Module `a2_solution`

Functions

```
def create_encounter(trainer: Trainer, wild_pokemon: Pokemon) -> Battle
```

(7030 Task) Creates a Battle corresponding to an encounter with a wild pokemon.

The enemy in this battle corresponds to a trainer with the empty string for a name, and whose only pokemon is the supplied wild pokemon. Masters students should leave this until they have completed all the non-masters classes.

Parameters

- `trainer`: The adventuring trainer.
- `wild_pokemon`: The pokemon that the player comes into contact with.

Classes

```
class PokemonStats
```

A class modelling the stats of a pokemon. These stats must be non-negative.

Examples

```
>>> stats = PokemonStats((1, 100, 110, 120))
>>> stats.get_hit_chance()
1
>>> stats.get_max_health()
100
>>> stats.get_attack()
110
>>> stats.get_defense()
120
>>> str(stats)
'PokemonStats((1, 100, 110, 120))'
>>> repr(stats)
'PokemonStats((1, 100, 110, 120))'
```

Constructs an instance of PokemonStats.

The format of the incoming stats are: `(hit_chance, health, attack, defense)` with the indices given by constants in the support code.

Parameters

- `stats`: The base list of stats to encapsulate. *These values can be assumed to be non-negative*

Methods

```
def __init__(self, stats: Tuple[float, int, int, int]) -> None
```

Constructs an instance of PokemonStats.

The format of the incoming stats are: `(hit_chance, health, attack, defense)` with the indices given by constants in the support code.

Parameters

- `stats`: The base list of stats to encapsulate. *These values can be assumed to be non-negative*

```
def level_up(self) -> None
```

Grows the PokemonStats instance after the pokemon has levelled up.

On leveling up, the base hit chance should always be = `1`, while the remaining stats grow by `5%` and are rounded down.

Examples

```
>>> stats = PokemonStats((1, 100, 110, 120))
>>> stats.level_up()
>>> stats.get_hit_chance()
1
>>> stats.get_max_health()
105
>>> stats.get_attack()
115
>>> stats.get_defense()
126
```

```
def get_hit_chance(self) -> float
```

Return the pokemon's current chance at making a successful attack.

```
def get_max_health(self) -> int
```

Return the pokemon's max health

```
def get_attack(self) -> int
```

Return the pokemon's attack stat

```
def get_defense(self) -> int
```

Return the pokemon's defense stat

```
def apply_modifier(self, modifier: Tuple[float, int, int, int]) -> PokemonStats
```

Applies a stat modifier and returns the newly constructed, modified pokemon stats.

The resulting pokemon stats are the elementwise sum of the current stats and incoming modification and should be bound by 0.

Parameters

- `modifier`: A list of stat modifications to apply, of the same structure as the initial supplied pokemon stats.

Examples

```
>>> stats = PokemonStats((1, 100, 110, 120))
>>> modified_stats = stats.apply_modifier((-0.5, -20, -10, -5))
>>> stats.get_hit_chance()
1
```

```
>>> modified_stats.get_hit_chance()
0.5
>>> stats.get_max_health()
100
>>> modified_stats.get_max_health()
80
```

`def __str__(self) -> str`

Returns the string representation of this class.

`def __repr__(self) -> str`

Returns the string representation of this class.

`class Pokemon`

A class which represents a Pokemon.

A pokemon's level is determined by its experience points, through the formula: `level = floor(experience ^ (1/3))`.

A pokemon can learn a maximum of 4 moves.

Examples

```
>>> stats = PokemonStats((1, 100, 200, 200))
>>> pokemon = Pokemon("Pikachu", stats, 'electric', [], level=5)
>>> pokemon.get_name()
'Pikachu'
>>> pokemon.get_health()
100
>>> pokemon.get_max_health()
100
>>> pokemon.get_element_type()
'electric'
>>> pokemon.get_level()
5
>>> pokemon.get_experience()
125
>>> pokemon.get_next_level_experience_requirement()
216
>>> pokemon.has_fainted()
False
>>> str(pokemon)
'Pikachu (lv5)'
```

For future examples in this class, you can assume the above example was run first.

Creates a Pokemon instance.

Parameters

- `name`: The name of this pokemon
- `stats`: The pokemon's stats
- `element_type`: The name of the type of this pokemon.
- `moves`: A list of containing the moves that this pokemon will have learned after it is instantiated.
- `level`: The pokemon's level.

Methods

```
def __init__(self, name: str, stats: PokemonStats, element_type: str,
moves: List[ForwardRef('Move')], level: int = 1) -> None
```

Creates a Pokemon instance.

Parameters

- `name`: The name of this pokemon
- `stats`: The pokemon's stats
- `element_type`: The name of the type of this pokemon.
- `moves`: A list of containing the moves that this pokemon will have learned after it is instantiated.
- `level`: The pokemon's level.

```
def get_name(self) -> str
```

Get this pokemon's name.

```
def get_health(self) -> int
```

Get the remaining health of this pokemon.

```
def get_max_health(self) -> int
```

Get the maximum health of this pokemon before stat modifiers are applied.

```
def get_element_type(self) -> str
```

Get the name of the type of this pokemon.

```
def get_remaining_move_uses(self, move: Move) -> int
```

Gets the number of moves left for the supplied move, or 0 if the pokemon doesn't know the move.

```
def get_level(self) -> int
```

Get the level of this pokemon.

```
def get_experience(self) -> int
```

Return the current pokemon's experience.

```
def get_next_level_experience_requirement(self) -> int
```

Return the total experience required for the pokemon to be one level higher.

```
def get_move_info(self) -> List[Tuple[Move, int]]
```

Return a list of the pokemon's known moves and their remaining uses.

This list should be sorted by the name of the moves.

Examples

```
>>> tackle = Attack('tackle', 'normal', 15, 80, 50, .95)
>>> flamethrower = Attack('flamethrower', 'fire', 10, 80, 60, 0.8)
>>> pokemon.get_move_info()
[]
>>> pokemon.learn_move(tackle)
>>> pokemon.learn_move(flamethrower)
>>> pokemon.get_move_info()
```

```
[(Attack('flamethrower', 'fire', 10), 10), (Attack('tackle', 'normal', 15),
15)]
```

`def has_fainted(self) -> bool`

Return true iff the pokemon has fainted.

`def modify_health(self, change: int) -> None`

Modify the pokemons health by the supplied amount.

The resulting health is clamped between 0 and the max health of this pokemon after stat modifiers are applied.

Parameters

- `change`: The health change to be applied to the pokemon.

Examples

```
>>> pokemon.get_max_health()
100
>>> pokemon.get_health()
100
>>> pokemon.modify_health(-20)
>>> pokemon.get_health()
80
>>> pokemon.modify_health(30)
>>> pokemon.get_health()
100
>>> pokemon.modify_health(-9000)
>>> pokemon.get_health()
0
>>> pokemon.has_fainted()
True
```

`def gain_experience(self, experience: int) -> None`

Increase the experience of this pokemon by the supplied amount, and level up if necessary.

Parameters

- `experience`: The amount of experience points to increase.

`def level_up(self) -> None`

Increase the level of this pokemon.

leveling up grows the pokemon's stats, and increase its current health by the amount that the maximum hp increased.

Examples

```
>>> pokemon.get_health()
100
>>> pokemon.get_max_health()
100
>>> pokemon.modify_health(-20)
>>> pokemon.get_health()
80
>>> pokemon.level_up()
>>> pokemon.get_health()
85
>>> pokemon.get_max_health()
105
```

```
def experience_on_death(self) -> int
```

The experience awarded to the victorious pokemon if this pokemon faints.

This is calculated through the formula: `200 * level / 7` and rounded down to the nearest integer, where `level`: the level of the pokemon who fainted.

```
def can_learn_move(self, move: Move) -> bool
```

Returns true iff the pokemon can learn the given move. i.e. they have learned less than the maximum number of moves for a pokemon and they haven't already learned the supplied move.

```
def learn_move(self, move: Move) -> None
```

Learns the given move, assuming the pokemon is able to.

After learning this move, this Pokemon can use this move for `max_uses` times. See `Move`.

Parameters

- `move`: move for pokemon to learn

```
def forget_move(self, move: Move) -> None
```

Forgets the supplied move, if the pokemon knows it.

```
def has_moves_left(self) -> bool
```

Returns true iff the pokemon has any moves they can use

```
def reduce_move_count(self, move: Move) -> None
```

Reduce the move count of the move if the pokemon has learnt it.

```
def add_stat_modifier(self, modifier: Tuple[float, int, int, int], rounds: int)
-> None
```

Adds a stat modifier for a supplied number of rounds.

Parameters

- `modifier`: A stat modifier to be applied to the pokemon.
- `rounds`: The number of rounds that the stat modifier will be in effect for.

```
def get_stats(self) -> PokemonStats
```

Return the pokemon stats after applying all current modifications.

```
def post_round_actions(self) -> None
```

Update the stat modifiers by decrementing the remaining number of rounds they are in effect for.

Hint: students should make sure that the pokemon's health is updated appropriately after status modifiers are removed, i.e. the pokemon's health should never exceed its max health.

```
def rest(self) -> None
```

Returns this pokemon to max health, removes any remaining status modifiers, and resets all move uses to their maximums.

```
def __str__(self) -> str
```

Returns a simple representation of this pokemons name and level.

Examples

```
>>> str(pokemon)
'Pikachu (lv5)'
```

```
def __repr__(self) -> str
```

Returns a string representation of this pokemon

```
class Trainer
```

A class representing a pokemon trainer. A trainer can have 6 Pokemon at maximum.

Notes

- The current pokemon should be kept track of with an instance variable representing the index of the currently selected Pokemon.

Examples

```
>>> DEFAULT_STATS = (1, 100, 200, 200)
>>> def create_pokemon(name):
...     return Pokemon(name, PokemonStats(DEFAULT_STATS), 'normal', moves=[])
...
>>> ash = Trainer('Ash Ketchup')
>>> ash.get_name()
'Ash Ketchup'
>>> ash.get_inventory()
{}
>>> ash.can_switch_pokemon(1)
False
>>> pikachu = create_pokemon("Pikachu")
>>> ash.can_add_pokemon(pikachu)
True
>>> ash.get_all_pokemon()
[]
>>> ash.add_pokemon(pikachu)
>>> ash.get_all_pokemon()
[Pikachu (lv1)]
>>> ash.get_current_pokemon()
Pikachu (lv1)
>>> ash.can_add_pokemon(pikachu)
False
>>> for _ in range(5):
...     ash.add_pokemon(create_pokemon('jynx'))
...
>>> ash.get_all_pokemon()
[Pikachu (lv1), jynx (lv1), jynx (lv1), jynx (lv1), jynx (lv1), jynx (lv1)]
>>> ash.get_current_pokemon()
Pikachu (lv1)
>>> ash.can_add_pokemon(create_pokemon('slugma'))
False
>>> ash
Trainer('Ash Ketchup')
```

```
>>> repr(ash)
"Trainer('Ash Ketchup')"
>>> brock = Trainer("Brock")
>>> brock.get_current_pokemon()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "a2.py", line X, in get_current_pokemon
    raise NoPokemonException()
a2_support.NoPokemonException
```

Create an instance of the Trainer class.

Parameters

- `name`: The name of the trainer.

Methods

```
def __init__(self, name: str) -> None
```

Create an instance of the Trainer class.

Parameters

- `name`: The name of the trainer.

```
def get_name(self) -> str
```

Return the trainer's name.

```
def get_inventory(self) -> Dict[Item, int]
```

Returns the trainer's inventory as a dictionary mapping items to the count of that item remaining in the dictionary.

```
def get_current_pokemon(self) -> Pokemon
```

Gets the current pokemon, or raises a NoPokemonException if the trainer doesn't have a single pokemon.

```
def get_all_pokemon(self) -> List[Pokemon]
```

Returns the trainer's pokemon.

- The order of the pokemon in the list should be the order in which they were added to the roster.
- Modifying the list returned by this method should not affect the state of this instance.

```
def rest_all_pokemon(self) -> None
```

Rests all pokemon in the party

```
def all_pokemon_fainted(self) -> bool
```

Return true iff all the trainer's pokemon have fainted.

```
def can_add_pokemon(self, pokemon: Pokemon) -> bool
```

Returns true iff the supplied pokemon can be added to this trainer's roster.

You shouldn't be able to add the same pokemon instance twice or more than the maximum amount of pokemon for a trainer.

```
def add_pokemon(self, pokemon: Pokemon) -> None
```

Adds a new pokemon into the roster, assuming that doing so would be valid.

If there were no Pokemon in the roster prior to calling this method, set the current pokemon to the one that was added.

```
def can_switch_pokemon(self, index: int) -> bool
```

Determines if the pokemon index would be valid to switch to, and returns true iff the switch would be valid.

You cannot swap to a pokemon which is currently out on battle, or which has fainted.

Parameters

- `index`: The index of the next pokemon in the roster.

```
def switch_pokemon(self, index: int) -> None
```

Switches pokemon to the one at the supplied index, assuming that the switch is valid.

Parameters

- `index`: The index of the pokemon to switch to.

```
def add_item(self, item: Item, uses: int) -> None
```

Adds an item to the trainer's inventory and increments its uses by the supplied amount.

Parameters

- `item`: The item to add.
- `uses`: The quantity of the item to be added to the inventory.

Examples

```
>>> food = Food("Burger King Foot Lettuce", 20)
>>> ash.get_inventory()
{}
>>> ash.add_item(food, 1)
>>> ash.get_inventory()
{Food('Burger King Foot Lettuce'): 1}
>>> ash.use_item(food)
>>> ash.get_inventory()
{}
>>> ash.add_item(food, 3)
>>> ash.add_item(food, 4)
>>> ash.get_inventory()
{Food('Burger King Foot Lettuce'): 7}
```

```
def has_item(self, item: Item) -> bool
```

Returns true if the item is in the trainer's inventory and has uses.

```
def use_item(self, item: Item) -> None
```

If the item is present in the trainer's inventory, decrement its count.

Removes the item from the inventory entirely if its count hits 0.

```
def __str__(self) -> str
```

Returns a string representation of a Trainer

```
def __repr__(self) -> str
```

Returns a string representation of a Trainer

```
class Battle
```

A class which represents a pokemon battle. A battle can be between trainers or between a trainer and a wild pokemon. In this assignment, non-trainer battles are represented by a battle between 2 trainers, namely a regular trainer and a 'dummy' trainer whose only pokemon is the wild pokemon.

The main state-components of the battle are the action queue, and the turn:

1. Pokemon battles aren't strictly turn-based, because the priority of each Action must be evaluated before they are performed. To make this happen, each round, each trainer adds their desired action to an `action queue`, and then the actions are performed in order of priority. In our implementation, the trainers cannot add an action to the queue unless it is valid for them to do so, based on the `turn`, and if the action would be valid.
2. The `turn` is the battle's way of determining who should be allowed to add actions to the action queue. Each round, the `turn` starts as None. The first time an action is performed by a trainer that round, the turn is set to the opposite trainer, becoming a boolean value which is True if the opposite trainer is the player, and False if they are the enemy. When the `turn` is a boolean, it means that only the trainer who it points to can add actions to the queue/enact them. When both trainers have enacted a valid action, the round is considered over, and the `turn` should be set to `None`.

Examples

```
>>> DEFAULT_STATS = (1, 100, 200, 200)
>>> def create_pokemon(name):
...      return Pokemon(name, PokemonStats(DEFAULT_STATS), 'normal', moves=[])
...
>>> ash = Trainer("Ash")
>>> brock = Trainer("Brock")
>>> ash.add_pokemon(create_pokemon("pikachu"))
>>> brock.add_pokemon(create_pokemon("geodude"))
>>> battle = Battle(ash, brock, True)
>>> print(battle.get_turn())
None
>>> battle.is_action_queue_empty()
True
>>> battle.is_over()
False
>>> battle.queue_action(Flee(), True)
>>> battle.trainer_has_action_queued(True)
True
>>> battle.get_trainer(True)
Trainer('Ash')
>>> battle.get_trainer(False)
Trainer('Brock')
>>> battle.is_ready()
False
>>> battle.queue_action(Flee(), False)
>>> battle.is_action_queue_full()
True
>>> battle.is_ready()
```

```
True
>>> summary = battle.enact_turn()
>>> summary.get_messages()
['Unable to escape a trainer battle.']
>>> battle.get_turn()
False
>>> battle.is_ready()
True
>>> battle.is_action_queue_full()
False
>>> other_summary = battle.enact_turn()
>>> other_summary.get_messages()
['Unable to escape a trainer battle.']
>>> print(battle.get_turn())
None
```

Creates an instance of a trainer battle.

Parameters

- `player`: The trainer corresponding to the player character.
- `enemy`: The enemy trainer.
- `is_trainer_battle`: True iff the battle takes place between trainers.

Methods

```
def __init__(self, player: Trainer, enemy: Trainer, is_trainer_battle: bool)
-> None
```

Creates an instance of a trainer battle.

Parameters

- `player`: The trainer corresponding to the player character.
- `enemy`: The enemy trainer.
- `is_trainer_battle`: True iff the battle takes place between trainers.

```
def get_turn(self) -> Optional[bool]
```

Get whose turn it currently is

```
def get_trainer(self, is_player: bool) -> Trainer
```

Gets the trainer corresponding to the supplied parameter.

Parameters

- `is_player`: True iff the trainer we want is the player.

```
def attempt_end_early(self) -> None
```

Ends the battle early if it's not a trainer battle

```
def is_trainer_battle(self) -> bool
```

Returns true iff the battle is between trainers

```
def is_action_queue_full(self) -> bool
```

Returns true if both trainers have an action queued.

```
def is_action_queue_empty(self) -> bool
```

Returns true if neither trainer have an action queued.

```
def trainer_has_action_queued(self, is_player: bool) -> bool
```

Returns true iff the supplied trainer has an action queued

Parameters

- `is_player`: True iff the trainer we want to check for is the player.

```
def is_ready(self) -> bool
```

Returns true iff the next action is ready to be performed.

The battle is deemed ready if neither trainer has performed an action this round and the action queue is full, or if one trainer has performed an action, and the other trainer is in the queue.

```
def queue_action(self, action: Action, is_player: bool) -> None
```

Attempts to queue the supplied action if it's valid given the battle state, and came from the right trainer.

An action is unable to be added to the queue if: The trainer is already in the queue; The queue is ready; The action is invalid given the game state;

Parameters

- `action`: The action we are attempting to queue
- `is_player`: True iff we're saying the action is going to be performed by the player.

```
def enact_turn(self) -> Optional[ActionSummary]
```

Attempts to perform the next action in the queue, and returns a summary of its effects if it was valid.

Notes

- If the next action in the queue is invalid, it should still be removed from the queue.
- If this was the last turn to be performed that round, perform the post round actions.

```
def is_over(self) -> bool
```

Returns true iff the battle is over.

A battle is over if all of the pokemon have fainted for either trainer, or if it ended early.

```
class ActionSummary
```

A class containing messages about actions and their effects.

These messages are handled by the view to display information about the flow of the game.

Constructs a new ActionSummary with an optional message.

Parameters

- `message`: An optional message to be included.

Methods

```
def __init__(self, message: Optional[str] = None) -> None
```

Constructs a new ActionSummary with an optional message.

Parameters

- `message`: An optional message to be included.

```
def get_messages(self) -> List[str]
```

Returns a list of the messages contained within this summary.

Examples

```
>>> msg = ActionSummary()
>>> msg.get_messages()
[]
>>> msg = ActionSummary("oh-wo")
>>> msg.get_messages()
['oh-wo']
```

```
def add_message(self, message: str) -> None
```

Adds the supplied message to the ActionSummary instance.

Parameters

- `message`: The message to add.

Examples

```
>>> msg = ActionSummary()
>>> msg.add_message("Action did a thing")
>>> msg.get_messages()
['Action did a thing']
```

```
def combine(self, summary: ActionSummary) -> None
```

Combines two ActionSummaries.

The messages contained in the supplied summary should be added after those currently contained.

Parameters

- `summary`: A summary containing the messages to add.

Examples

```
>>> msg = ActionSummary('first')
>>> msg2 = ActionSummary('second')
>>> msg.combine(msg2)
>>> msg.get_messages()
['first', 'second']
>>> msg2.get_messages()
[‘second’]
```

```
class Action
```

An abstract class detailing anything which takes up a turn in battle.

Applying an action can be thought of as moving the game from one state to the next.

Subclasses

- Flee
- Item
- Move
- SwitchPokemon

Methods

```
def get_priority(self) -> int
```

Returns the priority of this action, which is used to determine which action is performed first each round in the battle.

Lower values of priority are 'quicker' than higher values, e.g. an Action with priority 0 happens before one with priority 1.

You might want to take a look at the support code for a hint here.

```
def is_valid(self, battle: Battle, is_player: bool) -> bool
```

Determines if the action would be valid for the given trainer and battle state. Returns true iff it would be valid.

By default, no action is valid if the game is over, or if it's not the trainer's turn.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

```
def apply(self, battle: Battle, is_player: bool) -> ActionSummary
```

Applies the action to the game state and returns a summary of the effects of doing so.

On the base Action class, this method should raise a NotImplementedError.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

Examples

```
>>> action = Action()
>>> action.apply(battle, True)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "a2.py", line X, in apply
    raise NotImplementedError()
NotImplementedError
```

```
def __str__(self) -> str
```

Return a string representation of this class.

```
def __repr__(self) -> str
```

Return a string representation of this class

Examples

```
>>> action = Action()
>>> str(action)
'Action()'
>>> repr(action)
'Action()'
```

```
class Flee
```

An action where the trainer attempts to run away from the battle.

Notes

- While it may still be valid, it has no effect in trainer battles.
- If successful, this should end the battle early.

Examples

```
>>> flee = Flee()
>>> str(flee)
'Flee()'
>>> repr(flee)
'Flee()'
```

Ancestors

- Action

Methods

```
def is_valid(self, battle: Battle, is_player: bool) -> bool
```

Determines if an attempt to flee would be valid for a given battle state. Returns true iff it would be valid.

Fleeing is considered a valid action if the base action validity checks pass, and the trainer's current pokemon has not fainted. This does not mean, however, that a trainer can flee trainer battles. In that case, fleeing is considered wasting a turn.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

```
def apply(self, battle: Battle, is_player: bool) -> ActionSummary
```

The trainer attempts to flee the battle.

The resulting message depends on whether or not the action was successful. See the support code for a hint.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

Inherited members

- `Action`:
  - `__repr__`
  - `__str__`
  - `get_priority`

`class SwitchPokemon`

An action representing the trainer's intention to switch pokemon.

Examples

```
>>> switch = SwitchPokemon(2)
>>> str(switch)
'SwitchPokemon(2)'
>>> repr(switch)
'SwitchPokemon(2)'
```

Creates an instance of the SwitchPokemon class.

Parameters

- `next_pokemon_index`: The index of the pokemon the trainer wants to switch to.

Ancestors

- Action

Methods

`def __init__(self, next_pokemon_index: int) -> None`

Creates an instance of the SwitchPokemon class.

Parameters

- `next_pokemon_index`: The index of the pokemon the trainer wants to switch to.

`def is_valid(self, battle: Battle, is_player: bool) -> bool`

Determines if switching pokemon would be valid for a given trainer and battle state. Returns true iff it would be valid.

After checking the validity requirements specified on the base Action class, switching delegates validity checking to the `Trainer` class.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

`def apply(self, battle: Battle, is_player: bool) -> ActionSummary`

The trainer switches pokemon, assuming that the switch would be valid.

If the trainer using this action is the player, and their pokemon has not yet fainted, a message should be added to the action summary, in the form: `'{pokemon_name}, return!'`.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

Inherited members

- `Action`:
  - `__repr__`
  - `__str__`
  - `get_priority`

`class Item`

An abstract class representing an Item, which a trainer may attempt to use to influence the battle.

Creates an Item.

Parameters

- `name`: The name of this item

Ancestors

- Action

Subclasses

- Food
- Pokeball

Methods

```
def __init__(self, name: str) -> None
```

Creates an Item.

Parameters

- `name`: The name of this item

```
def get_name(self) -> str
```

Return the name of this item

```
def is_valid(self, battle: Battle, is_player: bool) -> bool
```

Determines if using the item would be a valid action for the given trainer and battle state. Returns true iff it would be valid.

In addition to the validity requirements specified on the base Action class, `Item` and its subclasses are considered valid if: 1. The trainer's current pokemon has not fainted. 2. The item exists in the inventory of the trainer attempting to use it.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

```
def decrement_item_count(self, trainer: Trainer) -> None
```

Decrease the count of this item by one in the trainer's inventory

Parameters

- `trainer`: The trainer attempting to use this item.

Inherited members

- `Action`:
  - `__repr__`
  - `__str__`
  - `apply`
  - `get_priority`

```
class Pokeball
```

An item which a trainer can use to attempt to catch wild pokemon.

Examples

```
>>> pokeball = Pokeball("Master Ball", 1)
>>> pokeball
Pokeball('Master Ball')
>>> str(pokeball)
"Pokeball('Master Ball')"
>>> repr(pokeball)
"Pokeball('Master Ball')"
```

Creates a pokeball instance, used to catch pokemon in wild battles

Parameters

- `name`: The name of this pokeball
- `catch_chance`: The chance this pokeball has of catching a pokemon.

Ancestors

- Item
- Action

Methods

```
def __init__(self, name, catch_chance) -> None
```

Creates a pokeball instance, used to catch pokemon in wild battles

Parameters

- `name`: The name of this pokeball
- `catch_chance`: The chance this pokeball has of catching a pokemon.

```
def apply(self, battle: Battle, is_player: bool) -> ActionSummary
```

Attempt to catch the enemy pokemon and returns an ActionSummary containing information about the catch attempt.

The returned summary will contain a different message based on the results of attempting to use the pokeball. See the support code for some hints as to what these messages might be.

Notes

- No matter the result of the catch attempt, a pokeball will be used.
- Catching pokemon is impossible in trainer battles.
- The `did_succeed` method from the support code must be used to determine if a catch attempt was successful.
- The wild pokemon will be added to the trainers roster if there is room
- In a wild battle, catching the enemy pokemon will end the battle.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

Inherited members

- `Item`:
  - `__repr__`
  - `__str__`
  - `decrement_item_count`
  - `get_name`
  - `get_priority`
  - `is_valid`

```
class Food
```

An Item which restores HP to the pokemon whose trainer uses it.

Examples

```
>>> soup = Food("Good Soup", 69)
>>> soup
Food('Good Soup')
>>> str(soup)
"Food('Good Soup')"
>>> repr(soup)
"Food('Good Soup')"
```

Create a Food instance.

Parameters

- `name`: The name of this food.
- `health_restored`: The number of health points restored when a pokemon eats this piece of food.

Ancestors

- Item
- Action

Methods

`def __init__(self, name: str, health_restored: int) -> None`

Create a Food instance.

Parameters

- `name`: The name of this food.
- `health_restored`: The number of health points restored when a pokemon eats this piece of food.

`def apply(self, battle: Battle, is_player: bool) -> ActionSummary`

The trainer's current pokemon eats the food.

Their current pokemon's health should consequently increase by the amount of health restored by this food.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this item.

Inherited members

- `Item`:
  - `__repr__`
  - `__str__`
  - `decrement_item_count`
  - `get_name`
  - `get_priority`
  - `is_valid`

`class Move`

An abstract class representing all learnable pokemon moves.

Creates an instance of the Move class.

Like pokemon, moves have a type which determines their effectiveness. They also have a speed which determines the move's priority.

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves priorities.

Ancestors

- Action

Subclasses

- Attack
- StatusModifier

Methods

```
def __init__(self, name: str, element_type: str, max_uses: int, speed: int) -> None
```

Creates an instance of the Move class.

Like pokemon, moves have a type which determines their effectiveness. They also have a speed which determines the move's priority.

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves priorities.

```
def get_name(self) -> str
```

Return the name of this move

```
def get_element_type(self) -> str
```

Return the type of this move

```
def get_max_uses(self) -> int
```

Return the maximum times this move can be used

```
def get_priority(self) -> int
```

Return the priority of this move.

Moves have a speed-based priority. By default they are slower than other actions, with their total priority being calculated by adding the default speed-based action priority to the individual move's speed.

```
def is_valid(self, battle: Battle, is_player: bool) -> bool
```

Determines if the move would be valid for the given trainer and battle state. Returns true iff it would be valid.

In addition to the validity requirements specified on the base Action class, a `Move` is considered valid if: 1. The trainer's current pokemon has not fainted. 2. The trainer's current pokemon has learnt this move. 3. The trainer's current pokemon has uses remaining for this move.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

```
def apply(self, battle: Battle, is_player: bool) -> ActionSummary
```

Applies the Move to the game state.

Generally, the move should be performed and its effects should be applied to the player and/or the enemy if needed. In addition, the appropriate pokemon's remaining moves should be updated.

Notes

- In the resulting ActionSummary, messages for ally effects should preceed enemy effects.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

```
def apply_ally_effects(self, trainer: Trainer) -> ActionSummary
```

Apply this move's effects to the ally trainer; if appropriate, and return the resulting ActionSummary.

Parameters

- `trainer`: The trainer whose pokemon is using the move.

```
def apply_enemy_effects(self, trainer: Trainer, enemy: Trainer) -> ActionSummary
```

Apply this move's effects to the enemy; if appropriate, and return the resulting ActionSummary.

Parameters

- `trainer`: The trainer whose pokemon is using the move.
- `enemy`: The trainer whose pokemon is the target of the move.

Inherited members

- `Action`:
  - `__repr__`
  - `__str__`

```
class Attack
```

A class representing damaging pokemon moves, that may be used against an enemy pokemon.

Notes

- In addition to regular move requirements, attacking moves have a base damage and hit chance.
- Base damage is the damage this move would do before the pokemon's attack, defense or type effectiveness is accounted for.
- Hit chance is a measure of how likely the move is to hit an enemy pokemon, before the pokemon's hit chance stat is taken into account.
- The `did_succeed` method from the support code must be used to determine if the attack hit.
- If an attack 'misses' it does no damage to the enemy pokemon.

Examples

```
>>> tackle = Attack('tackle', 'normal', 15, 100, 40, .95)
>>> tackle.get_name()
'tackle'
>>> tackle.get_element_type()
```

```
'normal'
>>> tackle.get_max_uses()
15
>>> tackle.get_priority()
101
>>> str(tackle)
"Attack('tackle', 'normal', 15)"
>>> repr(tackle)
"Attack('tackle', 'normal', 15)"
```

Creates an instance of an attacking move.

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `base_damage`: The base damage of this move.
- `hit_chance`: The base hit chance of this move.

Ancestors

- Move
- Action

Methods

```
def __init__(self, name: str, element_type: str, max_uses: int, speed: int,
base_damage: int, hit_chance: float) -> None
```

Creates an instance of an attacking move.

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `base_damage`: The base damage of this move.
- `hit_chance`: The base hit chance of this move.

```
def did_hit(self, pokemon: Pokemon) -> bool
```

Determine if the move hit, based on the product of the pokemon's current hit chance, and the move's hit chance. Returns True iff it hits.

Parameters

- `pokemon`: The attacking pokemon

```
def calculate_damage(self, pokemon: Pokemon, enemy_pokemon: Pokemon) -> int
```

Calculates what would be the total damage of using this move, assuming it hits, based on the stats of the attacking and defending pokemon.

The damage formula is given by: `d * e * a / (D + 1)`, rounded down to the nearest integer, where: `d` is the move's base damage; `e` is the move's type effectiveness (see support code); `a` is the attacking pokemon's attack stat; `D` is the defending pokemon's defense stat.

Parameters

- `pokemon`: The attacking trainer's pokemon
- `enemy_pokemon`: The defending trainer's pokemon

Inherited members

- `Move`:
  - `__repr__`
  - `__str__`
  - `apply`
  - `apply_ally_effects`
  - `apply_enemy_effects`
  - `get_element_type`
  - `get_max_uses`
  - `get_name`
  - `get_priority`
  - `is_valid`

```
class StatusModifier
```

An abstract class to group commonalities between buffs and debuffs.

Creates an instance of this class

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `modification`: A list of the same structure as the `PokemonStats`, to be applied for the duration of the supplied number of rounds.
- `rounds`: The number of rounds for the modification to be in effect.

Ancestors

- Move
- Action

Subclasses

- Buff
- Debuff

Methods

```
 def __init__(self, name: str, element_type: str, max_uses: int, speed: int,
modification: Tuple[float, int, int, int], rounds: int) -> None
```

Creates an instance of this class

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `modification`: A list of the same structure as the `PokemonStats`, to be applied for the duration of the supplied number of rounds.
- `rounds`: The number of rounds for the modification to be in effect.

Inherited members

- `Move`:
    - `__repr__`
    - `__str__`
    - `apply`
    - `apply_ally_effects`
    - `apply_enemy_effects`
    - `get_element_type`
    - `get_max_uses`
    - `get_name`
    - `get_priority`
    - `is_valid`

`class Buff`

Moves which buff the trainer's selected pokemon.

A buff is a stat modifier that is applied to the pokemon using the move.

Examples

```
>>> modifier = (0.2, 100, 100, 100)
>>> meditate = Buff('meditate', 'psychic', 5, 80, modifier, 5)
>>> meditate.get_name()
'meditate'
>>> meditate.get_element_type()
'psychic'
>>> meditate.get_max_uses()
5
>>> meditate.get_priority()
81
>>> str(meditate)
"Buff('meditate', 'psychic', 5)"
>>> repr(meditate)
"Buff('meditate', 'psychic', 5)"
```

Creates an instance of this class

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `modification`: A list of the same structure as the `PokemonStats`, to be applied for the duration of the supplied number of rounds.
- `rounds`: The number of rounds for the modification to be in effect.

Ancestors

- StatusModifier
- Move
- Action

Inherited members

- `StatusModifier`:
  - `__init__`
  - `__repr__`
  - `__str__`
  - `apply`
  - `apply_ally_effects`
  - `apply_enemy_effects`
  - `get_element_type`
  - `get_max_uses`
  - `get_name`
  - `get_priority`
  - `is_valid`

`class Debuff`

Moves which debuff the enemy trainer's selected pokemon.

A debuff is a stat modifier that is applied to the enemy pokemon which is the target of this move.

Examples

```
>>> modifier = (0, -50, 0, -50)
>>> toxic = Debuff('toxic', 'poison', 10, 70, modifier, 4)
>>> toxic.get_name()
'toxic'
>>> toxic.get_element_type()
'poison'
>>> toxic.get_max_uses()
10
>>> toxic.get_priority()
71
>>> str(toxic)
"Debuff('toxic', 'poison', 10)"
>>> repr(toxic)
"Debuff('toxic', 'poison', 10)"
```

Creates an instance of this class

Parameters

- `name`: The name of this move
- `element_type`: The name of the type of this move
- `max_uses`: The number of time this move can be used before resting
- `speed`: The speed of this move, with lower values corresponding to faster moves.
- `modification`: A list of the same structure as the `PokemonStats`, to be applied for the duration of the supplied number of rounds.
- `rounds`: The number of rounds for the modification to be in effect.

Ancestors

- StatusModifier
- Move
- Action

Inherited members

- `StatusModifier`:
  - `__init__`
  - `__repr__`
  - `__str__`
  - `apply`
  - `apply_ally_effects`
  - `apply_enemy_effects`
  - `get_element_type`
  - `get_max_uses`
  - `get_name`
  - `get_priority`
  - `is_valid`

`class Strategy`

An abstract class providing behaviour to determine a next action given a battle state.

Subclasses

- ScaredyCat
- TeamRocket

Methods

```
def get_next_action(self, battle: Battle, is_player: bool) -> Action
```

Determines and returns the next action for this strategy, given the battle state and trainer.

This method should be overriden on subclasses.

Parameters

- `battle`: The ongoing pokemon battle
- `is_player`: True iff the player is using this action.

`class ScaredyCat`

(7030 Task) A strategy where the trainer always attempts to flee.

Switches to the next available pokemon if the current one faints, and then keeps attempting to flee.

Ancestors

- Strategy

Inherited members

- `Strategy`:
  - `get_next_action`

`class TeamRocket`

(7030 Task) A tough strategy used by Pokemon Trainers that are members of Team Rocket.

Behaviour

1. Switch to the next available pokemon if the current one faints.
2. Attempt to flee any wild battle.
3. If the enemy trainer's current pokemon's name is 'pikachu', throw pokeballs at it, if some exist in the inventory.
4. Otherwise, use the first available move with an elemental type effectiveness greater than 1x against the defending pokemon's type.
5. Otherwise, use the first available move with uses.
6. Attempt to flee if the current pokemon has no moves with uses.

Ancestors

- Strategy

Inherited members

- `Strategy`:
    - `get_next_action`

Index

    - can_learn_move
    - learn_move
    - forget_move
    - has_moves_left
    - reduce_move_count
    - add_stat_modifier
    - get_stats
    - post_round_actions
    - rest
    - __str__
    - __repr__

  - **Trainer**

    - __init__
    - get_name
    - get_inventory
    - get_current_pokemon
    - get_all_pokemon
    - rest_all_pokemon
    - all_pokemon_fainted
    - can_add_pokemon
    - add_pokemon
    - can_switch_pokemon
    - switch_pokemon
    - add_item
    - has_item
    - use_item
    - __str__
    - __repr__

  - **Battle**

    - __init__
    - get_turn
    - get_trainer
    - attempt_end_early
    - is_trainer_battle
    - is_action_queue_full
    - is_action_queue_empty
    - trainer_has_action_queued
    - is_ready
    - queue_action
    - enact_turn
    - is_over

  - **ActionSummary**

    - __init__
    - get_messages
    - add_message
    - combine

  - **Action**

    - get_priority
    - is_valid
    - apply
    - __str__

        - `__repr__`

  - **Flee**

        - is_valid
        - apply

  - **SwitchPokemon**

        - `__init__`
        - is_valid
        - apply

  - **Item**

        - `__init__`
        - get_name
        - is_valid
        - decrement_item_count

  - **Pokeball**

        - `__init__`
        - apply

  - **Food**

        - `__init__`
        - apply

  - **Move**

        - `__init__`
        - get_name
        - get_element_type
        - get_max_uses
        - get_priority
        - is_valid
        - apply
        - apply_ally_effects
        - apply_enemy_effects

  - **Attack**

        - `__init__`
        - did_hit
        - calculate_damage

  - **StatusModifier**

        - `__init__`

  - **Buff**

  - **Debuff**

  - **Strategy**

Generated by pdoc 0.10.0.