)`: This should move all the `Item`s in `loot` into `self.inventory`. Note that `loot` is another dictionary, also mapping `Item`s to quantities. This means that moving it over involves updating the quantities in `self.inventory` accordingly, and then setting the quantities in `loot` to 0.

  *Hint:* You will find the `keys()` method of a dictionary helpful. This returns a collection of the keys of the dictionary, which can then be cast to a `list`.

- `perform_move(self, move)`: This method should have the calling character perform the given `Move`. The `Move` class is a data class, meaning it contains no methods, only being a convenient way to group up several attributes. In particular, a `Move` stores an `Item` in attribute `item` (what is being used...) and a `Character` in attribute `other_character` (... and on whom).

  To perform a given `Move move`, the performing character should execute the following steps:

  1. If the character does not have any of the relevant `move.item`, the function should exit, not changing anything. Otherwise,
  2. The character should subtract the item's `damage_points` from `other_character`'s `health_points`. Then,
  3. The character should add the item's `regeneration_points` to its own `health_points`.
  4. If the `Item` is consumable, then its count in the performing character's inventory should be decreaesd by 1.

  For example, applying a `Move` object with `basic_weapon` (which has `1` damage point and no other properties) to `small_enemy` would be done through the following code, and would have the corresponding effect:

```
>>> sample_move = Move(small_enemy, basic_weapon)
>>> print(small_enemy.health_points)
10
>>> player.perform_move(sample_move)
>>> print(small_enemy.health_points)
9
```

- `get_next_move(self, other_characters)`: This method should take in a list of other `Character` objects, and return the `Move` that this character intends to make.

  The default behaviour, which your `Character` object should follow, is to choose the `other_character` as the one in `other_characters` with the least remaining hit points, and to choose the `item` in the character's inventory with the most `damage_points`.

  *Hint:* This code can be written in surprisingly few lines. In particular, note that because we have overridden `__lt__` for both `Item` and `Character`, you can use `min` and `max` on collections of those types.

## Task 3: Inheritance

The classes we have let us now create characters with arbitrary inventories, and use them to perform arbitrary (legal) moves. However, our characters all behave the same way. The only differences come from having different inventories, but each character decides who they attack the same way, uses those items in the same way, and is otherwise indistinguishable.

Our solution is to use subclasses. We can create several subclasses of `Character`, which will inherit the majority of the functionality, but which will be able to implement some of these methods in their own unique way.

More specifically, we require you to implement the following subclasses: `Robot`, `Zombie`, and `PlayableCharacter`. All of these should inherit the methods of `Character` directly, without modification, except for the ones listed below.

### The `Robot` class

1. In addition to the rusty axe, `Robot` inventories start with a non-consumable `"shock baton"` item, which has 1 damage point of damage type "electrical".
2. For `get_next_move`, the `Robot` should always attack the first character in `other_characters`. The item it uses is up to you.
3. `perform_move` is the same as for `Character`, except that the robot deals double damage when using an item with damage type `"electrical"`.

### `Zombie` class

1. In addition to the rusty axe, `Zombie` inventories start with consumable `"brain grenade"` items. These deal 5 damage points of "viral" type. In particular, the Zombie must begin with at least 3 such grenades (but the exact amount is left open for you).
2. For `get_next_move`, the `Zombie` should always attack the first character in `other_characters`. The item it uses is up to you.
3. `perform_move` is the same as for `Character`, except that the zombie deals double damage when using an item with damage type `"viral"`.

### `PlayableCharacter` class

This class is special, as this is the class that will involve actual player interaction. Rather than the main game loop deciding what a player does, we will instead have the I/O be made in this subclass, instead.

The formal requirements are as follows:

1. The `PlayableCharacter` must be a subclass of `Character` or any other subclass of `Character`.
2. Starting inventories are completely up to you, depending on your desired game difficulty.
3. The `PlayableCharacter` has an additional method called `get_user_input(self, other_characters)`. This uses Python's `input` function to perform the following:
   (i) Print out the inventory of this character, and the list of enemies. (We do not require a specific format. Simply printing the unformatted list/dictionary is acceptable.)
   (ii) Queries the user for the name of an item.
   (iii) Queries the user for which character to attack. This can be done in any way you deem intuitive (for example, by specifying its index in the list).
   (iv) Returns a `Move` object using the chosen item on the chosen character.
   *Hint:* To make retrieval from the inventory by item name easier, we provided a helper function: `get_inventory_item_from_item_name(item_name, inventory)`, given an item name as a string, returns the corresponding `Item` object in `inventory`. If it cannot find a corresponding item, it returns None

You may assume that the user only enters valid inputs and do not need to implement error handling.

Here is an example interaction. Exactly what is printed and how is up to you, as long as it returns a valid `Move` according to the input.

```
>>> move = playable_character.get_user_input(self, other_characters)
Here is your inventory:
- rusty axe (x1)

 Here are your enemies:
little cubebot (10 HP)
armor cube (20 HP)

 What is your next move?
Enter item name: rusty axe
Enter enemy number: 1

>>> print(move)
Move:
    Item: Name: rusty axe Damage points: 1 Regen points: 0
    Target character: armor cube (20 HP)
```

4. The `PlayableCharacter`'s `get_next_move` method should call `get_user_input` to retrieve the user's next move.

**Report question**

Copy and paste the output of 1 sample interaction with your playable character. In particular, showcase what happens when you call `get_next_move` on that character. We have provided you some demo code at the bottom of the file which you can uncomment to try this.

### Custom character class (Optional)

You are encouraged but not required to add another character class to your game.

1. This class may inherit from any class or subclass of `Character`.
2. To avoid trivial subclasses, we request that at least one method be overridden from the paret class.
3. You may use any libraries **within the Python standard library** for your submission.

**Report question**

(Optional) Describe your custom class. In particular, what differs between your and the standard `Character` class? (at least one sentence)

## Task 4: Polymorphism

As your last task, we want you to implement the ability to have battles: To have two characters fight, inflicting damage to each other.

In particular, we want these battles to be independent of class: We want ot be able to use the same code regardless of if a `PlayableCharacter` fights a `Zombie`, a `Robot` fights a `Zombie`, or anything else.

The key idea will be to use polymorphism, which allows you to call the same functions of the superclass, but which runs the appropriately overridden methods of the subclasses.

While there are many different ways to implement the battle logic, we want you to begin by implementing the `standard_battle` method. This method has a signature of `standard_battle(main_character, enemies, enemy_that_will_attack)`, and must perform the following steps.

1. Assert that both the main character and the enemy that attacks have a positive amount of health points. If they don't, you should skip the following.

2. The `main_character` selects a move using `get_next_move(enemies)`.
3. The `main_character` applies that move to its chosen target.
4. If the target's health points reach 0, the `main_character` transfers all items from that target's inventory to the attacker.
5. Otherwise, `enemy_that_will_attack` selects a move using `get_next_move([main_character])` and performs it.
6. If the `main_character`'s health points reach 0, the attacking enemy transfers all items from the `main_character` 's inventory.

After implementing `standard_battle`, you are welcome to implement other battle logic. Implement a method that has a free-for-all battle, that has two teams duking it out, etc. But you must implement `standard_battle`, which must follow the logic above.

## Task 5: Playtesting, Testing And debugging

Although the tests in `test_hw5.py` test individual methods, they do not test how these methods *interact*. You can test the game by running `main_game_loop` and use the `PlayableCharacter` to try different attacks with the specific `Character`s and `Item`s you defined to make sure they behave as expected. As a bonus, get a friend to play your game. See if they can defeat all the enemies in your game, or break your game (cause a logical or runtime error). If you are unable to find a friend to play your game, *come to office hours to make one!*

**Report question**

1. Can you defeat all enemies in the game with the items and characters you have?
2. Have someone playtest your game. Were they able to find any errors? If so, describe *one* error and how you fixed your code.
3. Copy and paste the outputs that result from at least 3 rounds of running the `main_game_loop` in your game.

## Labor Log

We would like you to share your software development **process**. Especially, we want you to write about how you **debugged** your code, which is one of the most important parts of creating code.

In your labor log in `hw5_report.txt`, write about:

**Required components**:

1. How did you locate and correct any mistakes in your code? What strategies did you use? (At least 2 sentences)
2. Which strategies for correcting your code were the most and least effective? (At least 2 sentences)

In addition to the required reflection, we also suggest logging other important aspects of software development. This is not graded, but we hope it helps you raise awareness of your engineering process.

**Recommended optional components**:

1. Timing: when you started the homework, how many hours you worked on it
2. Environment: where you worked on the homework, with whom, ...
3. What you will continue and/or do differently for next homework
4. What you're proudest of?

## Submission Instructions

For this homework, you must submit the following files, with exactly the following names:

- `hw5.py`
- `hw5_report.txt`

**Remember to put your name and the Stevens Honor Pledge on every file you submit!** Failure to put the pledge can result in marks being deducted, with repeat penalties potentially getting stricter between assignments.

Submissions must be handled through Gradescope (accessible through the homework page on Canvas). Grades will be released through Gradescope once the homework is graded.

## Hints and Suggestions

- Spread function implementation out over multiple days.
- When debugging, start by testing on smaller examples before moving onto larger examples.
- You can run a few tests at a time using the `-k` flag in IDLE with the name of the functions you want to test. For example to test just the `perform_move` tests, go to IDLE, Custom Configurations, and type in `-k perform_move`. You can even include full test names to run one test at a time.
- Come to office hours to work on your homework!

## Notes and Reminders

- Keep track of the work you put into this homework, **especially in how you overcame difficulties**. Share in the "labor log" of the report.
- **Remember to include docstrings!** If your submission does not have docstrings, or has docstrings that are not related to your code, you will not get credit for them.
- Additionally, if your functions are more complicated than usual, you should add comments inside them explaining how they work.
- We have provided a test file, `test_hw5.py` for you to test your solutions to ensure that files are named correctly and to check the correctness of your code. Try out many different cases to verify that your code runs. Our provided test file may not be comprehensive, and so you should ensure to test your code on many other cases as well.
- Code that doesn't compile (throws a `SyntaxError` on being run) will receive a 0 for the assignment! Make sure you submit code that can be run, even if it does not pass all the test cases.
- **Don't forget to add your name and pledge!**