ex1_event_logger.py

`@dataclass` and Event

- We use `dataclass` to automatically generate `__init__` and other dunder methods.
- The type annotations reflect exactly the attributes requested: `id_num`, `description`, `next_command`, `next`, and `prev`.

Initialization in `EventList.__init__`

- `first` and `last` start as `None` to indicate an empty list.

`is_empty`

- Returns `True` if `first` is `None`, meaning there are no events.

`add_event`

- If the list is empty, set both `first` and `last` to the new `event`.
- Otherwise:
    - Set the old `last` event's `next_command` to the provided `command`.
    - Update the old `last` event's `next` to point to the new `event`.
    - Update the new `event` to have the `prev` pointer set to the old last event, and set its `next` to `None`.
    - Assign `self.last` to the new event.

`remove_last_event`

- If the list is empty, do nothing.
- If there is exactly one event (`self.first is self.last`), remove it by setting both `first` and `last` to `None`.
- Otherwise, update the list's `last` pointer to the second-last event. Set that event's `next` and `next_command` to `None` (since it is now the last event).

`get_id_log`

- Traverses from `first` to `last`, collecting `id_num` values into a list that is then returned.

ex1_simulation.py

`SimpleAdventureGame.get_location`

- If no `loc_id` is provided, it uses the game's current location ID.
- Otherwise, returns the `Location` object for the requested ID.

`AdventureGameSimulation.__init__`

- Creates an empty `EventList` and initializes `SimpleAdventureGame`.
- Retrieves the initial `Location` using `self._game.get_location()`.
- Creates the first `Event` with `next_command=None`, reflecting that no prior command led us to the initial location.
- Adds this initial event to the event list, then calls `generate_events` to process all subsequent commands.

`AdventureGameSimulation.generate_events`

- Iterates over each command in `commands`.
- Looks up the next location ID via `current_location.available_commands[cmd]`.
- Updates the `_game.current_location_id` so the game's state tracks the new location.
- Retrieves the corresponding `Location`, creates a new `Event` object, and adds it to `self._events`, specifying the command used to get there.
- Updates `current_location` to continue the loop.

adventure.py

1. Game Initialization

- **Step 1:** Parse the JSON game data file.
    - Ensure the file contains `locations` and `items`.
    - Use the `_load_game_data` method to load this data into Python objects (`Location` and `Item`).
- **Hint:** Validate the structure of the JSON file to avoid runtime errors.
- **Step 2:** Initialize the game state.
    - Set the starting `current_location_id`.
    - Keep track of whether the game is `ongoing` and initialize an empty player `inventory`.

---

2. Game Loop

- **Step 3:** Display the current location.
    - Use the `get_location` method to fetch the current location object.
    - Show a detailed description (`long_description`) if the location hasn't been visited before. Otherwise, show a `brief_description`.
- **Hint:** Use the `visited` attribute to track whether the location has been visited and update it accordingly.
- **Step 4:** Show available actions.
    - List global commands (`look`, `inventory`, etc.).
    - List location-specific commands (`go north`, `go east`, etc.).

---

3. Handling Player Input

- **Step 5:** Validate the player's input.
    - Allow global commands, location-specific commands, and special commands (`pick up <item>` or `drop <item>`).
    - Prompt the player again if the input is invalid.
- **Hint:** Use the `startswith` method to handle `pick up` and `drop` commands dynamically.

---

4. Inventory Management

- **Step 6:** Implement `pick_up_item`.
    - Locate the item in the current location using `location.items`.
    - Remove the item from the location and add it to the player's `inventory`.
- **Hint:** Use `next()` to retrieve the `Item` object from `_items` efficiently. Handle cases where the item isn't found in the location.
- **Step 7:** Implement `drop_item`.
    - Find the item in the player's inventory.
    - Remove it from the inventory and add it to the current location.
- **Hint:** Ensure the item is appended to the correct location's `items` list.

---

5. Moving the Player

- **Step 8:** Use `move_player`.
    - Update `current_location_id` based on the destination ID.
    - Increment `num_moves` to enforce a move limit (if applicable).
- **Hint:** Check whether `num_moves` exceeds `max_moves`. If so, end the game with a message.

---

6. Scoring System（one possible way）

- **Step 9:** Implement `calculate_score`.
    - Iterate through all items in `_items`.
    - Check whether each item is in its `target_position`.
    - Add the `target_points` for correctly placed items.
- **Hint:** Use `location.items` to verify the item's presence in a specific location.

---

7. Logging Events

- **Step 10:** Log events with `EventList`.
    - Create an `Event` object each time the player visits a location or performs an action.
    - Use the `add_event` method to append the event to the log.
- **Hint:** Store the `choice` (command) leading to the event for future reference.

---

8. Undoing Actions

- **Step 11:** Implement `undo_last_action`.
    - Use the `EventList` to retrieve the last event.
    - Revert the player's location to the previous one.
    - Remove the last event from the log.
- **Hint:** Extend the functionality to undo inventory changes or score updates if needed.
