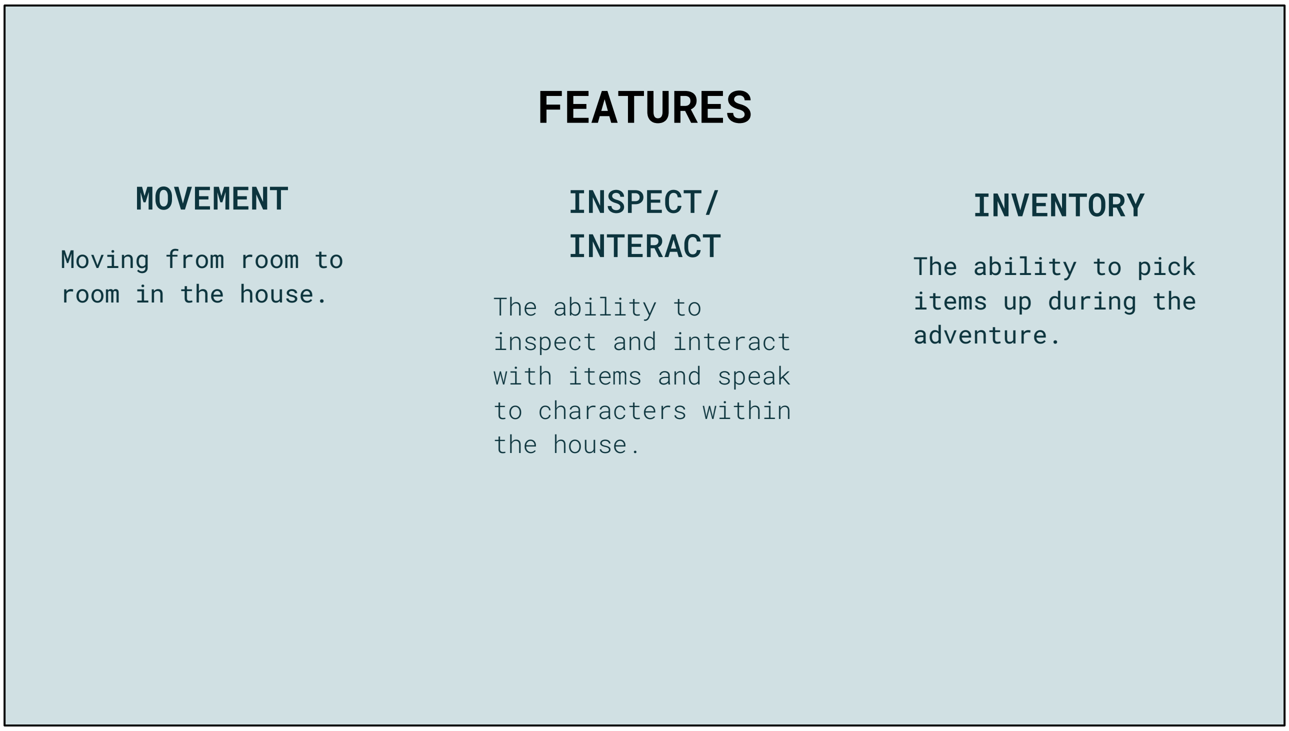
Terminal Application

Imogen Behan-Willett T1A3

Hello! Welcome to my presentation of my Terminal Application.
My name is Imogen Behan-Willett and for my terminal application I created a Text Adventure.

Walkthrough of the Terminal Application

The text adventure is set inside a haunted house. The house contains a waterlogged library, a scrawled upon study, a dilapidated entrance, a fiery kitchen, a mysterious statue room, a decaying dining room and finally a bedroom shrouded in darkness.
To escape the house the player needs to collect a candle and matches, so they can enter the dark bedroom.
Once they've achieved illumination they must answer the bedroom ghost's riddle correctly to get her key to the front locked door.
If they can answer the riddle correctly (perhaps after doing some research in the library) they receive the key to the locked door and can leave the haunted house!



FEATURES

MOVEMENT

Moving from room to room in the house.

INSPECT/ INTERACT

The ability to inspect and interact with items and speak to characters within the house.

INVENTORY

The ability to pick items up during the adventure.

This terminal application has three main features:

- **Movement:** The user can move from room to room within the house.
- **Inspection / Interaction:** The user can inspect or interact with items and characters within the game.
- **Inventory:** The user can pick items up during the adventure.

How to use the application

The Haunted House text adventure takes text inputs from the user after printing a prompt.
Prompts include:

- Type the direction you'd like to go, or type inv to check your inventory.
- Blow out candle?
- Light the candle with the matches?
- Which book would you like to read?

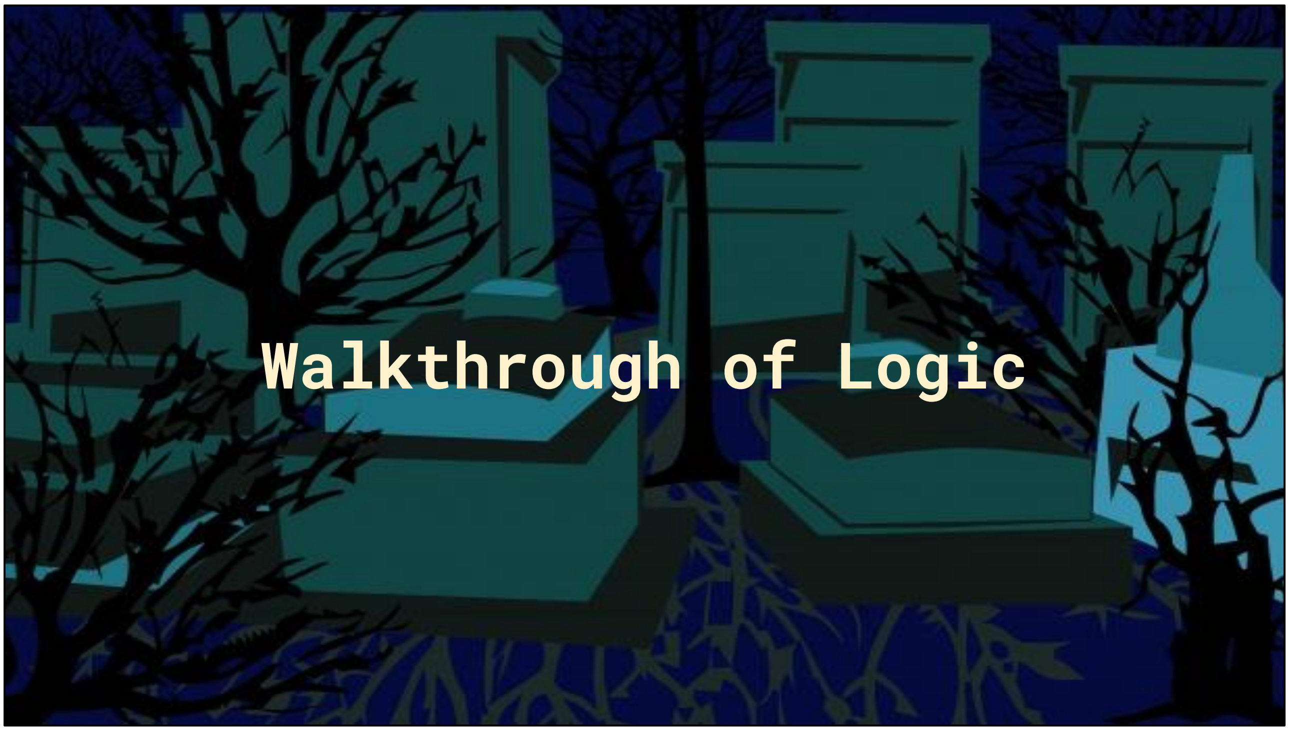
Example user inputs:

MOVEMENT	*INSPECT / INTERACT*	*INVENTORY*
north	*candle*	*inv*
south	*stove*	
east	*yes*	
west	*no*	

The Haunted House text adventure is played using text inputs from the user.
The application will use a different prompt for an user input depending on what feature is.

Movement: A typical prompt is ‘Type the direction you’d like to go’. The terminal accepts answers such as *‘north’, south’, ‘east’, ‘west’* as a valid response to this input. To interact with an item, when prompted to the user inputs the name of the item they’d like to interact with.
Interact / Inspect & Inventory: During movement, the user will be given the option to type ‘*inv*’ to see their inventory. Item interactions inputs are frequently either yes or no or the user specifying the name of the item they’d like to interact with.

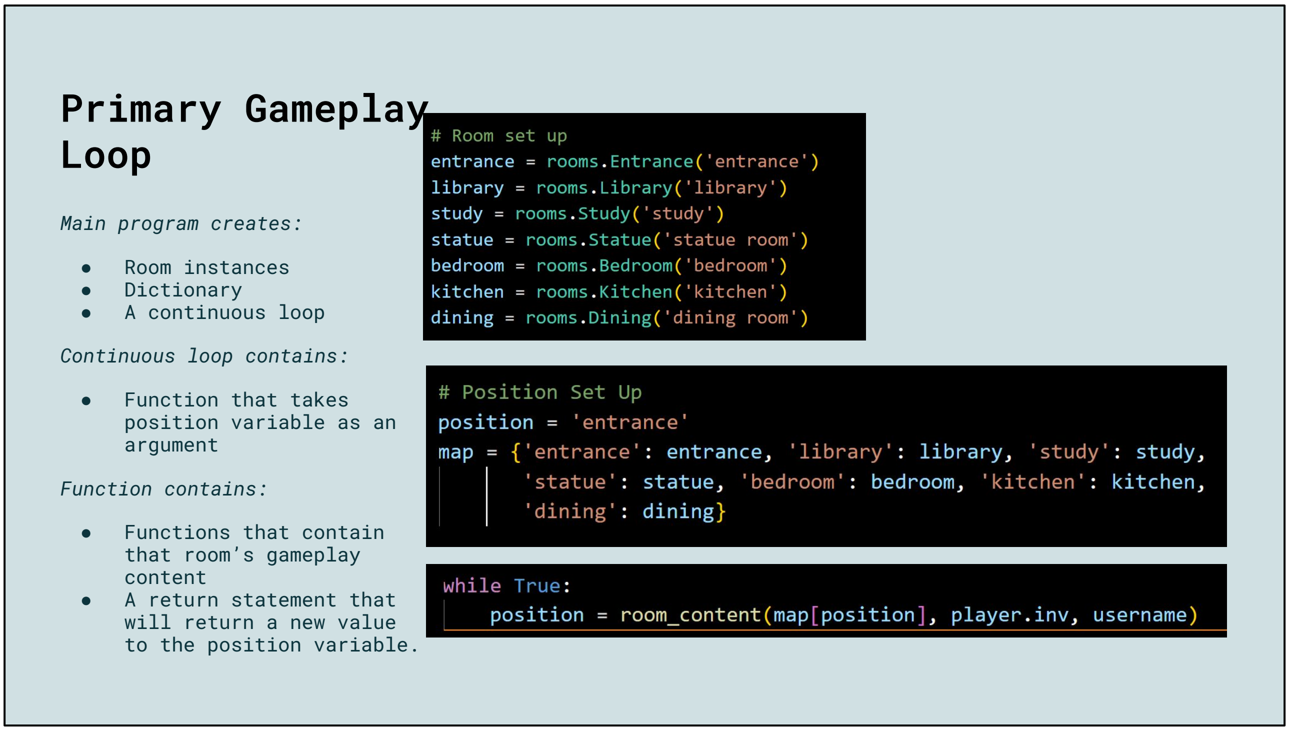
The user can quit at anytime by inputting quit.



Walkthrough of Logic

The logic of this terminal application will be broken down in four parts:

1. Primary gameplay logic
2. Logic of the rooms
3. Logic of items
4. Logic of characters



Primary Gameplay Loop

Main program creates:

- Room instances
- Dictionary
- A continuous loop

Continuous loop contains:

- Function that takes position variable as an argument

Function contains:

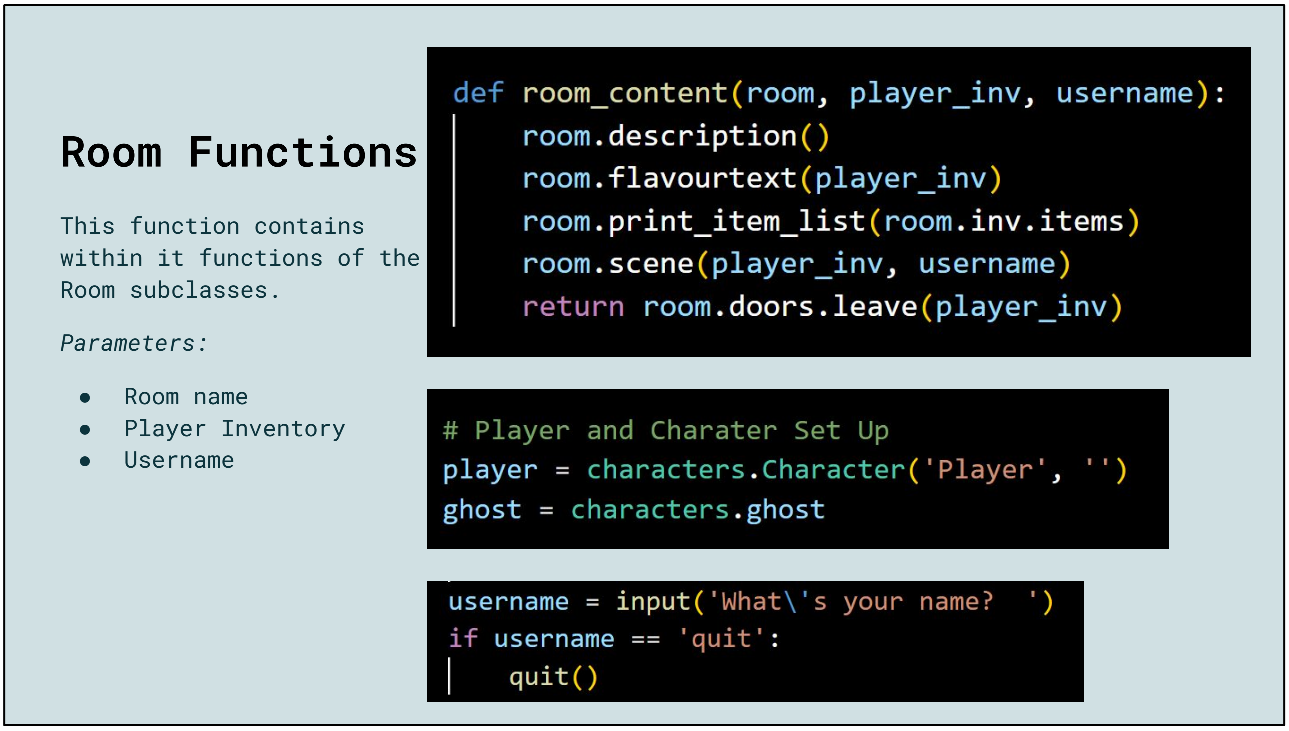
- Functions that contain that room's gameplay content
- A return statement that will return a new value to the position variable.

```python
# Room set up
entrance = rooms.Entrance('entrance')
library = rooms.Library('library')
study = rooms.Study('study')
statue = rooms.Statue('statue room')
bedroom = rooms.Bedroom('bedroom')
kitchen = rooms.Kitchen('kitchen')
dining = rooms.Dining('dining room')
```

```python
# Position Set Up
position = 'entrance'
map = {'entrance': entrance, 'library': library, 'study': study,
        'statue': statue, 'bedroom': bedroom, 'kitchen': kitchen,
        'dining': dining}
```

```python
while True:
    position = room_content(map[position], player.inv, username)
```

Other than an brief introduction, the entirety of the game is contained in one continuous While loop.
The value stored in the `position` variable dictates what room's functions called.
The main program initially stores '`entrance`' in the variable `position.`
After the various Room subclasses objects are instanced the variable name where they are stored get added to a dictionary called `map`.
These names are stored as a value with a string of the room's name being their key.
This dictionary is used by the *room_content function* to pass the particular Room object associated with the string stored in the position variable as an argument.
The *room_content* function will eventually return a new string that is stored in the position variable.
This will continue forever until either the player quits or the game is won.



Room Functions

This function contains within it functions of the Room subclasses.

Parameters:

- Room name
- Player Inventory
- Username

```python
def room_content(room, player_inv, username):
    room.description()
    room.flavourtext(player_inv)
    room.print_item_list(room.inv.items)
    room.scene(player_inv, username)
    return room.doors.leave(player_inv)
```

```python
# Player and Charater Set Up
player = characters.Character('Player', '')
ghost = characters.ghost
```

```python
username = input('What\'s your name?  ')
if username == 'quit':
    quit()
```

This function calls the Room subclasses’ functions. It takes three parameters: one for the room name, one for the player inventory and one for the username.

The room name is passed by the map dictionary as explained on the previous slide.

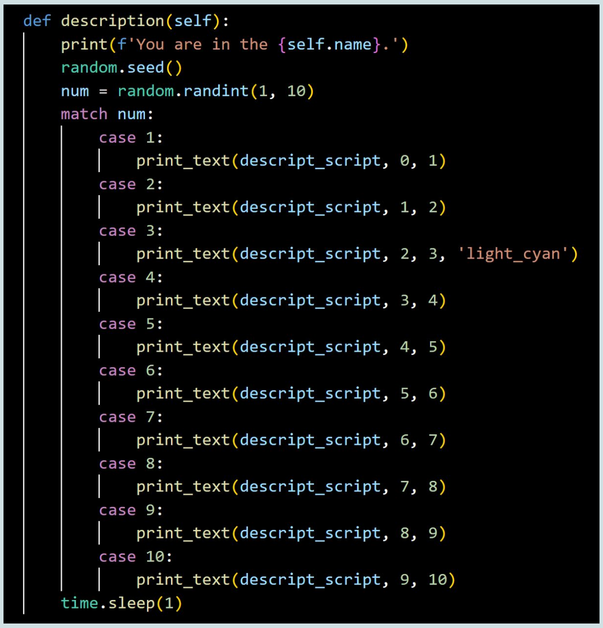
The player inventory is a instance of the Inventory that is part of the composition of the Character class. The ‘`player`’ variable stores an instance of the Character class and was created earlier in the main program.

The username is a variable storing the user’s answer to the prompt “What is your name?”

Description

First function.

- Tells player where they are.
- Prints random line of atmospheric text.

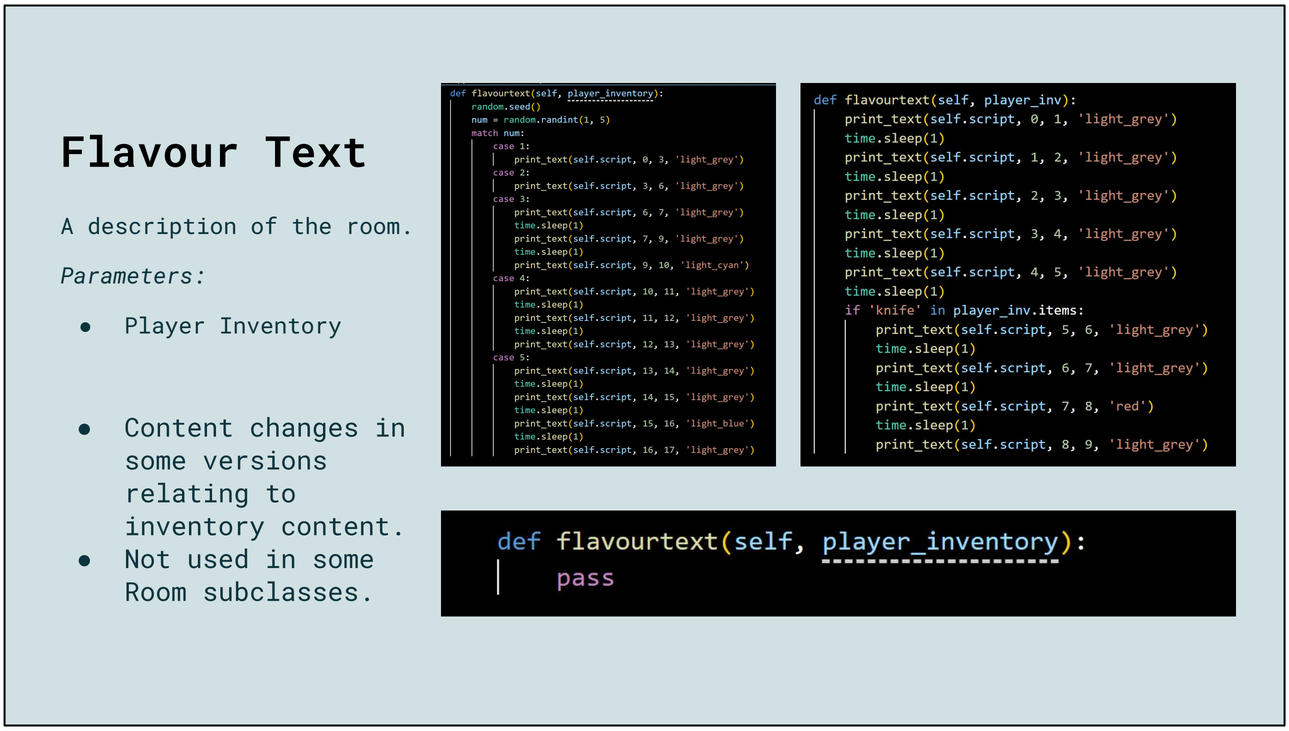


```python
def description(self):
    print(f'You are in the {self.name}.')
    random.seed()
    num = random.randint(1, 10)
    match num:
        case 1:
            print_text(descript_script, 0, 1)
        case 2:
            print_text(descript_script, 1, 2)
        case 3:
            print_text(descript_script, 2, 3, 'light_cyan')
        case 4:
            print_text(descript_script, 3, 4)
        case 5:
            print_text(descript_script, 4, 5)
        case 6:
            print_text(descript_script, 5, 6)
        case 7:
            print_text(descript_script, 6, 7)
        case 8:
            print_text(descript_script, 7, 8)
        case 9:
            print_text(descript_script, 8, 9)
        case 10:
            print_text(descript_script, 9, 10)
    time.sleep(1)
```

The description function prints the name of the room that is associated with the value stored in the `position` variable.

It runs upon either start of the game or when the player has moved into a new area. After printing the string “You are in {whatever the name of the room is}” it will then print a random text line of atmospheric narration. This is done by generating a random number and then matching that number to 10 possible match cases each of which will print it’s own unique line.

This is one of the functions stored in the abstract base class Room.



Flavour Text

A description of the room.

Parameters:

- Player Inventory

- Content changes in some versions relating to inventory content.
- Not used in some Room subclasses.

```python
def flavourtext(self, player_inventory):
    random.seed()
    num = random.randint(1, 5)
    match num:
        case 1:
            print_text(self.script, 0, 3, 'light_grey')
        case 2:
            print_text(self.script, 3, 6, 'light_grey')
        case 3:
            print_text(self.script, 6, 7, 'light_grey')
            time.sleep(1)
            print_text(self.script, 7, 9, 'light_grey')
            time.sleep(1)
            print_text(self.script, 9, 10, 'light_cyan')
        case 4:
            print_text(self.script, 10, 11, 'light_grey')
            time.sleep(1)
            print_text(self.script, 11, 12, 'light_grey')
            time.sleep(1)
            print_text(self.script, 12, 13, 'light_grey')
        case 5:
            print_text(self.script, 13, 14, 'light_grey')
            time.sleep(1)
            print_text(self.script, 14, 15, 'light_grey')
            time.sleep(1)
            print_text(self.script, 15, 16, 'light_blue')
            time.sleep(1)
            print_text(self.script, 16, 17, 'light_grey')
```

```python
def flavourtext(self, player_inv):
    print_text(self.script, 0, 1, 'light_grey')
    time.sleep(1)
    print_text(self.script, 1, 2, 'light_grey')
    time.sleep(1)
    print_text(self.script, 2, 3, 'light_grey')
    time.sleep(1)
    print_text(self.script, 3, 4, 'light_grey')
    time.sleep(1)
    print_text(self.script, 4, 5, 'light_grey')
    time.sleep(1)
    if 'knife' in player_inv.items:
        print_text(self.script, 5, 6, 'light_grey')
        time.sleep(1)
        print_text(self.script, 6, 7, 'light_grey')
        time.sleep(1)
        print_text(self.script, 7, 8, 'red')
        time.sleep(1)
        print_text(self.script, 8, 9, 'light_grey')
```

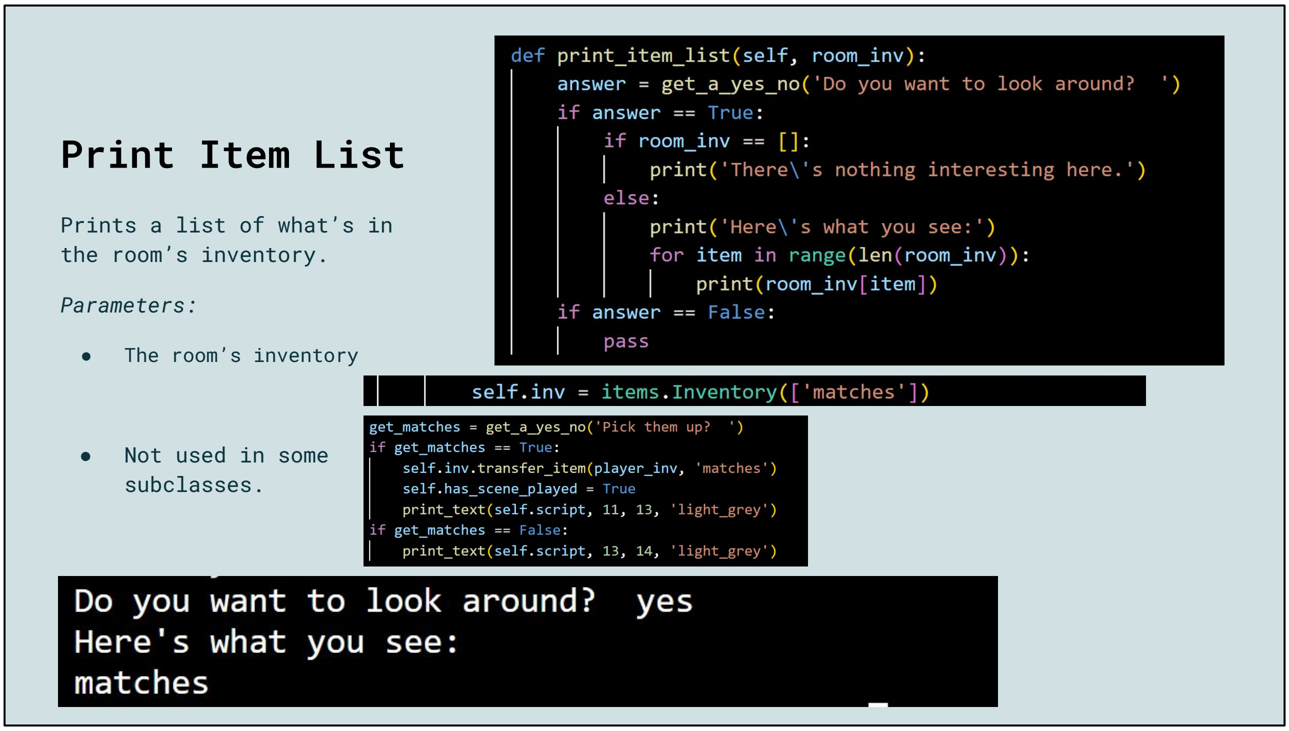
```python
def flavourtext(self, player_inventory):
    pass
```

The flavour text function describes the room in detail.
For certain room subclasses these descriptions change in relation to what the player has in their inventory. For example if you enter the study with ‘knife’ in your inventory there is extra text content. It does this by using and if else statement with the conditional being:

if ‘knife’ in player_inv.self

This is why the method takes player_inv as an argument, despite not all subclasses flavour text calling on the player’s inventory.
In the Statue subclass, the text is randomised in the same way as the description function had randomised text content.

In the Room base class, this is an abstract method.



Print Item List

Prints a list of what's in the room's inventory.

Parameters:

- The room's inventory
- Not used in some subclasses.

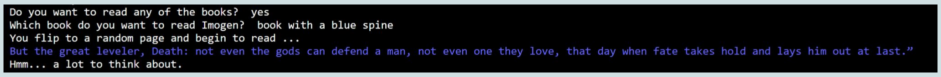
```python
def print_item_list(self, room_inv):
    answer = get_a_yes_no('Do you want to look around?  ')
    if answer == True:
        if room_inv == []:
            print('There\'s nothing interesting here.')
        else:
            print('Here\'s what you see:')
            for item in range(len(room_inv)):
                print(room_inv[item])
    if answer == False:
        pass
```

```python
        self.inv = items.Inventory(['matches'])
```

```python
get_matches = get_a_yes_no('Pick them up?  ')
if get_matches == True:
    self.inv.transfer_item(player_inv, 'matches')
    self.has_scene_played = True
    print_text(self.script, 11, 13, 'light_grey')
if get_matches == False:
    print_text(self.script, 13, 14, 'light_grey')
```

```
Do you want to look around?  yes
Here's what you see:
matches
```

Room Subclasses each have their own inventory.
For the example above, you can see the study stores its own instance an Inventory object, which itself contains the string ‘matches’.
When a player gains an items, that item is being transferred from the room’s inventory into the player’s.
This function prints what is stored in the room’s inventory.
If the inventory is empty, the function prints “There’s nothing interesting here.”
Some subclasses have nothing in their inventory, these classes simply pass the function.
Whatever is stored in the inventory is interactable, even if they’re not all members of the item class.



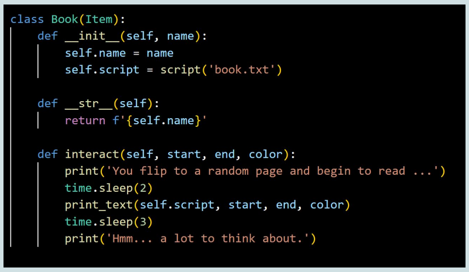
```
Do you want to read any of the books?  yes
Which book do you want to read Imogen?  book with a blue spine
You flip to a random page and begin to read ...
But the great leveler, Death: not even the gods can defend a man, not even one they love, that day when fate takes hold and lays him out at last.”
Hmm... a lot to think about.
```

Scene

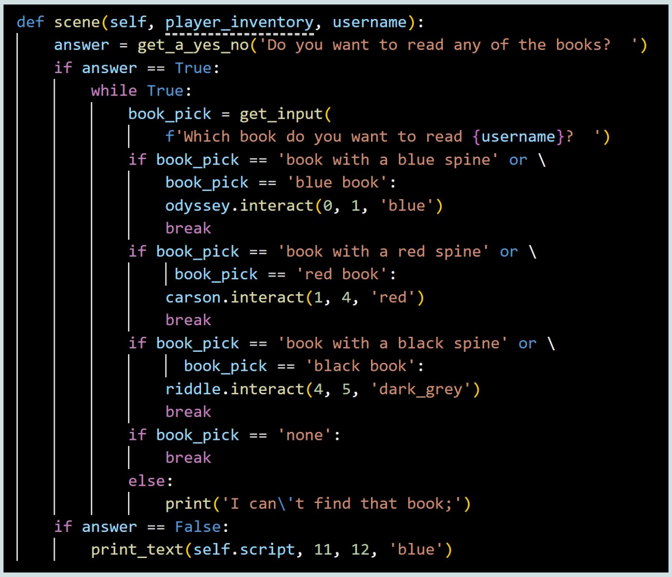
Where the narrative content of the room is stored.

Parameters:

- Player Inventory
- Username

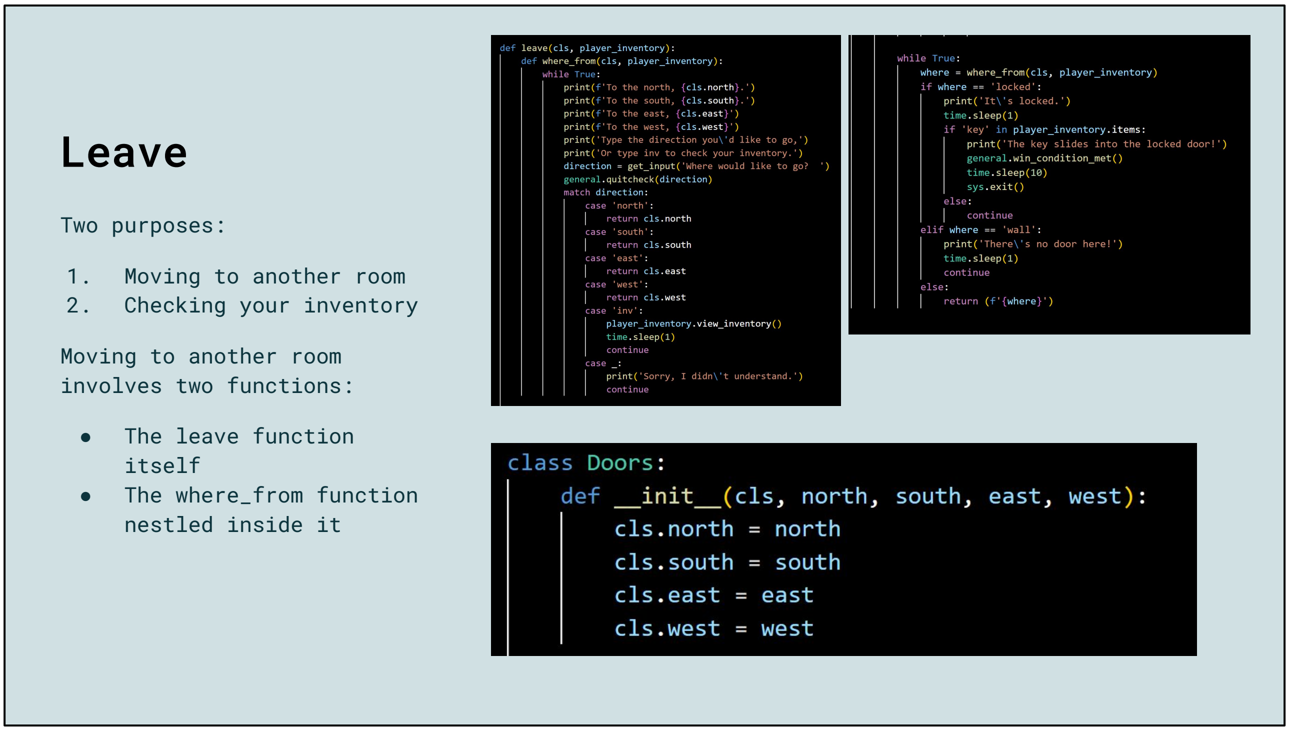


```python
class Book(Item):
    def __init__(self, name):
        self.name = name
        self.script = script('book.txt')

    def __str__(self):
        return f'{self.name}'

    def interact(self, start, end, color):
        print('You flip to a random page and begin to read ...')
        time.sleep(2)
        print_text(self.script, start, end, color)
        time.sleep(3)
        print('Hmm... a lot to think about.')
```



```python
def scene(self, player_inventory, username):
    answer = get_a_yes_no('Do you want to read any of the books?  ')
    if answer == True:
        while True:
            book_pick = get_input(
                f'Which book do you want to read {username}?  ')
            if book_pick == 'book with a blue spine' or \
                book_pick == 'blue book':
                odyssey.interact(0, 1, 'blue')
                break
            if book_pick == 'book with a red spine' or \
                 book_pick == 'red book':
                carson.interact(1, 4, 'red')
                break
            if book_pick == 'book with a black spine' or \
                  book_pick == 'black book':
                riddle.interact(4, 5, 'dark_grey')
                break
            if book_pick == 'none':
                break
            else:
                print('I can\'t find that book;')
    if answer == False:
        print_text(self.script, 11, 12, 'blue')
```

The subclasses from the base class Room were primarily created to contain their own versions of the scene method.
The scene method is where the game play and narrative content is stored.
Most scene methods across the subclasses start with asking the player what they'd like to interact with.
The player input is then used as a conditional in a if else statement.
In some of the code blocks that could be executed in the if else statement the player can get the option to pick something up, which as explained in the last slide transfers the item to the player from the room. This is why the player inventory is needed as an argument.
Other code blocks give the player an opportunity to interact with an item.
Objects of the item subclasses, sometimes stored in the room's inventory, each have their own version of an interact method.
For the Books subclass, the method prints a text line except from a book.
The conversation method from the Character class is called in one scene method.
Although other scene methods may use the username, the initial reason behind passing it as an argument was because this particular method requires it. This is the method for the Bedroom class.



Leave

Two purposes:

1. Moving to another room
2. Checking your inventory

Moving to another room involves two functions:

- The leave function itself
- The where_from function nestled inside it

```python
def leave(cls, player_inventory):
    def where_from(cls, player_inventory):
        while True:
            print(f'To the north, {cls.north}.')
            print(f'To the south, {cls.south}.')
            print(f'To the east, {cls.east}')
            print(f'To the west, {cls.west}')
            print('Type the direction you\'d like to go,')
            print('Or type inv to check your inventory.')
            direction = get_input('Where would like to go?  ')
            general.quitcheck(direction)
            match direction:
                case 'north':
                    return cls.north
                case 'south':
                    return cls.south
                case 'east':
                    return cls.east
                case 'west':
                    return cls.west
                case 'inv':
                    player_inventory.view_inventory()
                    time.sleep(1)
                    continue
                case _:
                    print('Sorry, I didn\'t understand.')
                    continue
```

```python
        while True:
            where = where_from(cls, player_inventory)
            if where == 'locked':
                print('It\'s locked.')
                time.sleep(1)
                if 'key' in player_inventory.items:
                    print('The key slides into the locked door!')
                    general.win_condition_met()
                    time.sleep(10)
                    sys.exit()
                else:
                    continue
            elif where == 'wall':
                print('There\'s no door here!')
                time.sleep(1)
                continue
            else:
                return (f'{where}')
```

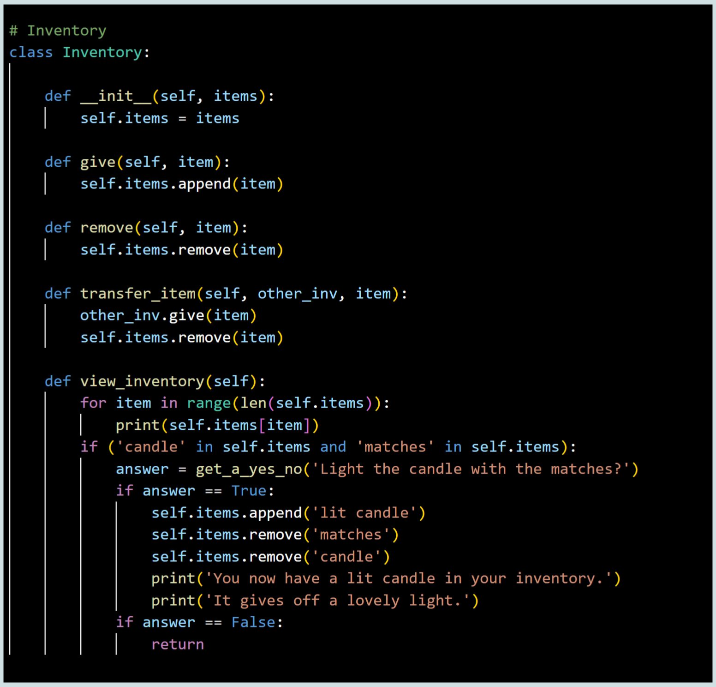
```python
class Doors:
    def __init__(cls, north, south, east, west):
        cls.north = north
        cls.south = south
        cls.east = east
        cls.west = west
```

The leave method, stored by the Room class's compositional class Doors, has two purposes.
Firstly, it gives the player a chance to check their inventory.
Secondly, it allows the player to move to another room.
The Doors class stores both the methods discussed here and the string associated with a particular direction (as in north, south, east or west).
The `where_from` method first retrieves the string associated with the direction the user typed in from the Door object inside whichever Room Class the Door object is stored in.
It does this by using a match case statement with the match variable storing the user input.
Then the leave method checks if the string is either '`locked`' or '`wall`'.
If the string is '`locked`' the leave method checks if '`key`' is in the player's inventory. If it is, it runs the `win_condition_met` function ending the game.
Otherwise, it restarts the loop. If the string is '`wall`' the loop is also restarted.
The loop will continue until it receive a valid input, which it will then return to the `room_content` function who will then return it as the new value stored in the position global variable.

Logic of Items

Quick discussion on viewing inventory:

- Prints each element in the list



```python
# Inventory
class Inventory:

    def __init__(self, items):
        self.items = items

    def give(self, item):
        self.items.append(item)

    def remove(self, item):
        self.items.remove(item)

    def transfer_item(self, other_inv, item):
        other_inv.give(item)
        self.items.remove(item)

    def view_inventory(self):
        for item in range(len(self.items)):
            print(self.items[item])
        if ('candle' in self.items and 'matches' in self.items):
            answer = get_a_yes_no('Light the candle with the matches?')
            if answer == True:
                self.items.append('lit candle')
                self.items.remove('matches')
                self.items.remove('candle')
                print('You now have a lit candle in your inventory.')
                print('It gives off a lovely light.')
            if answer == False:
                return
```

Most of the logic of the items has already been covered in this presentation. The viewing inventory method will be briefly discussed as it hasn't be covered.
The `view_inventory` method works by printing each element of the items list from 0 to the length of this list by using a for loop.

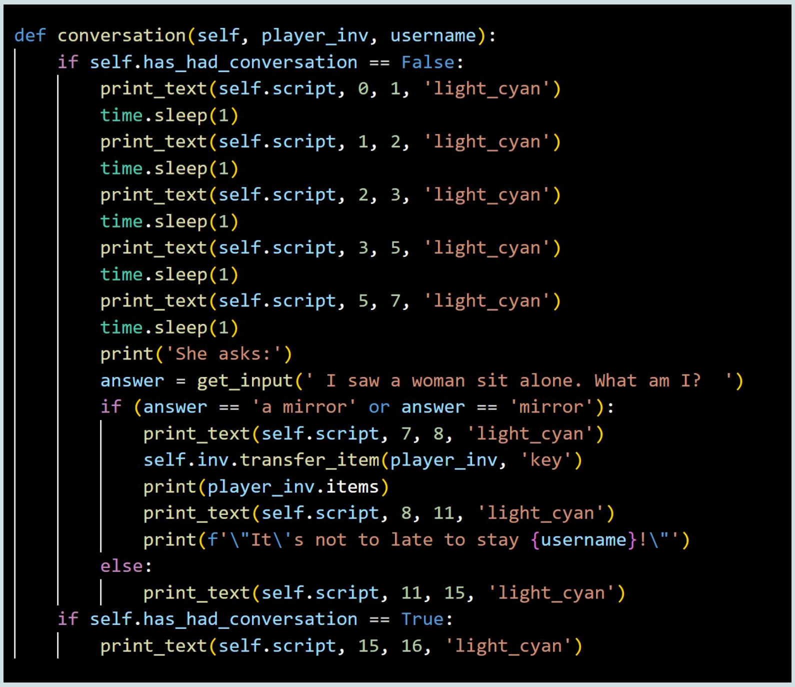
There is an if else statement also in this method, prompting players to combine two items if both are found to be in the items list.

Logic of Character

Contains conversation method.

Parameters:

- Player Inventory
- Username

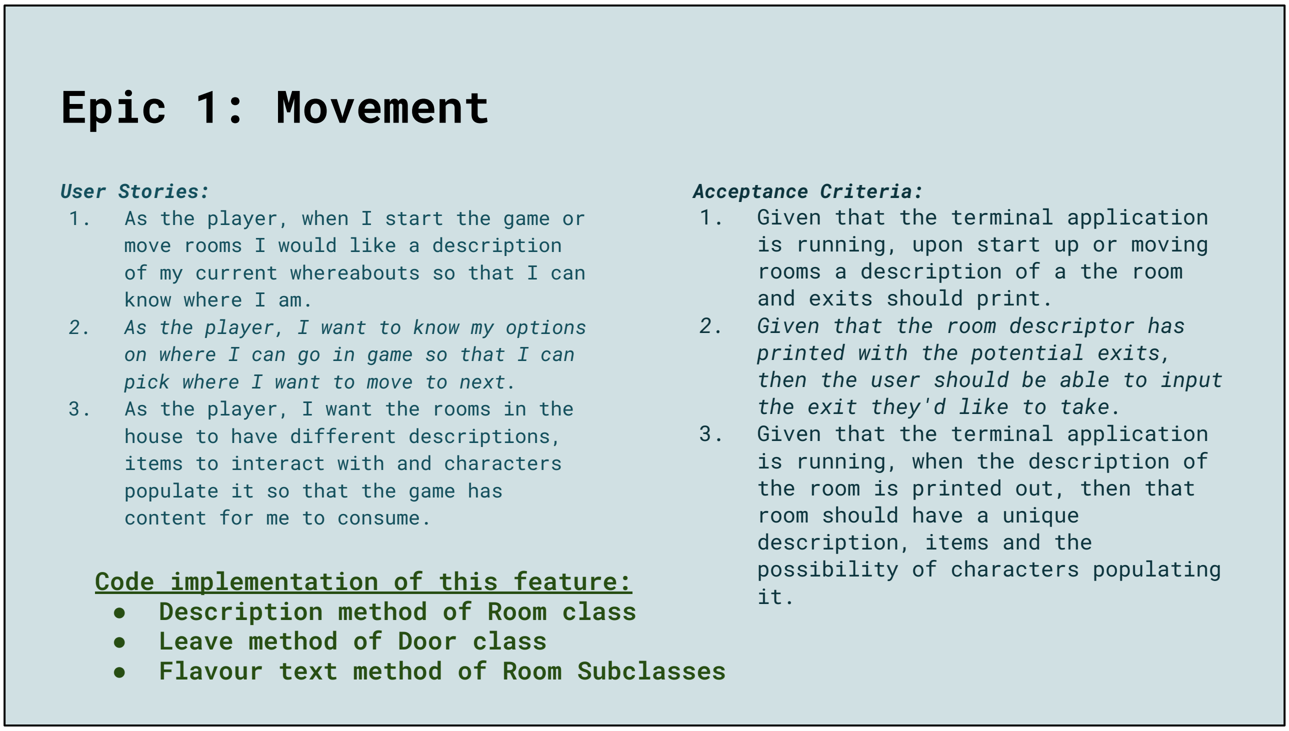


```python
def conversation(self, player_inv, username):
    if self.has_had_conversation == False:
        print_text(self.script, 0, 1, 'light_cyan')
        time.sleep(1)
        print_text(self.script, 1, 2, 'light_cyan')
        time.sleep(1)
        print_text(self.script, 2, 3, 'light_cyan')
        time.sleep(1)
        print_text(self.script, 3, 5, 'light_cyan')
        time.sleep(1)
        print_text(self.script, 5, 7, 'light_cyan')
        time.sleep(1)
        print('She asks:')
        answer = get_input(' I saw a woman sit alone. What am I?  ')
        if (answer == 'a mirror' or answer == 'mirror'):
            print_text(self.script, 7, 8, 'light_cyan')
            self.inv.transfer_item(player_inv, 'key')
            print(player_inv.items)
            print_text(self.script, 8, 11, 'light_cyan')
            print(f'\"It\'s not to late to stay {username}!\"')
        else:
            print_text(self.script, 11, 15, 'light_cyan')
    if self.has_had_conversation == True:
        print_text(self.script, 15, 16, 'light_cyan')
```

The Character class has also been discussed earlier in this presentation.
A quick overview of the logic of the remaining methods will be given.
The Character class contains the conversation method.
The conversation method is how the player receives the key. By answering the riddle correctly, the key is transferred from the ghost to the player.
The application determines if the player has answered the riddle correctly by using the user’s input as a conditional for an if else statement.

Development Plan

I'll now run you through my development plan. The first thing I did when I started my application was create four agile style epics with user stories.



Epic 1: Movement

User Stories:

1. As the player, when I start the game or move rooms I would like a description of my current whereabouts so that I can know where I am.
2. *As the player, I want to know my options on where I can go in game so that I can pick where I want to move to next.*
3. As the player, I want the rooms in the house to have different descriptions, items to interact with and characters populate it so that the game has content for me to consume.

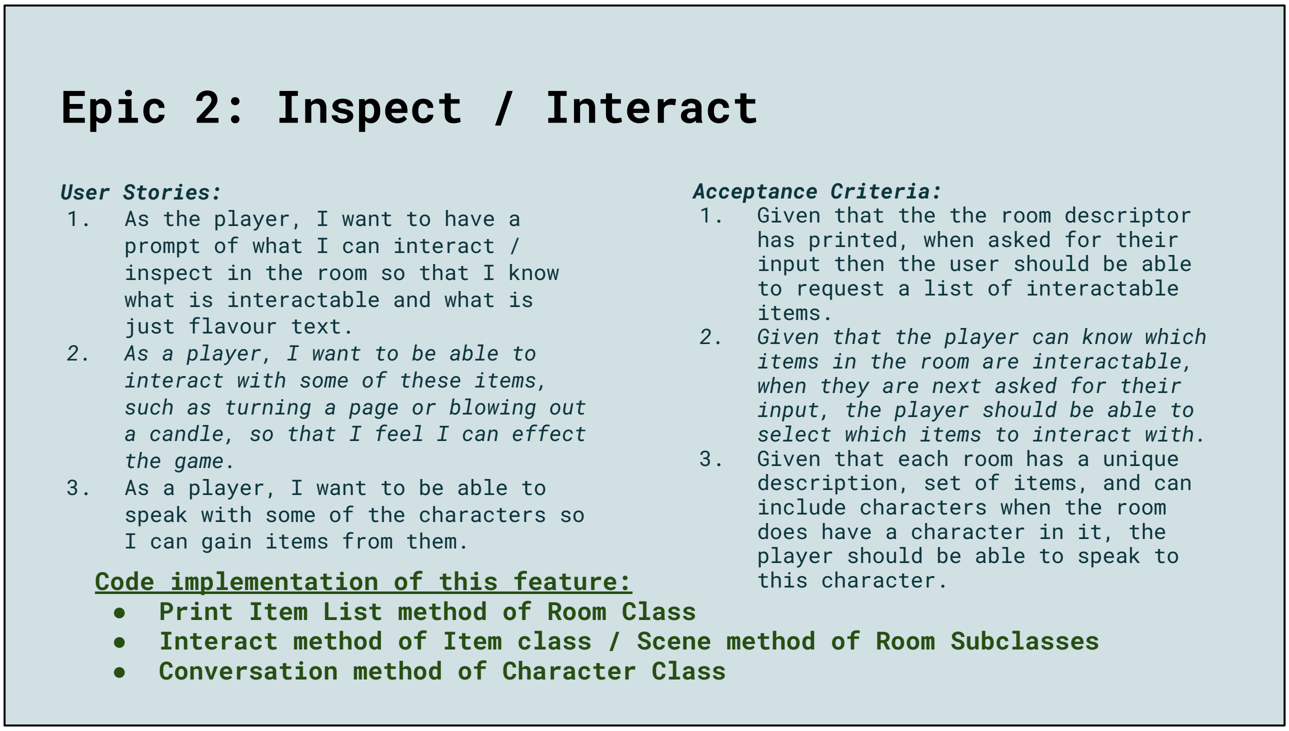
Acceptance Criteria:

1. Given that the terminal application is running, upon start up or moving rooms a description of a the room and exits should print.
2. *Given that the room descriptor has printed with the potential exits, then the user should be able to input the exit they'd like to take.*
3. Given that the terminal application is running, when the description of the room is printed out, then that room should have a unique description, items and the possibility of characters populating it.

Code implementation of this feature:

- **Description method of Room class**
- **Leave method of Door class**
- **Flavour text method of Room Subclasses**

My first epic was movement.
I wanted to get this feature done first as it was foundational to to implementation of my other features.
I couldn't implement items or characters without having a way to move from place to place to see those items or characters.



Epic 2: Inspect / Interact

User Stories:

1. As the player, I want to have a prompt of what I can interact / inspect in the room so that I know what is interactable and what is just flavour text.
2. *As a player, I want to be able to interact with some of these items, such as turning a page or blowing out a candle, so that I feel I can effect the game.*
3. As a player, I want to be able to speak with some of the characters so I can gain items from them.

Acceptance Criteria:

1. Given that the the room descriptor has printed, when asked for their input then the user should be able to request a list of interactable items.
2. *Given that the player can know which items in the room are interactable, when they are next asked for their input, the player should be able to select which items to interact with.*
3. Given that each room has a unique description, set of items, and can include characters when the room does have a character in it, the player should be able to speak to this character.

Code implementation of this feature:

- **Print Item List method of Room Class**
- **Interact method of Item class / Scene method of Room Subclasses**
- **Conversation method of Character Class**

My second epic was inspect / interact.
I planned to implement this feature second because obviously you need items prior to having an inventory.

Epic 3: Inventory

User Stories:

1. As the player, I would like to be able to pick up items in the game so that I can use them later on.
2. *As a player, I would like items in my inventory to change the game without my explicit input, so that the game feels more dynamic and interesting.*
3. As the player, I would like to explicitly use items to change the game so that I feel like I have control over the world.

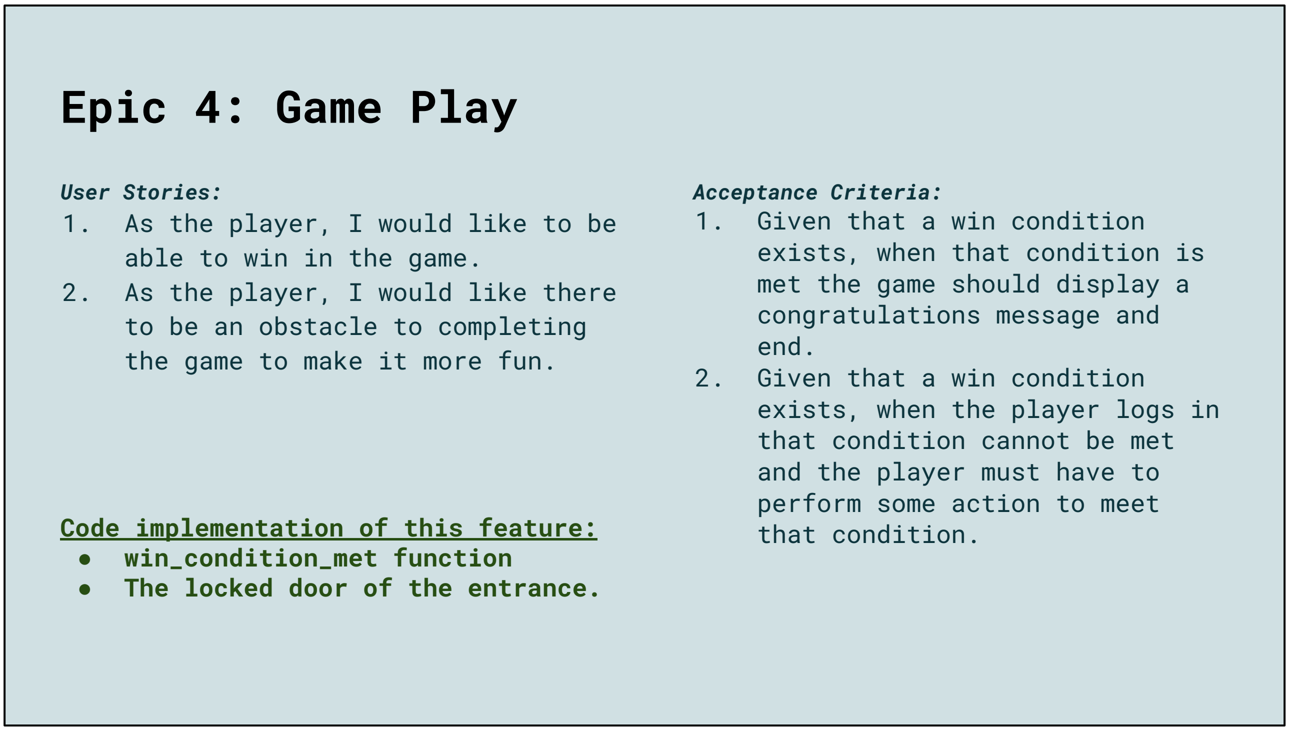
Acceptance Criteria:

1. Given that a item can be inspected, certain items should be able to be added to inventory after being inspected by the user.
2. *Given that an item is in the player's inventory, when the player enters a room the description should change without needing a player's input*
3. Given that an item is in the player's inventory, when the player is interacting with an object they should get a prompt to use the item in conjunction with the inspected item.

Code implementation of this feature:

- **Transfer Item method of inventory class**
- **Flavour Text method of Room subclasses (in particular, Study & Entrance)**
- **View Inventory method of the Inventory subclass (in particular, combining the matches and the candle)**

Thirdly, I planned to implement the inventory class.
Ideally I would have like to added a conditional statement relating to the player's inventory to all room's flavour text but I didn't have the time.



Epic 4: Game Play

User Stories:

1. As the player, I would like to be able to win in the game.
2. As the player, I would like there to be an obstacle to completing the game to make it more fun.

Acceptance Criteria:

1. Given that a win condition exists, when that condition is met the game should display a congratulations message and end.
2. Given that a win condition exists, when the player logs in that condition cannot be met and the player must have to perform some action to meet that condition.

Code implementation of this feature:

- **win_condition_met function**
- **The locked door of the entrance.**

Lastly, I wanted to implement the actual win condition and obstacle.
There are many different methods and functions surrounding the locked door that execute the win condition function if the key is in the player inventory.

Development Review

- Good colour coding my checklist rather than having each task being strictly sorted into one feature
- Great creating those epics & breaking them down into user stories.
- Getting sick early on in the development process: terrible.

A review of my development process:

- I started colour coding tasks for my Kanban board instead of sorting them into strict categories.

I'll definitely do that again, it was so useful as some tasks didn't strict fall into one feature in scope.
For example, many tasks both contributed to the inspect feature and the inventory feature.

- Starting the project by breaking what I needed to be done into epics and breaking those epics down into user stories was definitely a great idea.

It made a daunting task far more manageable and when I started coding I had a clear understanding of what I was aiming to do.

- Unfortunately I got quite unwell very early on in this project and was bed bound for a couple of days.

This made the project significantly more stressful, and it was terrible to see the deadlines I set for myself expire.
However, I was able to get back on my feet and keep going!

Thank you for your time

Thank you for your time!