python_wow Documentation

Release 0.0.6

Stanislav Kozlovski

Feb 26, 2018

Contents

CHAPTER 1

What is python_wow?

python_wow is a console turn-based RPG game written in Python 3. It is inspired by the Warcraft universe. The game works through user text commands. The goal with this pet project is to create a somewhat complex game with good code structure, tests and documentation but most important of all: to learn how to handle a project bigger than 500 lines of code. python_wow does not use pygame/rpeg intentionally, the motive being to see how far I can go writing my own RPG logic.

Contents:

1.1 Help

Need help or have any questions regarding the projects?

Feel free to e-mail: familyguyuser192@windowslive.com

1.2 Basics

For placeholders, this article will use these symbols { }

1.3 Starting out

When first starting the game, you're prompted to create a new character. Immediately after creation, you're popped into the world and the monsters are shown. You have the choice to type "?" to see all available commands or engage an attack on a monster with

    engage {monsterName}

You swing at the monster using the attack

command and all is well. Once you kill the monster, loot drops from it.

Loot dropped: 3 gold

Wolf Meat - Miscellaneous Item

Wolf Pelt - Miscellaneous Item

Strength Potion - Potion (Increases strength by 15 for 5 turns.)

You can take specific items with the take {itemName}

command, take everything with the take all

command or simply exit the menu, using the (you guessed it) exit

command.

But enough about the action of playing the game, we can go on forever with that, let's see how it works under the hood.

Creating/Loading a character

1.4 Creating/Loading a character

We get the character, which is an object of the class Character (more on that later) with this function in our main.py file:

```
from start_game_prompt import get_player_character
main_character = get_player_character()
```

In the start_game_prompt, we handle user input to see if we want to load or create a new character.

1.5 Creating a character

The function, stripped down to it's essentials:

```
def handle_create_character() -> Character:
    class_choice = str.lower(input())

    while class_choice not in AVAILABLE_CLASSES:  # check for valid class
        class_choice = str.lower(input())

    character_name = input()

    if class_choice == 'paladin':
        character = Paladin(name=character_name)

    return character
```

Really really straightforward, what we do is create a new Paladin object with default values. Part of the Paladin constructor:

```
def __init__(self, name: str, level: int = 1, health: int = 12, mana: int = 15,␣
↪strength: int = 4):
```

1.6 Loading a character

We do this by calling the load_saved_character function from the loader.py file.:

```
from loader import load_saved_character, load_all_saved_characters_general_info
character = load_saved_character(character_name)
```

I think the docstring to the method explains things fairly well:

```
"""
This function loads the information about a saved chacacter in the saved_character DB␣
↪table.

   name,   class,  level,  loaded_scripts_ID,  killed_monsters_ID, completed_quests_
↪ID, inventory_ID, gold
   Netherblood, Paladin,     10,                 1,                    1,           ␣
↪      1,            1,    23

The attributes that end in ID like loaded_scripts_ID are references to other tables.

For more information:
https://github.com/Enether/python_wow/wiki/How-saving-a-Character-works-and-
↪information-about-the-saved_character-database-table.
"""
```

In steps:

1. We query the database and save the IDs used for sub-tables:

   ```
   sv_char_reader = cursor.execute("SELECT * FROM saved_character WHERE name = ?",␣
   ↪[name]).fetchone()
   char_loaded_scripts_ID = sv_char_reader[DBINDEX_SAVED_CHARACTER_LOADED_SCRIPTS_
   ↪TABLE_ID]
   char_killed_monsters_ID = sv_char_reader[DBINDEX_SAVED_CHARACTER_KILLED_MONSTERS_
   ↪ID]
   char_completed_quests_ID = sv_char_reader[DBINDEX_SAVED_CHARACTER_COMPLETED_
   ↪QUESTS_ID]
   char_equipment_ID = sv_char_reader[DBINDEX_SAVED_CHARACTER_EQUIPMENT_ID]
   char_inventory_ID = sv_char_reader[DBINDEX_SAVED_CHARACTER_INVENTORY_ID]
   ```

2. **Using the IDs, we call a function associated with each sub-table.**

   - load_saved_character_loaded_scripts returns a set, containing the name of special in-game scripts that the character has already seen, because we do not want him to see them again.
   - load_saved_character_killed_monsters returns a set, containing the unique GUID for every special monster that the character has killed. Only monsters that should be killed once in the game are added here.
   - load_saved_character_completed_quests returns a set, containing the names of the character's completed quests. This, like the previous two, is stored so as to not load the quests in the game again.
   - load_saved_character_inventory returns a dictionary, holding the inventory of the player as it is stored in the Character class, Key: item_name, Value: tuple(object of class Item, int item_count)
   - load_saved_character_equipment returns a dictionary, holding the equipment of the player as it is stored in the Character class. Key: the equipment's slot e.g: "Shoulderpad", Value: an object of class Item

In the DB, the actual equipment's value is stored as the item's ID. In the function, we use a list comprehension to convert all the loaded IDs into objects of class Item:

```
saved_equipment_info = [load_item(id) if id is not None else None for id in␣
↪saved_equipment_info]
```

1.7 Saving a character

A character is saved in one of three scenarios:

1. He dies and given the choice to revive, the user declines.
2. The user types in the *save* command
3. The user quits the game in the conventional way, using Ctrl-C from the command line.

Saving a character is handled by the *save_character* command in the *models/characters/saver.py* file. There, we generate IDs for the saved character sub-tables or load them from the DB, if the character has been saved before.

We save the character's info in the main table:

```
char_to_save = SavedCharacterSchema(name=character.name, character_class=character_
↪class, level=character_level, gold=character_gold,
                                    scripts_id=character_loaded_scripts_id, monsters_
↪id=character_killed_monsters_id,
                                    quests_id=character_completed_quests_id,␣
↪inventory_id=character_inventory_id,
                                    head_id=headpiece_id, shoulder_id=shoulderpad_id,␣
↪necklace_id=necklace_id,
                                    chestguard_id=chestguard_id, belt_id=belt_id,␣
↪bracer_id=bracer_id,
                                    gloves_id=gloves_id, leggings_id=leggings_id,␣
↪boots_id=boots_id)
session.add(char_to_save)
session.commit()
```

It is worth noting that before inserting rows into the database, each function calls the *delete_rows_from_table*:

```
def delete_rows_from_table(table_name: str, id: int):
"""
This function will delete every row in TABLE_NAME with an id of ID
:param table_name: a string -> "saved_character_loaded_scripts" for example
:param id:  the id of the rows we want to delete -> 1

The function is used whenever we want to save new information. To save the new␣
↪updated information, we have to
delete the old one first.
"""
if table_name in ALLOWED_TABLES_TO_DELETE_FROM:
    session.query(ALLOWED_TABLES_TO_DELETE_FROM[table_name]).filter_by(id=id).delete()
    session.commit()
else:
    raise Exception(f'You do not have permission to delete from the {table_name}␣
↪table!')
```

Finally, we save each sub-table:

```
save_loaded_scripts(character_loaded_scripts_ID, character.loaded_scripts)
save_killed_monsters(character_killed_monsters_ID, character.killed_monsters)
save_completed_quests(character_completed_quests_ID, character.completed_quests)
save_inventory(character_inventory_ID, character.inventory)
```

The functions in there are pretty straightforward, the Character class has sets for the scripts he's loaded, special monsters he's killed, quests he's completed and inventory he has. In the functions above, we simply iterate through the sets and insert a row for each value.

Next:

Character Basics

CHAPTER 2

Indices and tables

- genindex
- modindex
- search