Slay the Spire

Assignment 2
Semester 1, 2023
CSSE1001/CSSE7030

Due date: 28th April 2023 16:00 GMT+10

1 Introduction

Slay the Spire is a rogue-like deck building card game in which a player must build a deck of cards, which they use during encounters with monsters. Details of the original *Slay the Spire* game can be found here. In Assignment 2, you will create an object-oriented text-based game inspired by (though heavily simplified and altered from) *Slay the Spire*[1].

You are required to implement a collection of classes and methods as specified in Section 5 of this document. Your program's output must match the expected output *exactly*; minor differences in output (such as whitespace or casing) *will* cause tests to fail, resulting in *zero marks* for those tests. Any changes to this document will be listed in a changelog on Blackboard.

2 Getting Started

Download `a2.zip` from Blackboard — this archive contains the necessary files to start this assignment. Once extracted, the `a2.zip` archive will provide the following files / directories:

`a2.py`
: The game engine. This is *the only file you submit and modify. Do not* make changes to any other files.

`a2_support.py`
: Support code to assist in more complex parts of the assignment, and to handle randomness. Note that when implementing random behaviour, you **must** use the support functions, rather than calling functions from random yourself.

`games`
: A folder containing several text files of playable games.

`game_examples`
: A folder containing example output from running the completed assignment.

3 Gameplay

At the beginning of the game, the user selects a player character; different player characters have different advantanges and disadvantages, such as starting with higher HP or a better deck of cards. The user then selects a game file, which specifies the encounters that they will play. After this, gameplay can begin.

During gameplay, the player users a deck of cards to work through a series of encounters with monsters. Each encounter involves between one and three monsters, which the player must battle in parallel over a series of turns. At the start of each turn the user draws 5 cards at random from their deck into their hand. Each card costs between 0 and 3 energy point to play. The user may play as many cards as they like from their hand during their turn provided they still have the energy points required to play the requested cards. The user opts to end their turn when they are finished playing cards, at which point the monsters in the encounter each take an action (which may affect the player's HP or other stats, or the monster's own stats). When a card is played it is immediately sent to the player's discard pile. At the end of a turn, all cards in the players hand (regardless of whether they were played that turn) are sent to the discard pile. Cards in the discard pile cannot be drawn

[1] Where the behaviour of the original game differs from this specification, implement the assignment as per the specification (not to the original behaviour of the game).

until the entire deck has been drawn, at which point the deck is replenished with all the cards from the discard pile. An encounter ends when either the player has killed all monsters (reduced their HP to 0) or when the monsters have killed the player (reduced the player's HP to 0). If the player wins an encounter, an encounter win message is printed and the next encounter begins. If no more encounters remain, the game terminates with a game win message, and if the player loses an encounter, the program terminates with a loss message. See `a2_support.py` for the relevant messages. You can find examples of gameplay in the `game_examples` folder provided in `a2.zip`. For more details on the behaviour of `main`, see Section 5.4.

4 Class overview and relationships

You are *required* to implement a number of classes, as well as a main function. You should develop the classes first and ensure they all work (at minimum, ensure they pass the Gradescope tests) before beginning work on the main function. The class diagram in Figure 1 provides an overview of *all* of these classes, and the basic relationships between them. The details of these classes and their methods are described in depth in Section 5.

- Hollow-arrowheads indicate *inheritance* (i.e. the "is-a" relationship).
- Dotted arrows indicates *composition* (i.e. the "has-a" relationship). An arrow marked with 1-1 denotes that each instance of the class at the base of the arrow contains exactly one instance of the class at the head of the arrow. An arrow marked with 1-n denotes that each instance of the class at the base of the arrow may contain many instances of the class at the head of the arrow. E.g. an `Encounter` instance may contain between 1 and 3 `Monster` instances, but only one `Player` instance.
- Blue classes are *abstract* classes. You should only ever instantiate the green classes in your program, though you should instantiate the blue classes to test them before beginning work on their subclasses.

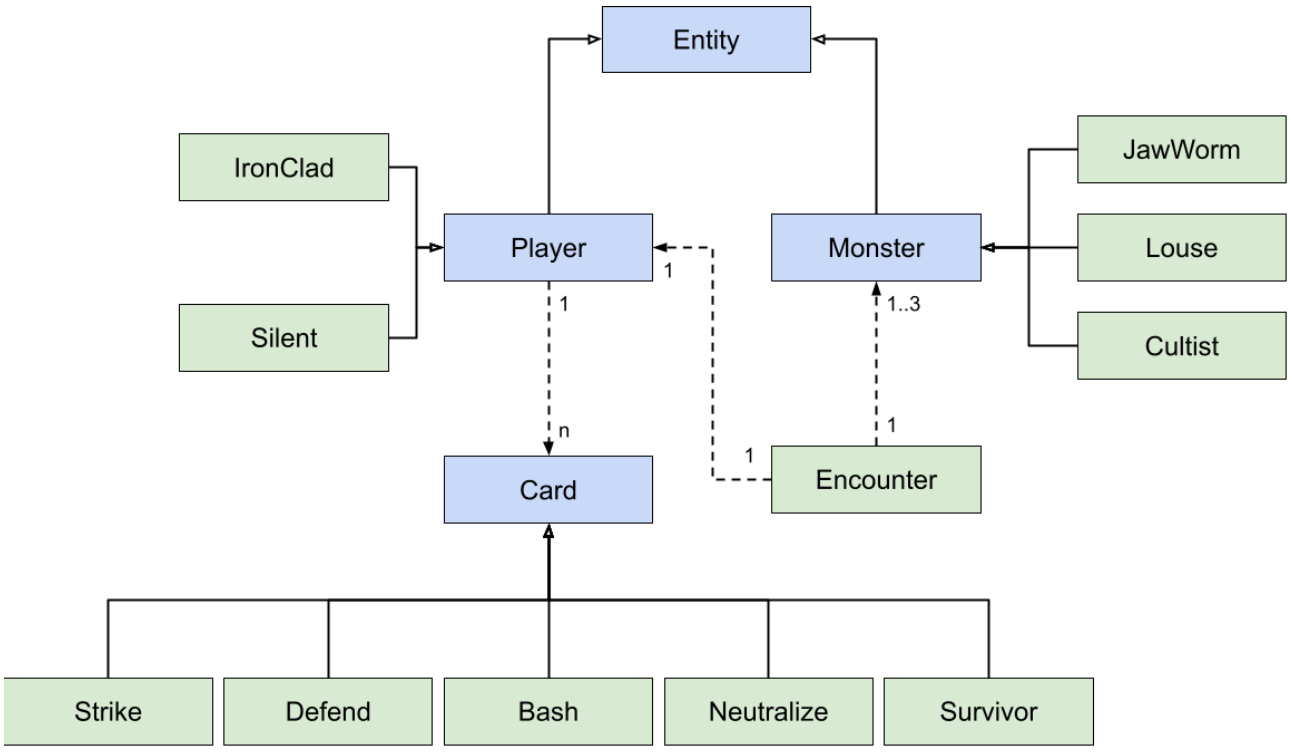

Figure 1: Basic class relationship diagram for the classes which need to be implemented for this assignment.

5 Implementation

This section outlines the classes, methods, and functions that you are *required* to implement as part of your assignment. It is recommended that you implement the classes in the order in which they are described. Ensure each class behaves as per the examples (and where possible, the Gradescope tests) before moving on to the next class.

5.1 Cards

Cards are used by the player during encounters to attack monsters, defend, or apply status modifiers. When implementing this class, it is not necessary that you yet understand the mechanics of *how* these effects will be applied. You will handle this later in your implementation. Card classes simply provide information about the effects each type of card has; they are not responsible for directly causing these effects.

All instantiable cards inherit from the abstract `Card` class, and should inheret the default `Card` behaviour except where specified in the descriptions of each specific type of card.

Card *(abstract class)*

An abstract class from which all instantiable types of cards inheret. Provides the default card behaviour, which can be inhereted or overwritten by specific types of cards. The `__init__` method for all cards do not take any arguments beyond `self`.

`get_damage_amount(self) -> int` *(method)*

Returns the amount of damage this card does to its target (i.e. the opponent it is played on). By default, the damage done by a card is 0.

`get_block(self) -> int` *(method)*

Returns the amount of block this card adds to its user. By default, the block amount provided by a card is 0.

`get_energy_cost(self) -> int` *(method)*

Returns the amount of energy this card costs to play. By default, the energy cost should be 1.

`get_status_modifiers(self) -> dict[str, int]` *(method)*

Returns a dictionary describing each status modifiers applied when this card is played. By default, no status modifiers are applied; that is, this method should return an empty dictionary in the abstract `Card` class.

`get_name(self) -> str` *(method)*

Returns the name of the card. In the `Card` superclass, this is just the string 'Card'.

`get_description(self) -> str` *(method)*

Returns a description of the card. In the `Card` superclass, this is just the string 'A card.'.

`requires_target(self) -> bool` *(method)*

Returns True if playing this card requires a target, and False if it does not. By default, a card does require a target.

`__str__(self) -> str` *(method)*

Returns the string representation for the card, in the format `'{Card name}: {Card description}'`.

`__repr__(self) -> str` *(method)*

Returns the text that would be required to create a new instance of this class identical to `self`.

Examples

```
>>> card = Card()
>>> card.get_damage_amount()
0
>>> card.get_block()
0
>>> card.get_energy_cost()
1
>>> card.get_status_modifiers()
{}
```

```
>>> card.get_name()
'Card'
>>> card.get_description()
'A card.'
>>> card.requires_target()
True
>>> str(card)
'Card: A card.'
>>> card
Card()
```

Strike *(class)*

Inherits from Card
Strike is a type of Card that deals 6 damage to its target. It costs 1 energy point to play.

Examples

```
>>> strike = Strike()
>>> print(strike.get_damage_amount(), strike.get_block(), strike.get_energy_cost())
6 0 1
>>> strike.get_name()
'Strike'
>>> strike.get_description()
'Deal 6 damage.'
>>> strike.requires_target()
True
>>> str(strike)
'Strike: Deal 6 damage.'
>>> strike
Strike()
```

Defend *(class)*

Inherits from Card
Defend is a type of Card that adds 5 block to its user. Defend does not require a target. It costs 1 energy point to play.

Examples

```
>>> defend = Defend()
>>> print(defend.get_damage_amount(), defend.get_block(), defend.get_energy_cost())
0 5 1
>>> defend.get_name()
'Defend'
>>> defend.get_description()
'Gain 5 block.'
>>> defend.requires_target()
False
>>> str(defend)
'Defend: Gain 5 block.'
>>> defend
Defend()
```

Bash *(class)*

Inherits from Card
Bash is a type of Card that adds 5 block to its user and causes 7 damage to its target. It costs 2 energy points to play.

Examples

```
>>> bash = Bash()
>>> print(bash.get_damage_amount(), bash.get_block(), bash.get_energy_cost())
7 5 2
>>> bash.get_name()
'Bash'
>>> bash.get_description()
'Deal 7 damage. Gain 5 block.'
>>> bash.requires_target()
True
>>> str(bash)
'Bash: Deal 7 damage. Gain 5 block.'
>>> bash
Bash()
```

Neutralize *(class)*

Inherits from Card
Neutralize is a type of card that deals 3 damage to its target. It also applies status modifiers to its target; namely, it applies 1 weak and 2 vulnerable. Neutralize does not cost any energy points to play.

Examples

```
>>> neutralize = Neutralize()
>>> print(neutralize.get_damage_amount(), neutralize.get_block(), neutralize.get_energy_cost())
3 0 0
>>> neutralize.get_status_modifiers()
{'weak': 1, 'vulnerable': 2}
>>> neutralize.get_name()
'Neutralize'
>>> neutralize.get_description()
'Deal 3 damage. Apply 1 weak. Apply 2 vulnerable.'
>>> str(neutralize)
'Neutralize: Deal 3 damage. Apply 1 weak. Apply 2 vulnerable.'
>>> neutralize
Neutralize()
```

Survivor *(class)*

Inherits from Card
Survivor is a type of card that adds 8 block and applies 1 strength to its user. Survivor does not require a target.

Examples

```
>>> survivor = Survivor()
>>> print(survivor.get_damage_amount(), survivor.get_block(), survivor.get_energy_cost())
0 8 1
>>> survivor.get_status_modifiers()
{'strength': 1}
```

```
>>> survivor.requires_target()
False
>>> survivor.get_name()
'Survivor'
>>> survivor.get_description()
'Gain 8 block and 1 strength.'
>>> str(survivor)
'Survivor: Gain 8 block and 1 strength.'
>>> survivor
Survivor()
```

5.2 Entities

Entities in the game include the player and enemies (monsters). All entities have:

- Health points (HP): This starts at the maximum HP for the entity, and may decrease over the course of one or more encounters. An entity is defeated when its HP is reduced to 0.
- Block: This is the amount of defense the entity has. When an entity is attacked, damage is applied to the block first. Only once the block has been reduced to 0 will any remaining damage be caused to the entity's HP. For example, if an entity with 10 HP and 5 block is attacked with a damage of 8, their block will be reduced to 0 and their HP reduced to 7.
- Strength: The amount of additional strength this entity has. When an entity plays a card that causes damage to a target, the damage caused will increase by 1 for each strength point the entity has. Strength does not wear off until the end of an encounter.
- Weak: The number of turns for which this entity is weak. If an entity is weak on a given turn, all cards played by the entity that cause damage will cause 25% less damage.
- Vulnerable: The number of turns for which this entity is vulnerable. If an entity is vulnerable on a turn, damage caused to it will be increased by 50%.

In this assignment you must implement an abstract Entity class, which provides the base entity functionality that all entities inherit. Except where specified, entities have the default behaviour inherited from the Entity superclass.

Entity *(abstract class)*

Abstract base class from which all entities inherit.

`__init__(self, max_hp: int) -> None` *(method)*

Sets up a new entity with the given `max_hp`. An entity starts with the maximum amount of HP it can have. Block, strength, weak, and vulnerable all start at 0.

`get_hp(self) -> int` *(method)*

Returns the current HP for this entity.

`get_max_hp(self) -> int` *(method)*

Returns the maximum possible HP for this entity.

`get_block(self) -> int` *(method)*

Returns the amount of block for this entity.

`get_strength(self) -> int` *(method)*

Returns the amount of strength for this entity.

`get_weak(self) -> int` *(method)*

Returns the number of turns for which this entity is weak.

`get_vulnerable(self) -> int` *(method)*

Returns the number of turns for which this entity is vulnerable.

`get_name(self) -> str` *(method)*

Returns the name of the entity. The name of an entity is just the name of the most specific class it belongs to.

`reduce_hp(self, amount: int) -> None` *(method)*

Attacks the entity with a damage of `amount`. This involves reducing block until the amount of damage has been done or until block has reduced to zero, in which case the HP is reduced by the remaining amount. For example, if an entity has 20 HP and 5 block, calling `reduce_hp` with an amount of 10 would result in 15 HP and 0 block. HP cannot go below 0.

`is_defeated(self) -> bool` *(method)*

Returns True if the entity is defeated, and False otherwise. An entity is defeated if it has no HP remaining.

`add_block(self, amount: int) -> None` *(method)*

Adds the given amount to the amount of block this entity has.

`add_strength(self, amount: int) -> None` *(method)*

Adds the given amount to the amount of strength this entity has.

`add_weak(self, amount: int) -> None` *(method)*

Adds the given amount to the amount of weak this entity has.

`add_vulnerable(self, amount: int) -> None` *(method)*

Adds the given amount to the amount of vulnerable this entity has.

`new_turn(self) -> None` *(method)*

Applies any status changes that occur when a new turn begins. For the base Entity class, this involves setting block back to 0, and reducing weak and vulnerable each by 1 if they are greater than 0.

`__str__(self) -> str` *(method)*

Returns the string representation for the entity in the format `'{entity name}: {current HP}/{max HP} HP'`.

`__repr__(self) -> str` *(method)*

Returns the text that would be required to create a new instance of this class identical to `self`.

Examples

```
>>> entity = Entity(20)
>>> entity.get_name()
'Entity'
>>> print(entity.get_hp(), entity.get_max_hp(), entity.get_block())
20 20 0
>>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
0 0 0
>>> entity.reduce_hp(2)
>>> entity.get_hp()
18
>>> entity.add_block(5)
>>> entity.get_block()
5
>>> entity.reduce_hp(10)
>>> entity.get_hp()
```

```
13
>>> entity.get_block()
0
>>> entity.is_defeated()
False
>>> entity.add_strength(2)
>>> entity.add_weak(3)
>>> entity.add_vulnerable(4)
>>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
2 3 4
>>> entity.add_block(5)
>>> entity.get_block()
5
>>> entity.new_turn()
>>> print(entity.get_strength(), entity.get_weak(), entity.get_vulnerable())
2 2 3
>>> entity.get_block()
0
>>> entity.get_hp()
13
>>> entity.reduce_hp(15)
>>> entity.get_hp()
0
>>> entity.is_defeated()
True
```

Player *(abstract class)*

Inherits from Entity
A Player is a type of entity that the user controls. In addition to regular entity functionality, a player also has energy and cards. Player's must manage three sets of cards; the deck (cards remaining to be drawn), their hand (cards playable in the current turn), and a discard pile (cards that have been played already this encounter).

`__init__(self, max_hp: int, cards: list[Card] | None = None) -> None` *(method)*

In addition to executing the initializer for the Entity superclass, this method must initialize the player's energy which starts at 3, as well as three lists of cards (deck, hand, and discard pile). If the `cards` parameter is not None, the deck is initialized to be `cards`. Otherwise, it should be initialized as an empty list. The players hand and discard piles start as empty lists.

`get_energy(self) -> int` *(method)*

Returns the amount of energy the user has remaining.

`get_hand(self) -> list[Card]` *(method)*

Returns the players current hand.

`get_deck(self) -> list[Card]` *(method)*

Returns the players current deck.

`get_discarded(self) -> list[Card]` *(method)*

Returns the players current discard pile.

`start_new_encounter(self) -> None` *(method)*

This method adds all cards from the player's discard pile to the end of their deck, and sets the discard pile to be an empty list. A pre-condition to this method is that the player's hand should be empty when it is called.

`end_turn(self) -> None` *(method)*

This method adds all remaining cards from the player's hand to the end of their discard pile, and sets their hand back to an empty list.

`new_turn(self) -> None` *(method)*

This method sets the player up for a new turn. This involves everything that a regular entity requires for a new turn, but also requires that the player be dealt a new hand of 5 cards, and energy be reset to 3.

Note: You **must** use the `draw_cards` function from `a2_support.py` to achieve dealing the player new cards. You must call the `draw_cards` function exactly **once** from this method. Do **not** use the `select_cards` method from `a2_support.py` to achieve the random selection of cards.

`play_card(self, card_name: str) -> Card | None` *(method)*

Attempts to play a card from the player's hand. If a card with the given name exists in the player's hand and the player has enough energy to play said card, the card is removed from the player's hand and added to the discard pile, the required energy is deducted from the player's energy, and the card is returned. If no card with the given name exists in the player's hand, or the player doesn't have enough energy to play the requested card, this function returns None.

Examples
Note: this example section, and many that follow, should be completely replicable if you start a new IDLE shell (i.e. re-run your program before entering the commands). If you run multiple example sections without restarting the IDLE shell in between, the cards allocated to the player's hand during `new_turn` may differ from what is shown in the examples.

```
>>> player = Player(50, [Strike(), Strike(), Strike(), Defend(), Defend(), Defend(), Bash()])
>>> player.get_name()
'Player'
>>> player.get_hp()
50
>>> player.get_energy()
3
>>> print(player.get_hand(), player.get_discarded())
[] []
>>> player.get_deck()
[Strike(), Strike(), Strike(), Defend(), Defend(), Defend(), Bash()]
>>> player.new_turn()
>>> player.get_hand()
[Strike(), Defend(), Strike(), Strike(), Bash()]
>>> player.get_deck()
[Defend(), Defend()]
>>> player.get_discarded()
[]
>>> player.play_card('Bash')
Bash()
>>> player.get_hand()
[Strike(), Defend(), Strike(), Strike()]
>>> player.get_deck()
[Defend(), Defend()]
>>> player.get_discarded()
[Bash()]
>>> player.end_turn()
>>> player.get_hand()
[]
>>> player.get_deck()
[Defend(), Defend()]
>>> player.get_discarded()
[Bash(), Strike(), Defend(), Strike(), Strike()]
>>> player.reduce_hp(10)
```

```
>>> str(player)
'Player: 40/50 HP'
```

IronClad *(class)*

Inherits from Player
IronClad is a type of player that starts with 80 HP. IronClad's deck contains 5 Strike cards, 4 Defend cards, and 1 Bash card. The `__init__` method for IronClad does not take any arguments beyond `self`.

Examples
Note: restart your IDLE shell before running these examples to replicate the cards drawn into the hand.

```
>>> iron_clad = IronClad()
>>> iron_clad.get_name()
'IronClad'
>>> str(iron_clad)
'IronClad: 80/80 HP'
>>> iron_clad.get_hp()
80
>>> iron_clad.get_hand()
[]
>>> iron_clad.get_deck()
[Strike(), Strike(), Strike(), Strike(), Strike(), Defend(), Defend(), Defend(), Defend(), Bash()]
>>> iron_clad.new_turn()
>>> iron_clad.get_hand()
[Strike(), Strike(), Strike(), Bash(), Strike()]
>>> iron_clad.get_deck()
[Strike(), Defend(), Defend(), Defend(), Defend()]
```

Silent *(class)*

Inherits from Player
Silent is a type of player that starts with 70 HP. Silent's deck contains 5 Strike cards, 5 Defend cards, 1 Neutralize card, and 1 Survivor card. The `__init__` method for Silent does not take any arguments beyond `self`.

Examples
Note: restart your IDLE shell before running these examples to replicate the cards drawn into the hand.

```
>>> silent = Silent()
>>> silent.get_name()
'Silent'
>>> str(silent)
'Silent: 70/70 HP'
>>> silent.get_hp()
70
>>> silent.get_hand()
[]
>>> silent.get_deck()
[Strike(), Strike(), Strike(), Strike(), Strike(), Defend(), Defend(), Defend(), Defend(),
Defend(), Neutralize(), Survivor()]
>>> silent.new_turn()
>>> silent.get_hand()
[Strike(), Strike(), Strike(), Neutralize(), Defend()]
>>> silent.get_deck()
[Strike(), Strike(), Defend(), Defend(), Defend(), Defend(), Survivor()]
```

Monster *(abstract class)*

Inherits from Entity
A Monster is a type of entity that the user battles during encounters. In addition to regular entity functionality, each monster also has a unique id, and an action method that handles the effects of the monster's action on itself, and describes the effect the monster's action would have on its target.

During gameplay, each monster in an encounter will get to take one action per turn (see Section 5.3 for more details). When implementing the `Monster` class and subclasses, however, it is not important to understand how the turn-taking system or how the monster's action effects will be applied to the player. The monster classes are only responsible for applying the effects of a monster's actions to the monster itself (e.g. some monsters will increase their own stats during their action) and returning a dictionary describing how the action would affect the player.

`__init__(self, max_hp: int) -> None` *(method)*

Sets up a new monster with the given maximum HP and a unique id number. The first monster created should have an id of 0, the second monster created should have an id of 1, etc.

`get_id(self) -> int` *(method)*

Returns the unique id number of this monster.

`action(self) -> dict[str, int]` *(method)*

Performs the current action for this monster, and returns a dictionary describing the effects this monster's action should cause to its target. In the abstract Monster superclass, this method should just raise a NotImplementedError. This method must be overwritten by the instantiable subclasses of Monster, with the strategies specific to each type of monster.

Examples
Note: you may need to restart your IDLE shell before running these examples to replicate the monster IDs.

```
>>> monster = Monster(20)
>>> monster.get_id()
0
>>> another_monster = Monster(3)
>>> another_monster.get_id()
1
>>> monster.get_id()
0
>>> monster.action()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
...
raise NotImplementedError
NotImplementedError
>>> monster.get_name()
'Monster'
>>> monster.reduce_hp(10)
>>> str(monster)
'Monster: 10/20 HP'
```

Louse *(class)*

Inherits from Monster
The Louse's action method simply returns a dictionary of `{'damage': amount}`, where `amount` is an amount between 5 and 7 (inclusive), randomly generated when the Louse instance is created. You **must** use the `random_louse_amount` function from `a2_support.py` to generate the amount each louse will attack. You **must** only call the `random_louse_amount` function once for each Louse instance, when the louse is created.

Examples
Note: you may need to restart your IDLE shell before running these examples to replicate the monster IDs.

```
>>> louse = Louse(20)
>>> str(louse)
'Louse: 20/20 HP'
>>> louse.get_id()
0
>>> louse.action()
{'damage': 6}
>>> louse.action() # should be the same amount of damage
{'damage': 6}
>>> louse.action()
{'damage': 6}
>>> another_louse = Louse(30)
>>> another_louse.action()
{'damage': 7}
>>> another_louse.get_id()
1
```

Cultist *(class)*

Inherits from Monster
The action method for Cultist should return a dictionary of `{'damage': damage_amount, 'weak': weak_amount}`. For each Cultist instance, `damage_amount` is 0 the first time action is called. For each subsequent call to `action`, `damage_amount = 6 + num_calls`, where `num_calls` is the number of times the `action` method has been called on this specific Cultist instance. The `weak_amount` alternates between 0 and 1 each time the `action` method is called on a specific Cultist instance, starting at 0 for the first call.

Examples

```
>>> cultist = Cultist(20)
>>> cultist.action()
{'damage': 0, 'weak': 0}
>>> cultist.action()
{'damage': 7, 'weak': 1}
>>> cultist.action()
{'damage': 8, 'weak': 0}
>>> cultist.action()
{'damage': 9, 'weak': 1}
>>> cultist.action()
{'damage': 10, 'weak': 0}
>>> another_cultist = Cultist(30)
>>> another_cultist.action()
{'damage': 0, 'weak': 0}
```

JawWorm *(class)*

Inherits from Monster
Each time `action` is called on a JawWorm instance, the following effects occur:

- Half of the amount of damage the jaw worm has taken so far (rounding up) is added to the jaw worm's own block amount.
- Half of the amount of damage the jaw worm has taken so far (rounding down) is used for damage to the target.

The amount of damage taken so far is the difference between the jaw worm's maximum HP and its current HP.

Examples

```
>>> jaw_worm = JawWorm(20)
>>> jaw_worm.get_block()
0
>>> jaw_worm.action() # Should generate 0 at the start as jaw_worm hasn't lost any HP
{'damage': 0}
>>> jaw_worm.get_block()
0
>>> jaw_worm.reduce_hp(11)
>>> jaw_worm.action()
{'damage': 5}
>>> jaw_worm.get_block()
6
>>> jaw_worm.reduce_hp(5)
>>> jaw_worm.get_hp()
9
>>> jaw_worm.get_block()
1
>>> jaw_worm.reduce_hp(5)
>>> jaw_worm.get_hp()
5
>>> jaw_worm.action()
{'damage': 7}
>>> jaw_worm.get_block()
8
```

5.3 Encounters

Encounter *(class)*

Each encounter in the game is represented as an instance of the `Encounter` class. This class manages one player and a set of 1 to 3 monsters, and facilitates the interactions between the player and monsters. This section describes the methods that must be implemented as part of the `Encounter` class.

`__init__(self, player: Player, monsters: list[tuple[str, int]]) -> None` *(method)*

The initializer for an encounter takes the player instance, as well as a list of tuples describing the monsters in the encounter. Each tuple contains the name (type) of monster and the monster's max HP. The initializer should use these tuples to construct monster instances in the order in which they are described. The initializer should also tell the player to start a new encounter (see `Player.start_new_encounter`), and should also start a new turn (see `start_new_turn` below for a description).

`start_new_turn(self) -> None` *(method)*

This method sets it to be the player's turn (i.e. the player is permitted to attempt to apply cards) and called `new_turn` on the player instance.

`end_player_turn(self) -> None` *(method)*

This method sets it to not be the player's turn (i.e. the player is not permitted to attempt to apply cards), and ensures all cards remaining in the player's hand move into their discard pile. This method also calls the `new_turn` method on all monster instances remaining in the encounter.

`get_player(self) -> Player` *(method)*

Returns the player in this encounter.

`get_monsters(self) -> list[Monster]` *(method)*

Returns the monsters remaining in this encounter.

`is_active(self) -> bool` *(method)*

Returns True if there are monsters remaining in this encounter, and False otherwise.

`player_apply_card(self, card_name: str, target_id: int | None = None) -> bool` *(method)*

This method attempts to apply the first card with the given name from the player's hand (where relevant, the target for the card is specified by the given `target_id`). The steps executed by this method are as follows:

1. Return False if the application of the card is invalid for any of the following reasons:
   - If it is not the player's turn
   - If the card with the given name requires a target but no target was given
   - If a target was given but no monster remains in this encounter with that id.
2. The player attempts to play a card with the given name. If this is not successful (i.e. the card did not exist in the player's hand, the player didn't have enough energy, or the card name doesn't map to a card), this function returns False. Otherwise, the function should execute the remaining steps.
3. Any block and strength from the card should be added to the player.
4. If a target was specified:
   (a) Any vulnerable and weak from the card should be applied to the target.
   (b) Damage is calculated and applied to the target. The base damage is the amount of damage caused by the card, plus the strength of the player. If the target is vulnerable (i.e. their vulnerable stat is non-zero) the damage should be multiplied by 1.5 and if the player is weak (i.e. their weak stat is non-zero) it should be multiplied by 0.75. The damage amount should be converted to an int before being applied to the target. Int conversions should round down (note that this is the default behaviour of type casting to an int).
   (c) If the target has been defeated, remove them from the list of monsters.
5. Return True to indicate that the function executed successfully.

Note: The order of these steps is important. For example, status modifiers such as strength, vulnerable, and weak should be applied *before* calculating the amount of damage to apply to a target. Checking all conditions that would make a card invalid in step 1 must occur before step 2, so as not to reduce the player's energy if the card application is invalid.

`enemy_turn(self) -> None` *(method)*

This method attempts to allow all remaining monsters in the encounter to take an action. This method immediately returns if it is the player's turn. Otherwise, each monster takes a turn (in order) as follows:

1. The monster attempts its action (see the `action` method in the `Monster` class).
2. Any weak and vulnerable generated by the monster's action are added to the player.
3. Any strength generated by the monster's action are added to the monster.
4. Damage is calculated and applied to the target. The base damage is the amount of damage caused by the monster's action, plus the strength of the monster. If the player is vulnerable the damage should be multiplied by 1.5 and if the monster is weak it should be multiplied by 0.75. The damage amount should be converted to an int before being applied to the player.

Once all monster's have played an action, this method starts a new turn.

5.4 main function

The main function is run when your file is run, and manages overall gameplay. For examples of the behaviour of the main function, and the exact prompts required, see the `gameplay/` folder provided as part of `a2.zip`. You will also find the constants in `a2_support.py` useful for some of the prompts (you are encouraged to also

define your own global constants where relevant). The main function should do the following:

1. Prompt the user for the type of player they want (either 'ironclad' or 'silent') and create the relevant player instance. You will use this same player instance for the entire game.

2. Prompt the user for a game file. A function to assist in reading this file can be found in `a2_support.py`.

3. For each encounter described in the file:

   (a) Start a new encounter with this set of monsters and display it.

   (b) Until the encounter is no longer active (or until the game has terminated due to the player losing), the user should be continually prompted for moves. Table 1 describes the available moves, and the behaviour they should cause. If at the end of a move, the player has won the encounter, their turn should be ended before starting the next encounter. If you do not end the player's turn before starting the new encounter, the player may lose access to the cards in their hand, and you may find the cards selected by your program differ from the expected outputs.

4. If the player makes it through all encounters, the program should terminate with the game win message.

You may assume your program will not be tested with incorrect inputs, except that the play command may be tested with invalid `card_name` and/or invalid or missing `target_id`.

Move name	Behaviour
'end turn'	When the user enters this command, the player's turn should end, and the enemy turn should be run. If the player is defeated after the enemy turn, the game should terminate with the game lose message. Otherwise the resulting encounter state should be displayed.
'inspect {deck | discard}'	When the user enters 'inspect deck', the player's deck should be printed. When the user enters 'inspect discard', the player's discard pile should be printed.
'describe {card_name}'	When the user enters this command, the description for the card with the given card_name should be printed. Note that the description of the card should be printed even if the player does not have an instance of the requested card.
'play {card_name}' or 'play {card_name} {target_id}'	Attempts to play a card. If the card application fails for any reason, the card failure message should be printed. Otherwise if the card is successfully applied, the resulting encounter state should be printed. You may assume that if a target_id is entered it will be an integer, but not that a monster with that ID exists in the encounter.

Table 1: The four types of commands a user can input at the prompt for a move.

6 Postgraduate Task

Postgraduate students are required to implement two additional cards and a third type of player. This section provides a brief overview of the player and cards required. As this is an advanced task, you are expected to determine how to design these classes, how to test them effectively, and how to integrate them into main yourself.

6.1 Eruption

Eruption is a type of card which costs 2 energy points to play, and deals 9 damage to its target.

6.2 Vigilance

Vigilance is a type of card which costs 2 energy points to play and adds 8 block and 1 strength to its user. It does not require a target.

6.3 Watcher

Watcher is a type of player which starts with 72 HP and a deck containing 4 Strike cards, 4 Defend cards, 1 Eruption card, and 1 Vigilance card. If the user enters 'watcher' at the prompt for player type, the player used should be a `Watcher` instance.

7 Assessment and Marking Criteria

This assignment assesses course learning objectives:

1. apply program constructs such as variables, selection, iteration and sub-routines,
2. apply basic object-oriented concepts such as classes, instances and methods,
3. read and analyse code written by others,
4. analyse a problem and design an algorithmic solution to the problem,
5. read and analyse a design and be able to translate the design into a working program, and
6. apply techniques for testing and debugging.

7.1 Marking Breakdown

Your total grade for this assessment piece will be a combination of your functionality and style marks. For this assignment, functionality and style have equal weighting, meaning you should be devoting at least as much time towards proper styling of your code as you do trying to make it functional.

7.2 Functionality Marking

Your program's functionality will be marked out of a total of 50 marks. As in assignment 1, your assignment will be put through a series of tests and your functionality mark will be proportional to the number of tests you pass. You will be given a *subset* of the functionality tests before the due date for the assignment. You may receive partial marks within each class for partially working methods, or for implementing only a few classes. Note that you do not need to implement the `main` function or `Encounter` class in order to pass this assignment; implementing all classes except `Encounter` in a well-designed, well-styled way is sufficient to earn a passing grade for this assignment.

You need to perform your *own* testing of your program to make sure that it meets *all* specifications given in the assignment. Only relying on the provided tests is likely to result in your program failing in some cases and you losing some functionality marks. Note: Functionality tests are automated, so string outputs need to match *exactly* what is expected.

Your program must run in the Python interpreter (the IDLE environment). Partial solutions will be marked, but if there are errors in your code that cause the interpreter to fail to execute your program, you will get zero for functionality marks. If there is a part of your code that causes the interpreter to fail, comment out the code so that the remainder can run. Your program must run using the Python 3.11 interpreter. If it runs in another environment (e.g. Python 3.10 or PyCharm) but not in the Python 3.11 interpreter, you will get zero for the functionality mark.

If your program cannot not run on Gradescope, you will receive **no marks** for functionality. It is your responsibility to upload to Gradescope in time to debug any issues that may cause Gradescope to be unable to run your submission. Tutors will not fix any aspect of your code (including file names).

7.3 Style Marking

The style of your assignment will be assessed by a tutor. The style mark will also be out of 50. The key consideration in marking your code style is whether the code is easy to understand and demonstrates understanding of object-oriented programming concepts. In this assignment, your code style will be assessed against the following criteria.

- Readability
  - Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
  - Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.) The main reason for this restriction is that most people who follow the *Hungarian Notation* convention, use it poorly

(including Microsoft).

  - Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.

- Documentation

  - Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.

  - Informative Docstrings: Every class, method and function should have a docstring that summarises its purpose. This includes describing parameters and return values so that others can understand how to use the method or function correctly.

  - Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method or function, the logic should usually be clear from the code and docstring. For long or complex methods or functions, each logical block should have an in-line comment describing its logic.

Structure will be assessed as to how well your code design conforms to good object-oriented programming practices.

- Object-Oriented Program Structure

  - Classes & Instances: Objects are used as entities to which messages are sent, demonstrating understanding of the differences between classes and instances.

  - Encapsulation: Classes are designed as independent modules with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.

  - Inheritance & Polymorphism: Subclasses are designed as specialised versions of their superclasses. Subclasses extend the behaviour of their superclass without re-implementing behaviour, or breaking the superclass behaviour or design. Subclasses redefine behaviour of appropriate methods to extend the superclasses' type. Subclasses do not break their superclass' interface.

- Algorithmic Logic

  - Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method or function.

  - Variable Scope: Variables should be declared locally in the method or function in which they are needed. Attributes should be declared clearly within the `__init__` method. Class variables are avoided, except where they simplify program logic. Global variables should not be used.

  - Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. loops and conditional statements).

7.4 Documentation Requirements

There are a significant number of classes and contained methods you have to implement for this assignment. For each one, *you must provide documentation* in the form of a docstring. The only exception is for overridden methods on subclasses, as python docstrings are inherited.

7.5 Assignment Submission

This assignment follows the same assignment submission policy as assignment 1. Please refer to the assignment 1 task sheet. You must submit your assignment as a single Python file called `a2.py` (use this name – all lower case), and *nothing else*. Your submission will be automatically run to determine the functionality mark. If you submit a file with a *different name*, the tests will *fail* and you will get *zero* for functionality. Do *not* submit the `a2_support.py` file, or any other files. Do *not* submit any sort of archive file (e.g. zip, rar, 7z, etc.).

7.6 Plagiarism

This assignment follows the same plagiarism policy as assignment 1. Please refer to the assignment 1 task sheet.