Adventure Documentation:

Step 0:

This step was quite straightforward as the instructions for the class Room() and its methods were quite clear, especially as it was already specified what and how the connections to different rooms should be represented in the form of dictionaries.

Step 1:

The syntax for reading a file including the use of the .strip() method were already explored in previous assignments. While the instructions suggested a for-loop format to iterate over the lines, the length of each data file is different so I would have to iterate over each line first to determine how many lines there were before iterating over each line again to read the file. It seemed silly to me to iterate over the file twice when you could just read the file during the first iteration, which is why I just used a while-loop with a break at the newline.

```
with open(filename) as f:

    while True:
        line = f.readline().strip()
        if len(line) == 0:
            break
```

Because readline() “remembers” its location in a file, each section of the data file could be read by reusing the same while-loop as above. Otherwise, this step mostly involved using the various Room() methods and following the instructions.

Step 2:

Since we are already supplied with .current_room in adventure.py, I understood that the goal of the move method was to assign a new room to .current_room. Since the get_connection method returns a room given a direction, I used it to assign a new room to .current_room.

```
if self.current_room.has_connection(direction):
    self.current_room = self.current_room.get_connection(direction)
    return True
return False
```

Step 3:

The visited methods were very straightforward. The instructions mentioned adding the set_visited method to the move method and the is_visited method to the get_description method but since we are already checking if a room has been visited in get_description, why not just set it to visited once we've established that a certain room hasn't been visited yet?

```
def get_description(self):
    if self.current_room.is_visited():
        return self.current_room.name
    self.current_room.set_visited()
    return self.current_room.desc
```

Step 4:

This step was quite simple to implement is it just involved printing things in main if certain commands were entered.

```
if command == "HELP":
    print(adventure.help())
    continue

# Long description
if command == "LOOK":
    print(adventure.get_long_description())
```

Step 5:

History is basically just a reskinning of the stack class designed for palindrome. Instead of "items" we use "rooms.

```
class History():

    def __init__(self) -> None:
        self.rooms = []

    def push(self, room: Any) -> None:
        self.rooms.append(room)

    def pop(self, adventure: type) -> Any:
        adventure.current_room = self.rooms.pop()
```

Before each move, the previous current room is pushed to the history stack and

```
history.push(self.current_room)
self.current_room = self.current_room.get_connection(direction)
```

Using the “BACK” command pops out the previous room and ‘continues’ the main loop.

```
if command == "BACK":
    try:
        history.pop(adventure)
        print(adventure.get_description())
    except IndexError:
        print("Can't go back.")
    continue
```

Step 6:

Everything worked fine here.

Step 7:

I implemented is_forced by returning whether FORCED was in the connections of the current_room.

```
def is_forced(self) -> bool:
    return "FORCED" in self.current_room.connections
```

I then added a check to the at the start of the main loop before any commands are taken and fed “FORCED” to the move method if is_forced is True.

I then modified the move method to skip pushing to history if direction is “FORCED” in order to have “BACK” function correctly.

```
if direction == "FORCED":
    self.current_room = self.current_room.get_connection(direction)
    return True
history.push(self.current_room)
self.current_room = self.current_room.get_connection(direction)
return True
```

Step 8:

Because most of logic and manipulation behind the Item class is handled in other classes, my representation of an item is just a name and a description.

```
class Item():

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

        self.name = name
        self.desc = desc
```

I added an item list to the Room class and an add_item method that appends an item to the list. I used the same loop I used in step 1 to load the items to a particular room.

```
def add_item(self, item: type) -> None:

    self.items.append(item)
```

```
line_list = line.split("\t")
for i in range(1, len(line_list), 2):
    item = Item(line_list[0], line_list[1])
    self.rooms[int(line_list[2])].add_item(item)
```

My inventory is just a list in the Adventure class. The take and drop methods that I implemented work in basically the same way. They search through either the current_room.items list or inventory for the item supplied as an argument and first appends the item to the other list before popping the item out.

```
def take(self, item: str) -> str:
    for i in range(len(self.current_room.items)):
        if self.current_room.items[i].name == item:
            self.inventory.append(self.current_room.items[i])
            self.current_room.items.pop(i)
            return (item + " taken.")
    return ("No such item.")

def drop(self, item: str) -> str:
    for i in range(len(self.inventory)):
        if self.inventory[i].name == item:
            self.current_room.items.append(self.inventory[i])
            self.inventory.pop(i)
            return (item + " dropped.")
    return ("No such item.")
```

Step 9:

To handle conditional movements, I first had to figure out a way to parse the datafile to separate conditional rooms. I managed to do this by using try and except. I realized that if I tried to typecast a conditional room (such as 12/LAMP) to an int, it would raise a ValueError. So, by trying to typecast each room to an int as it is being read from the datafile, I could separate conditional rooms to a separate dictionary in the Room class. I split the conditional room further into the room value and item using the split method. Because the conditional rooms had 3 parameters, I had to create a new method called add_cond_connection to attach the connection. I initially accomplished this by using the following form:

```
self.cond_connections[item]= {direction: room}
```

Room 13 gave me a lot of trouble here because it had multiple conditional connections with the same item so the key: value pair would be overwritten each time a new connection was loaded. I solved this by making the value of each item key a list of direction: room pairs and appending to the list instead.

```
try:
    self.cond_connections[item].append({direction: room})
except KeyError:
    self.cond_connections[item] = []
    self.cond_connections[item].append({direction: room})
```

The last thing that I did was to modify the move method in Adventure to go through the inventory each time to check if any of the items show up in the conditional connections in the current room and attempt to use the direction as a key. If a key error is raised, that means that the direction supplied not a conditional connection and regular movement as in step 2 is used.

```
for i in range(len(self.inventory)):
    if self.inventory[i].name in self.current_room.cond_connections:
        for movement in self.current_room.cond_connections[self.inventory[i].name]:
            try:
                previous_room = self.current_room
                self.current_room = self.rooms[int(movement[direction])]
                history.push(previous_room)
                return True
            except KeyError:
                break
```

Step 10:

Synonym was not very difficult to implement. By using a dictionary to hold the synonyms as the keys and full commands as keys.

```
if command in adventure.synonyms:
    command = adventure.synonyms[command]
```

Improvements:

While I have started implementing type hints to my code, they are not complete, so I finished type hinting the rest of my code.

Since the main function of adventure.py was not wrapped in a main function, I made sure to do that so that the Adventure class can be used elsewhere if desired. I did this by separating the code in the main section into an init_game function and a play_game function containing the initialization code and the game logic respectively.

```
def init_game() -> 'Adventure':
    """
    Initialize the game by loading the requested game or default game.

    post: Return a game instance: an Adventure object.
    """
def play_game(adventure: 'Adventure') -> None:
    """
    Game logic for playing the provided adventure.

    post: Display the welcome message: a str.
          Display description of initial room: a str.
          Process user commands, update game state, and display relevant
          message.
    """
```

Reading through my code, I noticed that some of the variable names are a little cryptic. For example:

```
line: str = f.readline().strip()
if len(line) == 0:
    break
line_list = line.split("\t")
```

Line is obviously a line from the file but what is line_list? I therefore replaced line_list with room_data, connection_data, item_data, and synonym_data to represent the information on each line being split.

```
room_data = line.split("\t")
```

This chunk of code to check for items and find conditional connections is also hard to decipher:

```
for i in range(len(self.inventory)):
    if self.inventory[i].name in self.current_room.cond_connections:
        for movement in self.current_room.cond_connections[
            self.inventory[i].name]:
```

Iterating through the items feels a lot more pythonic and easy to understand what is going on.

```
for item in self.inventory:
    if item.name in self.current_room.connections:
        for movement in self.current_room.cond_connections[item.name]:
```

The same change was made for displaying the inventory in play_game and the take() and drop() methods:

```
if command == "INVENTORY":
    if not adventure.inventory:
        print("Your inventory is empty.")
        continue
    for item in adventure.inventory:
        print(item.name + ": " + item.desc)
    continue
```

```
for room_item in self.current_room.items:
    if room_item.name == item:
        self.inventory.append(room_item)
        self.current_room.items.remove(room_item)
        return item + " taken."
return "No such item."
```

I realized that the whole move method could be split into a conditional movement method and a regular movement method in order to make the code more modular.

```
def conditional_move(self, direction: str, history: History) -> bool:
    """
    Move to a different room based on conditional connections

    pre: direction is a direction to move toward: a str and history is
        a stack holding previous rooms: a History object
    post: Return True if move was successful, else False: a bool
    """

    for item in self.inventory:
        if item.name in self.current_room.cond_connections:
            for movement in self.current_room.cond_connections[item.name]:
                try:
                    previous_room = self.current_room
                    self.current_room = self.rooms[
                        int(movement[direction])]
                    history.push(previous_room)
                    return True
                except KeyError:
                    break
    return False
```

```
def move(self, direction: str, history: History) -> bool:
    """
    Move to a different room based on the given direction

    pre: direction is a direction to move toward: a str and history is
        a stack holding previous rooms: a History object
    post: Return True if move was successful, else False: a bool
    """
    if self.conditional_move(direction, history):
        return True
```

Though I should probably have done this before all the other changes to check if I broke anything; I finished by creating pytests for each of the classes.